

Tools Overview

The simplest way to create and edit Connection Methods is to use the **Connection Editor**, which allows you to make selections and enter settings through a graphical interface. The **Connection Editor** translates your selections and input into the appropriate command line options to the *debug server* that MULTI uses to connect to your specific target. Connection Methods created using the **Connection Editor** are referred to as *Standard Connection Methods*. Standard Connection Methods can be saved, invoked quickly using the **Connection Chooser**, and managed using the **Connection Organizer**. For more information about creating, configuring, and using Standard Connection Methods, see “Standard Connection Methods” on page 42.



Tip

Users who prefer to enter command line options rather than use the GUI interface can create *Custom Connection Methods*. Like Standard Connection Methods, Custom Connection Methods can be saved, invoked quickly using the **Connection Chooser**, and managed using the **Connection Organizer**. For more information about using Custom Connection Methods, see “Custom Connection Methods” on page 47.

After you have created one or more Connection Methods, you can connect to your target from the **Connection Chooser** or **Connection Organizer**. You can use the **Connection Organizer** to save Connection Methods and organize them into Connection Files in your projects.

Standard Connection Methods

The following sections explain how to use MULTI to create, configure, save, edit, and use Standard Connection Methods.

Creating a Standard Connection Using the Project Wizard

If you use the **Project Wizard** to create a project, one or more Standard Connection Methods are created for you automatically. The new project contains a **default.con** Connection File that contains these Methods. (The **default.con** file is located in the **Target Resources** project in the Project Manager.) You may need to use the **Connection Editor** to edit the settings of automatically created Connection Methods

before you can use them to connect to your target. For instructions about how to edit new or existing Standard Connection Methods, see “Configuring a Standard Connection with the Connection Editor” on page 44.

Creating a Standard Connection Using the Connection Chooser

You can create a new Standard Connection Method using the **Connection Chooser**. To do so, perform the following steps.

1. Open the **Connection Chooser**.

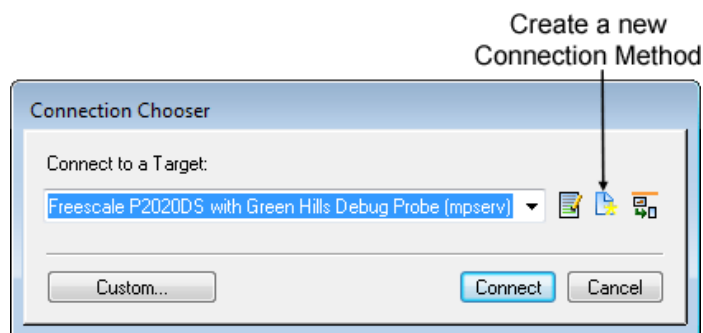
You can open the **Connection Chooser** from the MULTI Project Manager, Debugger, or Launcher. Perform one of the following steps to open the **Connection Chooser**:

- From the MULTI Launcher, click  and select **Connect**, or select **Components** → **Connect**.
- From the MULTI Project Manager, click , or select **Connect** → **Connect**.
- From the MULTI Debugger, click , or select **Target** → **Connect**.

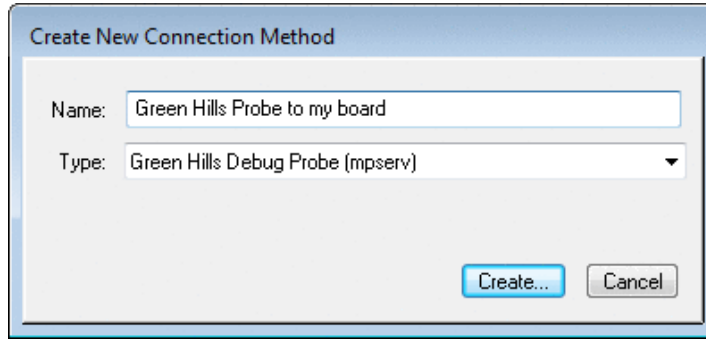


Note

In a native environment, opening the **Connection Chooser** from the Debugger automatically connects to a native debug server. Instead, open the **Connection Editor** by selecting **Method** → **New** in the **Connection Organizer**.



2. Click  to open the **Create New Connection Method** dialog box.



3. Enter a name for your Connection Method (for example, `Green Hills Probe to my board`). If you do not enter a name, the Method you create is a Temporary Connection Method (see “Temporary Connection Methods” on page 49).
4. Select an appropriate connection type from the drop-down list. The types listed depend on your particular Compiler installation. Select the type that best describes your debug connectivity method.
5. Click **Create**. The Connection Method is stored in the **[User Methods]** file. For more information, see “The Default Connection File [User Methods]” on page 53.

When you click **Create**, a **Connection Editor** window opens for your new Standard Connection Method. You may need to edit the new Method before you use it for the first time. The next section describes how to do this.

Configuring a Standard Connection with the Connection Editor

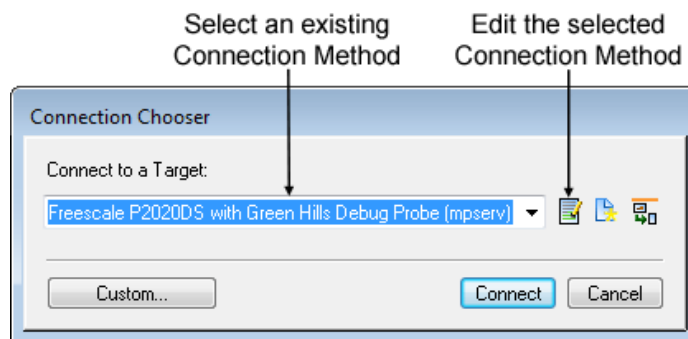
Before you can use a Standard Connection Method for the first time, you may need to configure it for your specific target system and debugging options. You can use the **Connection Editor** to configure a Standard Connection Method.

Opening the Connection Editor

If you create a new Standard Connection Method from the **Create New Connection Method** dialog box, the **Connection Editor** appears automatically when you click **Create**.

If you used the **Project Wizard** to create a project and you need to edit a default Connection Method created by the wizard, or if you want to edit a Standard Connection Method you have previously created and saved, perform the following steps to open the **Connection Editor**.

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 43).



2. From the drop-down list, select the Connection Method you want to edit.
3. Click  to open the **Connection Editor** for the selected Connection Method.

For information about the basic features of the **Connection Editor**, see the documentation about the Connection Editor in the *MULTI: Configuring Connections* book.

Connecting with a Standard Connection

After you have created and configured a Connection Method, you can use it to connect to your target from the MULTI Launcher, the MULTI Project Manager, the MULTI Debugger, the **Connection Chooser**, the **Connection Editor**, or the **Connection Organizer**, as described in the following table.

From the	Perform These Steps
MULTI Launcher	Click the  button, and select the Connection Method you want to use.
MULTI Project Manager	Select Connect → Connect , or click  to open the Connection Chooser . Use the Connection Chooser to connect to your target. (See Connection Chooser below.)

From the	Perform These Steps
MULTI Debugger	Select Target → Connect , or click  to open the Connection Chooser . Use the Connection Chooser to connect to your target. (See Connection Chooser below.) In a native environment, the Debugger automatically connects to a native debug server without opening the Connection Chooser .
Connection Chooser	Select the Connection Method from the drop-down list, and click Connect .
Connection Editor	Click the Connect button if it is available. If the Connect button appears dimmed, click OK and use the Connection Chooser to connect to your target. (See Connection Chooser above.)
Connection Organizer	Select a Connection Method. Then select Method → Connect to Target .

If your attempt to connect is unsuccessful, MULTI displays diagnostic information to help you understand the problem. The amount of the diagnostic information that MULTI can provide depends on the nature of your target. When you know what changes you need to make to your Connection Method, you can use the **Connection Editor**. For general information about using the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 44. For more information about diagnosing connection problems and for more information about the **Connection Editor** settings available for:

- INTEGRITY run-mode target connections, see Chapter 4, “INDRT2 (rtserv2) Connections” on page 59 or Chapter 5, “INDRT (rtserv) Connections” on page 77.
- Green Hills Probe or SuperTrace Probe target connections, see the *Green Hills Debug Probes User's Guide*.
- Other target connections, see the *MULTI: Configuring Connections* book for your target processor.

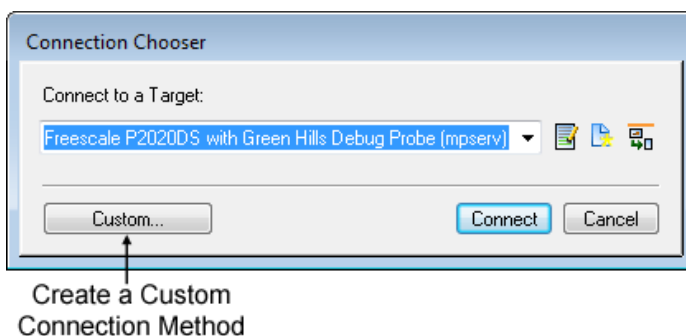
Custom Connection Methods

To create a Custom Connection Method, do one of the following:

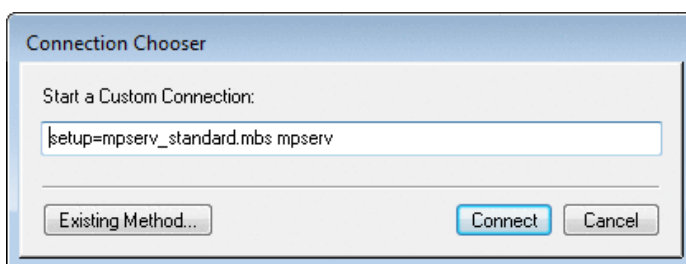
- Use the **connect** Debugger command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- Choose **Custom** as the connection type in the **Create New Connection Method** dialog box. For remaining steps about how to create a Custom Connection Method from the **Create New Connection Method** dialog box, see the instructions that follow the appearance of this dialog box in “Creating a Standard Connection Using the Connection Chooser” on page 43.
- Manually enter commands and options into the **Connection Chooser**. For more information, see below.

To create a Custom Connection Method from the **Connection Chooser**, perform the following steps:

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 43).



2. Click **Custom**.



3. In the **Start a Custom Connection** text field, enter the connection command for your particular target connection.

The general format of the command that starts a debug server and connects to a target is:

```
[setup=setup_script] xserv required_arguments ... [options]...
```

where:

- *setup_script* is the filename of the target setup script written in the MULTI scripting language. *setup=setup_script* may not be required for your target.
- *xserv* is the name of the Green Hills debug server that supports your debugging interface, monitor, or simulator. For example, the **mpserv** debug server supports the Green Hills Probe. For the name of the correct debug server, see the *MULTI: Configuring Connections* book for your target processor.
- *required_arguments* and available *options* for:
 - INTEGRITY run-mode target connections are documented in Chapter 4, “INDRT2 (rtserv2) Connections” on page 59 and Chapter 5, “INDRT (rtserv) Connections” on page 77.
 - Green Hills Probe or SuperTrace Probe target connections are documented in the *Green Hills Debug Probes User's Guide*.
 - Other target connections are documented in the *MULTI: Configuring Connections* book for your target processor.

4. Click **Connect**.

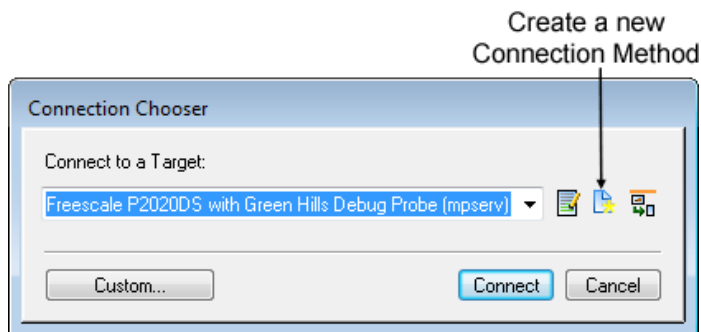
MULTI connects to your target and saves your Custom Connection Method. In the future, the Method you entered in the text field will appear in the list of Connection Methods available in the **Connection Chooser** and **Connection Organizer**.

For more information about using and managing existing Connection Methods, see “Using the Connection Organizer” on page 50. To troubleshoot a connection problem, see the *MULTI: Configuring Connections* book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User's Guide*.

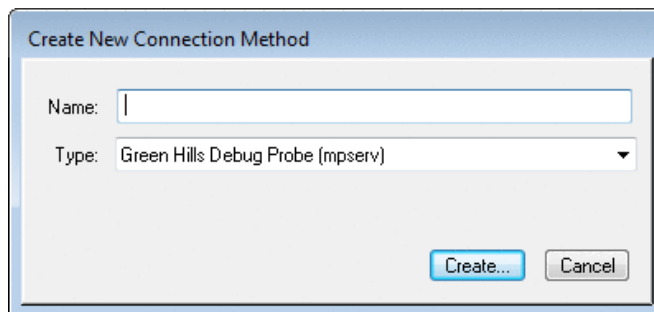
Temporary Connection Methods

You can create a connection that only exists for your current MULTI session, even if you save the file containing it, by creating a *Temporary Connection Method*. Temporary Connection Methods can be either Standard or Custom Connection Methods. To create a Temporary Connection Method, perform the following steps:

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 43).



2. Click  to open the **Create New Connection Method** dialog box.
3. Leave the name field blank, and select the appropriate connection type from the drop-down list. To create a Custom Connection Method, select **Custom** from the drop-down list.



4. Click **Create**.
5. The new Connection Method is named `Temporary Connection (#)`. Use the **Connection Editor** to configure the connection. For information about the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 44.

To convert a Temporary Connection Method to a permanent Standard or Custom Connection Method, open the **Connection Organizer**, right-click the Temporary Connection Method you want to change, and select **Make Permanent** from the shortcut menu. The method is saved and the name is changed to `Connection (#)`.

Using the Connection Organizer

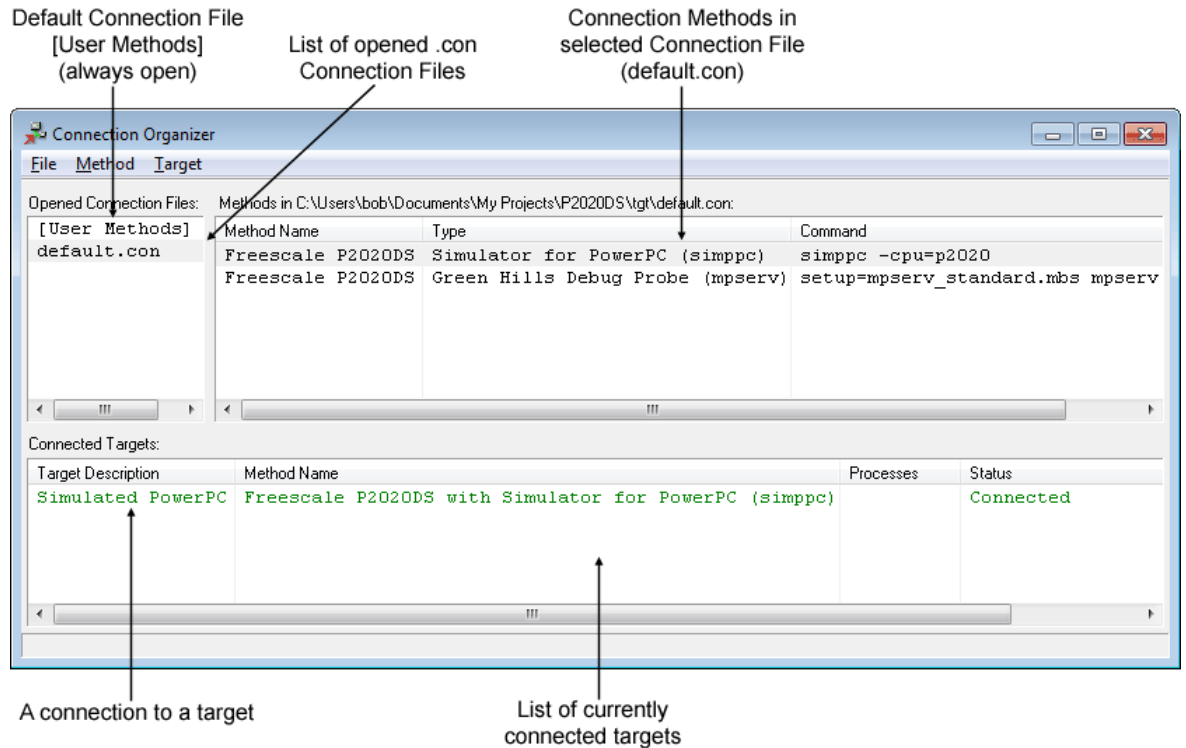
The **Connection Organizer** allows you to create, edit, copy, load, and save Connection Methods and connect to your target using Connection Methods. The following sections describe how to use the various features of the **Connection Organizer**. For more information about Connection Methods, see “Working with Connection Methods” on page 40.

Opening the Connection Organizer

You can open the **Connection Organizer** in any of the following ways:

- From the Launcher, click  and select **Open Connection Organizer**, or select **Components** → **Open Connection Organizer**.
- From the Project Manager, select **Connect** → **Connection Organizer**.
- From the Debugger, select **Target** → **Show Connection Organizer**.
- From the Debugger command pane, enter the **connectionview** command. For information about the **connectionview** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- From the **Connection Chooser** dialog box, click . If the **Connection Chooser** dialog box automatically appeared as the result of an action you took (such as trying to run your program without first connecting), clicking the  button aborts the action and opens the **Connection Organizer**.

A sample **Connection Organizer** follows.



The **Connection Organizer** has three sections:

- **Opened Connection Files** — Lists all the Connection Files currently open. Each Connection File contains one or more Connection Methods. You can use Connection Files to save, restore, and transport your connection configurations. The **Connection Organizer** always has at least one Connection File open: the **[User Methods]** file. The **Opened Connection Files** list may also contain **default.con** files created by the **Project Wizard**, or it may contain other **.con** files you have created. For more detailed information about Connection Files, see “Creating and Managing Connection Files” on page 52.
- **Methods in selected file** — Lists all the Connection Methods present in the Connection File that is selected in the **Opened Connection Files** list. You can use the **Connection Organizer**'s menu choices and shortcuts to start, copy, edit, move, or delete Connection Methods from this list.
- **Connected Targets** — Lists all the currently established target hardware or simulator connections. For more detailed information about connected targets, see “Managing Your Connected Targets” on page 54.

Creating a Connection Method

You can create a new Connection Method from the **Connection Organizer**. To do this, select **Method** → **New** from the menu bar. The **Create New Connection Method** dialog box appears. For remaining steps, see the instructions that follow the appearance of this dialog box in “Creating a Standard Connection Using the Connection Chooser” on page 43. The Connection Method is stored in the file selected in the **Connection Organizer's Opened Connection Files** list.

Editing a Connection Method

To edit a previously saved Connection Method from the **Connection Organizer**:

1. From the **Opened Connection Files** list, select the Connection File that contains your desired Connection Method.
2. From the **Methods in selected file** list, select a Connection Method.
3. From the **Connection Organizer's** menu, select **Method** → **Edit** or right-click the selected method and select **Edit** from the shortcut menu.
4. Using the **Connection Editor** that appears, modify your Connection Method settings. For general information about using the **Connection Editor**, see the documentation about the Connection Editor in the *MULTI: Configuring Connections* book. For detailed information about the settings available for:
 - INTEGRITY run-mode target connections, see Chapter 4, “INDRT2 (rtserv2) Connections” on page 59 or Chapter 5, “INDRT (rtserv) Connections” on page 77.
 - Green Hills Probe or SuperTrace Probe target connections, see the *Green Hills Debug Probes User's Guide*.
 - Other target connections, see the *MULTI: Configuring Connections* book for your target processor.
5. Click **OK** to save your changes.

Creating and Managing Connection Files

You can save one or more Connection Methods in a Connection File, which is a regular text file that usually ends with a **.con** extension. The Connection Methods stored in a Connection File are self-contained and portable. As a result, you can

open and use Connection Files created by other installations. You can even email Connection Files to other users and computers.

You can open any number of Connection Files in the **Connection Organizer**. Some are opened for you automatically. The default Connection File, **[User Methods]** is always open. If the **Connection Chooser** appears, the **Connection Organizer** also opens the Connection File that contains the Connection Method selected by default in the **Connection Chooser**.

To view the Connection Methods stored in an open Connection File, select the Connection File in the **Opened Connection Files** list.

By default, changes to open Connection Files are saved immediately. If you change the default setting, you must manually save your Connection Files via the **Connection Organizer**. For information about changing the default settings, see the **autoSaveConnectionsInFiles** and **autoSaveUserConnections** options in the “Session Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

The Default Connection File [User Methods]

The first file in the **Opened Connection Files** list is always the **[User Methods]** file. It cannot be closed or renamed. This file serves as a default location for new Connection Methods when you do not explicitly create a Connection File for them. The contents of **[User Methods]** are stored in your Green Hills user directory as **multiconnections.con**. By default, changes to **[User Methods]** are immediately saved. This behavior is configured separately from automatically saving other Connection Files. For more information, see the **autoSaveUserConnections** option in “Session Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Connection Files in a Project

You can use the MULTI Project Manager to add Connection Files with the **.con** extension to your **.gpj** projects. By default, the **Connection Organizer** automatically opens any Connection Files in a project when the Project Manager opens that project. You can select the Connection File from the **Opened Connection Files** list and work with the Connection Methods that it contains.

To add a Connection File to a project:

1. Open the Project Manager, select **File** → **Open Project**. Select the filename of the project you want to work with.
2. Select **Edit** → **Add File into *project.gpj***. Select the filename of the Connection File you want associated with this project. You can specify a Connection File that does not exist; the **Connection Organizer** simply creates it for you.

Connecting from the Connection Organizer

To connect to your target, you can use any appropriate Connection Method listed in the **Connection Organizer**. To connect from the **Connection Organizer**:

1. From the **Opened Connection Files** list, select the Connection File that contains your desired Connection Method.
2. From the **Methods in *selected file*** list, select a Connection Method.
3. From the **Connection Organizer** menu bar, select **Method** → **Connect to Target**, or right-click the method and select **Connect to Target** from the shortcut menu.
4. If the connection is successful, a new connection appears in the **Connected Targets** list. For information about connected targets, see “Managing Your Connected Targets” on page 54.

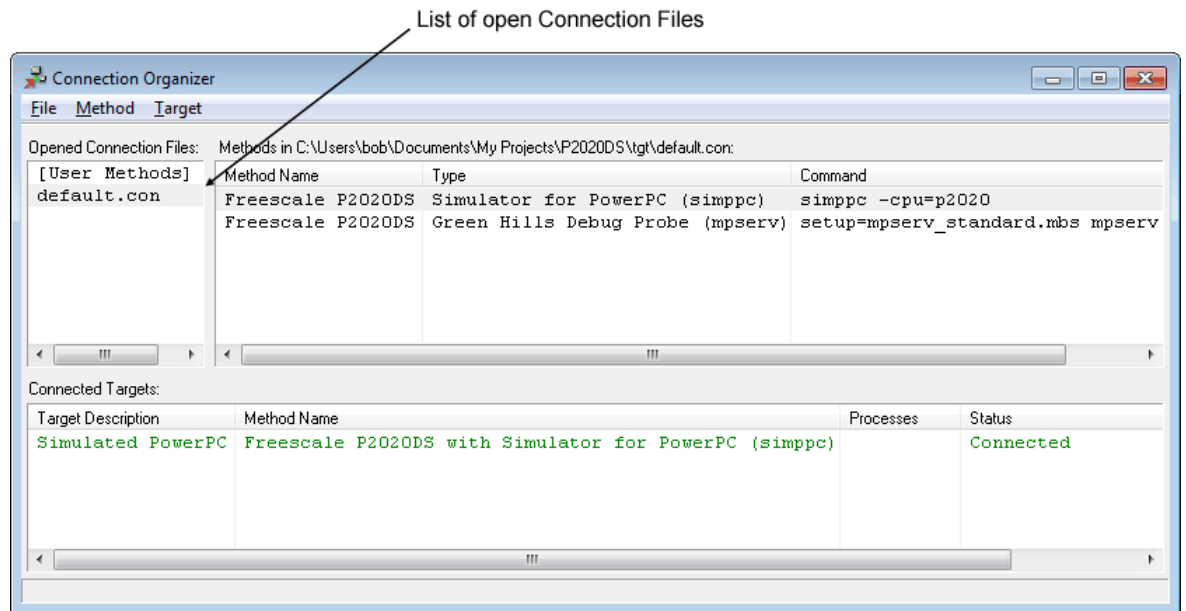
To troubleshoot a connection problem, see the *MULTI: Configuring Connections* book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User's Guide*.

Managing Your Connected Targets

After you have connected using a Connection Method, the connected target appears in the **Connected Targets** list. The **Connected Targets** list is located at the bottom of the **Connection Organizer**. From the **Connected Targets** list, you can find information about the target and control the target.

Connection Organizer Menu and Action Reference

The primary interface for working with Connection Files is the list of **Opened Connection Files** located in the **Connection Organizer**.

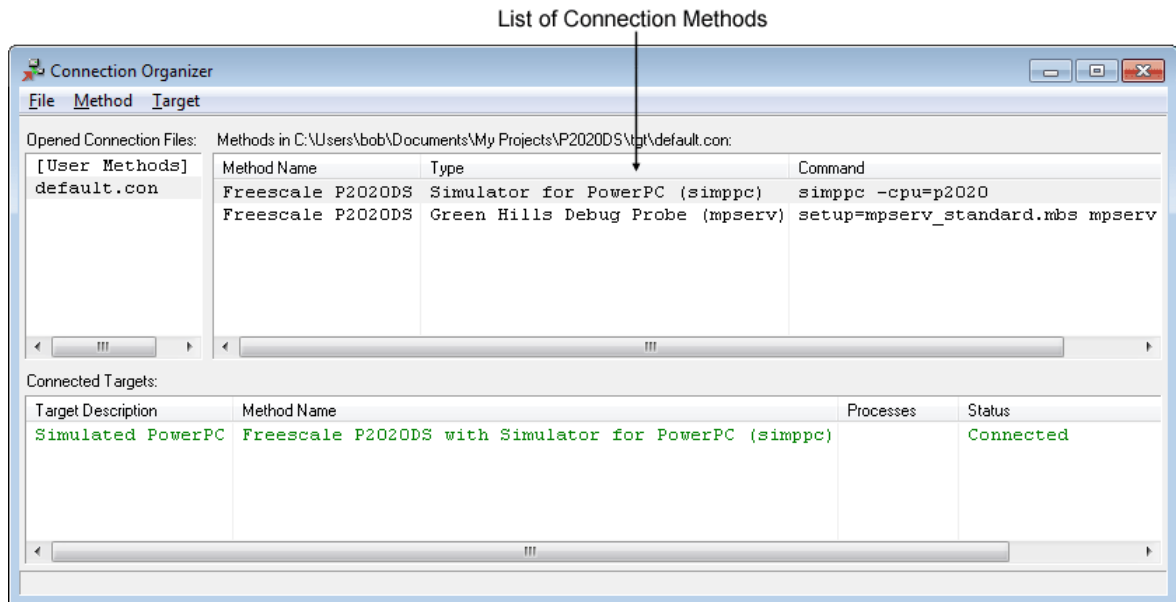


You can select Connection Files from this list and perform the following actions by using the **File** menu. Most of the **File** menu options are also available via the right-click menu.

Menu Item	Meaning
New	Opens a dialog box that prompts you to name a new Connection File. When you click the Create button, the Connection File is created and added to the Opened Connection Files list.
Open	Opens a dialog box that prompts you to open an existing .con file. After you open it, the file appears in the Opened Connection Files list, and you can work with the Connection Methods contained in it.
Close	Closes the selected Connection File. Note that the [User Methods] Connection File is always open and cannot be closed.
Save	Saves any changes to the selected Connection File.
Save a copy	Opens a dialog box that prompts you to choose a different filename for the selected Connection File before saving it.
Save all	Saves all changes to all open Connection Files.

Menu Item	Meaning
Close Window	Closes the window. This menu item does not affect the Connection File.

The primary interface for working with Connection Methods is the list of Connection Methods located under **Methods in selected file** in the **Connection Organizer**.

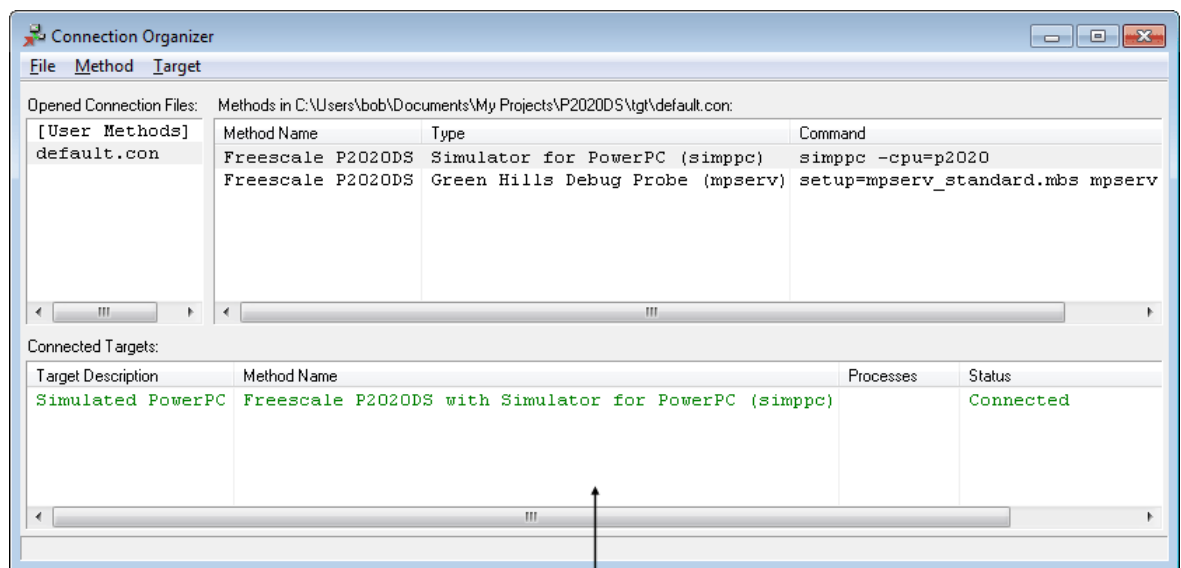


You can select Connection Methods from this list and perform the following actions by using the **Method** menu or the right-click menu.

Menu Item	Meaning
New	Opens a dialog box that prompts you to define a new Connection Method with a name, a type, and options that you specify.
Connect to Target	Uses the settings of the selected Connection Method to connect to the target. If the connection is successful, the connected target appears in the Connected Targets list. If the connection is unsuccessful, an error message and diagnostic information appear.
Connect and Flash	Opens the MULTI Fast Flash Programmer after connecting to the target selected under Methods in selected file . This window allows you to enter parameters for downloading a file to flash memory on the target. See Chapter 22, “Programming Flash Memory” on page 539.
Edit	Opens a Connection Editor that allows you to change the settings of the selected Connection Method. See the documentation about the Connection Editor in the <i>MULTI: Configuring Connections</i> book.

Menu Item	Meaning
Copy	Opens a dialog box that allows you to copy the selected Connection Method into either the same or a different Connection File. If a Connection Method with the same name already exists in the Connection File you copy to, the copy you create is named <i>method_name (2)</i> or <i>method_name (3)</i> , etc.
Move	Opens a dialog box that allows you to move the selected Connection Method into another Connection File. If a Connection Method with the same name already exists in the Connection File you move the selected Connection Method into, the Method you move is renamed <i>method_name (2)</i> or <i>method_name (3)</i> , etc.
Delete	Deletes the Connection Method selected in Methods in selected file from the current Connection File.

The primary interface for working with connected targets is the list of **Connected Targets** located in the **Connection Organizer**.



Connected targets list

You can select connected targets from this list and perform the following actions by using the **Target** menu. Most of the following **Target** menu options are also available via the right-click menu.

Menu Item	Meaning
Flash	Opens the MULTI Fast Flash Programmer . This window allows you to use the selected target connection to write a file located on the host to flash memory on the target. See Chapter 22, “Programming Flash Memory” on page 539.
Show Task Manager	Displays the Task Manager associated with the selected target. This option is only available if the target supports multiple tasks. For more information about the Task Manager, see “The Task Manager” on page 580.
Set Logging	Opens the Target Logging Settings dialog box, which allows you to capture communications between MULTI and the target's debug agent.
Disconnect	Disconnects the selected target from MULTI. Processes that are running or being debugged on the target may be halted or may continue running undisturbed. The exact behavior of Disconnect is dependent on your target and connection. For example, when you disconnect from a simulator, the simulator exits, ending all processes it is running. However, an RTOS debug agent detaches and allows the underlying RTOS to continue running.

Disconnecting from Your Target

To disconnect from your target, do one of the following:

- In the Debugger command pane, enter the **disconnect** command. For information about the **disconnect** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the target list, select a connected executable. Click the **Disconnect** button () or select **Target** → **Disconnect from Target**.
- In the target list, right-click a connected executable and then select **Disconnect from Target** from the shortcut menu that appears.
- In the **Connection Organizer**, select a connection from the **Connected Targets** list and then select **Target** → **Disconnect**.
- In the **Connection Organizer**, right-click a connection in the **Connected Targets** list and then select **Disconnect** from the shortcut menu that appears.

Chapter 4

INDRT2 (rtserv2) Connections

Contents

Introduction to rtserv2 and INDRT2	60
Building in Run-Mode Debugging Support	61
Connecting Overview	62
INDRT2 Connection Methods	62
Automatically Establishing Run-Mode Connections	69
Connecting to Multiple ISIM Targets	74

INDRT2 is a software interface provided by Green Hills that facilitates run-mode debugging of the INTEGRITY real-time operating system (version 10 or later). INDRT2 and **rtserv2**, the debug server that supports INDRT2 connections, are installed automatically when you install a distribution of the MULTI IDE that supports INDRT2 connections.

This chapter supplements the general target connection information in Chapter 3, “Connecting to Your Target” on page 39 with specific information for INDRT2 connections.

For more information about run-mode connections, including a caveat to simultaneously debugging multiple targets in run mode, see “Establishing Run-Mode Connections” on page 578.

Introduction to rtserv2 and INDRT2

Debug servers such as **rtserv2** enable communication between the host and the target. During a debugging session, the MULTI Debugger sends **rtserv2**, a host-resident debug server, requests to read registers, write to memory, etc. In turn, **rtserv2** sends the debugging requests to the target. You can use the **rtserv2** debug server to connect to a running INTEGRITY system (version 10 or later) to perform run-mode debugging.

rtserv2 enables advanced features such as dynamic downloading of virtual AddressSpaces, and use of tools such as the **Profile** window and the EventAnalyzer. When you are connected to **rtserv2**, the target list displays all the tasks in the system. Any user task in the target list can be selected and individually debugged.

Communication Media

An INDRT2 (**rtserv2**) connection can be established over an IP network (typically Ethernet) or a serial link. A BSP typically provides drivers to support an Ethernet or serial device. For information about configuring INTEGRITY's network settings, see the *INTEGRITY Networking Guide*.

The following are general recommendations for INDRT2 (**rtserv2**) connections:

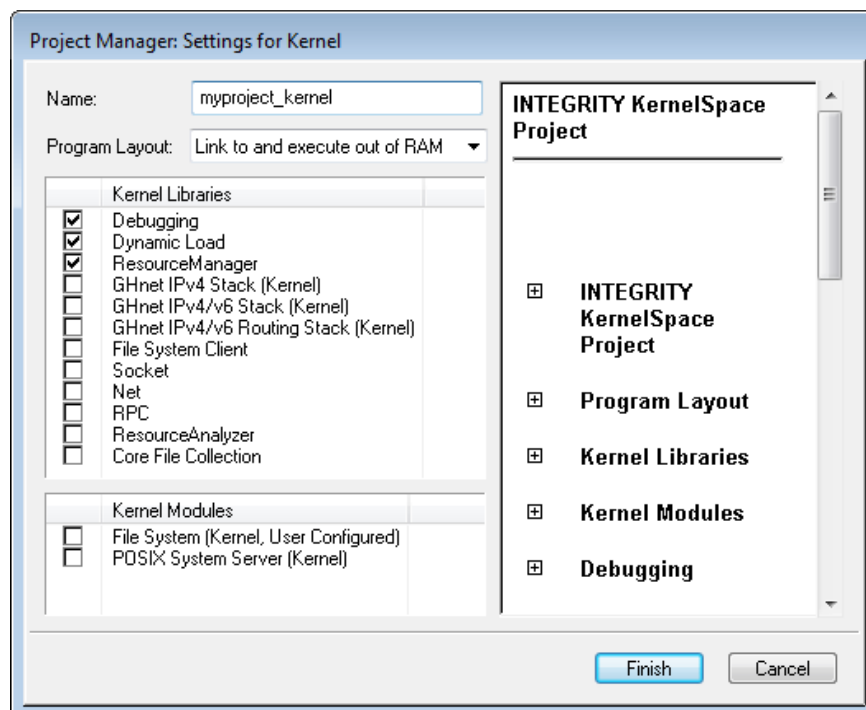
- Use a fast and reliable IP network (such as Ethernet) whenever possible for better performance.

- For BSPs that support only one serial port, the port cannot be used for debugging when it is being used for diagnostics. If the serial port is already in use, close the terminal program used to view the serial port output before establishing your run-mode connection.
- The baud rate at which a particular BSP communicates over a serial port varies. For details, see the documentation that came with your BSP.

Building in Run-Mode Debugging Support

To enable run-mode debugging support, an INTEGRITY kernel program must be linked with a debug library supplied by Green Hills. To include this library in an existing kernel or monolith project:

1. In the Project Manager, locate the **.gpj** file for the kernel within the project (by default, **myproject_kernel.gpj**).
2. Right-click the file, and select **Configure**.
3. In the **Settings for Kernel** window that appears, select the **Debugging** check box.



Connecting Overview

You can establish a debug server connection in any of the following ways. Each is discussed in more detail later in this chapter.

- Graphically configure a Connection Method. For a set of steps to follow, see “Connecting to rtserv2 via the Connection Organizer and Connection Editor” on page 65. For reference information that supplements the general information in Chapter 3, “Connecting to Your Target” on page 39, see the next section.
- Use a custom connection command. See “Using Custom INDRT2 (rtserv2) Connection Methods” on page 66.
- Set up a run-mode partner. See “Automatically Establishing Run-Mode Connections” on page 69.

INDRT2 Connection Methods

To help you connect to your target quickly and easily, MULTI allows you to create and save Connection Methods that correspond to your particular host and target systems and your desired debugging options.

For general instructions that explain how to create and use Connection Methods, see Chapter 3, “Connecting to Your Target” on page 39. The information in the following sections supplements the instructions provided there with information that is specific to INDRT2 connections.

Using the INDRT2 (rtserv2) Connection Editor

In addition to the generic fields that appear on all **Connection Editors** for Standard Connection Methods (see the documentation about the Connection Editor in the *MULTI: Configuring Connections* book), the **INDRT2 (rtserv2) Connection Editor** includes **Connection** and **Debug** tabs that provide settings and options specific to your target and host operating systems.

When the **Connection Editor** is first displayed after you create a new Connection Method, the settings and options are set to default values. Settings and options that are not available on your host operating system may appear dimmed. Some of the fields may require user input before the Connection Method can be used.

Each field is described in the following sections.

INDRT2 (rtserv2) Connection Settings

Ethernet/IP Connection

Sets your desired connection type as Ethernet/IP. This radio button is mutually exclusive with the **Serial Connection** button. If you select an Ethernet/IP connection (this is the default), the following field will also be available:

- **Target Name or Address** — Specifies the host name or IP address of your target. You must specify a host name or IP address to create a valid Ethernet/IP connection.

Note: If you need to connect to your target using a non-standard UDP port number (that is, not 2220), you must use a Custom Connection Method. See “Custom INDRT2 (rtserv2) Connections Over an IP Network” on page 67 for instructions.

Serial Connection

Specifies that a serial connection to the target should be used. This radio button is mutually exclusive with the **Ethernet/IP Connection** button and is not supported on Solaris. If you select a serial connection, the following option fields will also be available:

- **Serial Port** — Specifies which host serial port to use for your serial INDRT2 connection. In the event that you do not specify a port, the default serial port for each supported host operating system is listed below:
 - Windows — **COM1**
 - Linux — **ttyS0**
- **Baud Rate** — Specifies the serial port communication speed. The default baud rate is 9600.

INDRT2 (rtserv2) Debug Settings

The screenshot shows a dialog box with two tabs: 'Connection' and 'Debug'. The 'Debug' tab is active. It features a checkbox labeled 'Log Host-Target Debug Output'. Below this checkbox are two radio buttons: 'StdErr / StdOut' (which is selected) and 'File'. Under the 'File' radio button, there is a text input field containing the text 'rtserv2.log' and a button labeled 'Choose...'. At the bottom of the dialog, there is a text input field labeled 'Other Options:'.



Warning

Do not change the settings on the **Debug** tab unless you are instructed to do so by Green Hills Technical Support.

Log Host-Target Debug Output

Enables logging of all communications between **rtserv2** and your target. Logging is disabled by default.

StdErr/StdOut

File

Allows you to specify the destination for host-target debug output. These fields will be dimmed unless **Log Host-Target Debug Output** is selected.

If you choose **StdErr/StdOut**, host-target debug output will be directed to the console. This setting is not supported on Windows hosts. To log debug output on Windows, select **File**.

If you select **File**, host-target debug output will be directed to the file you specify in the text field. You may enter a filename directly into this text field, or click **Choose** to browse the file system.

Other Options

Allows you to add other optional arguments directly to the command used for connecting. You should only use this field if directed to do so by Green Hills Technical Support.

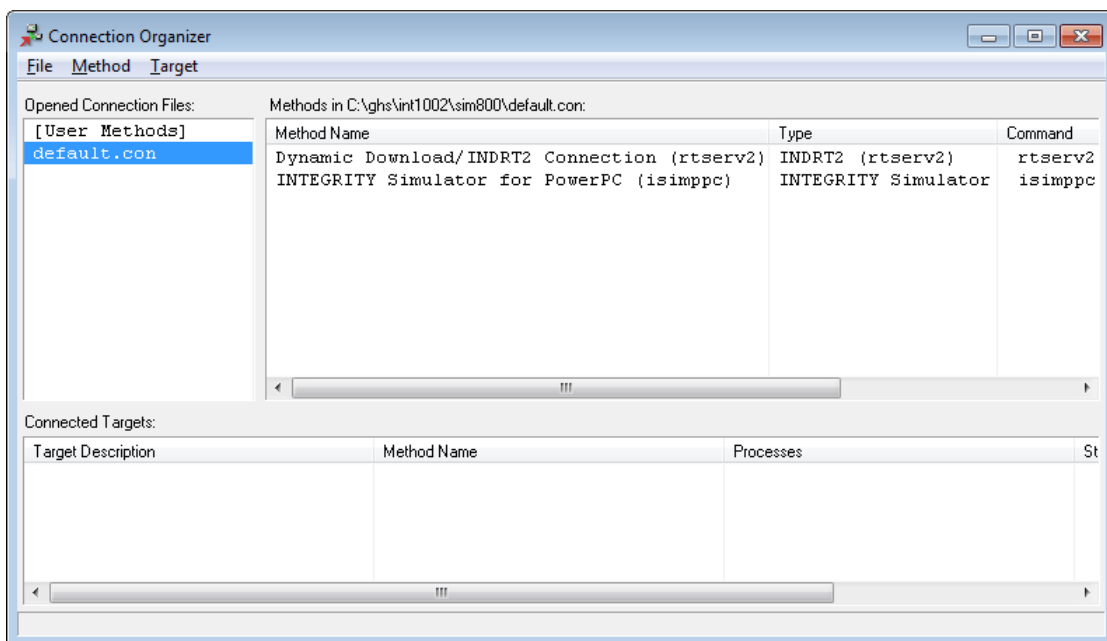
Connecting to rtserv2 via the Connection Organizer and Connection Editor

This section provides a practical set of steps that you can follow to connect to your INTEGRITY target. This set of steps, which uses the **Connection Organizer** and **Connection Editor**, outlines one of the many possible ways in which you can connect. For information about other ways to connect to your target, see Chapter 3, “Connecting to Your Target” on page 39.

1. In the MULTI Project Manager, open the Top Project that came with your BSP, or, if you already have an existing project for your BSP, open its Top Project.

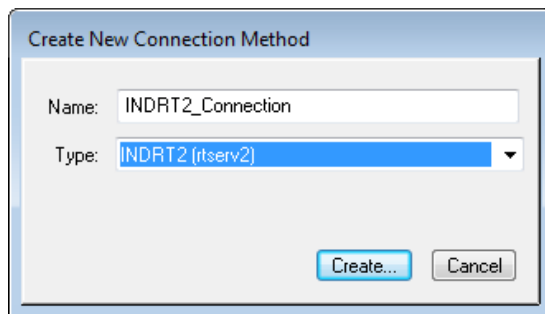
Each BSP contains example connections that you can customize for your own use. These same connections are used by the **Project Wizard** for every project created for that BSP.

2. In the MULTI Project Manager, select **Connect** → **Connection Organizer**.
3. In the **Opened Connection Files** list, select **default.con**.



4. Modify or create a connection:

- To modify an existing connection (recommended) — Double-click the **Dynamic Download/INDRT2 Connection** to open the **Connection Editor**.
- To create a new connection — Select **Method** → **New**. In the **Create New Connection Method** dialog box, specify a **Name**, and select **INDRT2 (rtserv2)** as the **Type**. Click **Create** to open the **Connection Editor**.



5. The **INDRT2 (rtserv2) Connection Editor** includes **Connection** and **Debug** tabs to set options specific to your target and host operating systems. When you first open the **Connection Editor**, the options are set to default values. Some fields may require user input. For a description of each field, see “INDRT2 (rtserv2) Connection Settings” on page 63 and “INDRT2 (rtserv2) Debug Settings” on page 64.
6. After customizing the connection, click **OK** to return to the **Connection Organizer**.
7. Ensure that you are running a properly configured INTEGRITY kernel (see “Building in Run-Mode Debugging Support” on page 61).
8. Right-click the **Dynamic Download/INDRT2 Connection**, and select **Connect to Target**.

Using Custom INDRT2 (rtserv2) Connection Methods

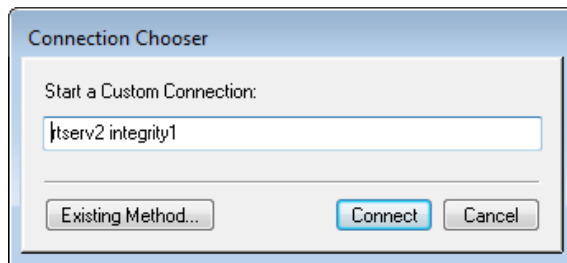
You can create a Custom INDRT2 (rtserv2) Connection Method by manually entering a connection command into the **Connection Chooser** instead of using the graphical **Connection Editor**.

The appropriate commands for Custom INDRT2 Connection Methods are described in the following sections.

Custom INDRT2 (rtserv2) Connections Over an IP Network

To use a Custom Connection Method to establish an INDRT2 connection over an IP network:

1. Open the **Connection Chooser**. One way to do so is to click the **Connect** button () in the MULTI Project Manager.
2. In the **Connection Chooser**, click **Custom**.
3. In the **Start a Custom Connection** field, enter a connection command with appropriate options. The following graphic is only an example. The complete syntax is provided below.



4. Click **Connect**.

The syntax of the command that should be entered into the **Start a Custom Connection** field is:

```
rtserv2 [-log=filename] hostname[:portnumber]
```

where:

- `-log=filename` enables logging of communications between **rtserv2** and your target.
- `hostname` and `:portnumber` should not be separated by spaces.



Note

You can also issue the above connection command from the Debugger's command pane, where it must be preceded by the **connect** command.

For more information, see “Custom Connection Methods” on page 47.

Example 4.1. Establishing a Connection Over an IP Network

To establish an IP connection to an INTEGRITY board named `integrity1`, you could enter the following text in the **Start a Custom Connection** field:

```
rtserv2 integrity1
```

Example 4.2. Logging Communications

To enable logging of communications between **rtserv2** and your target, use the **-log** option in your custom connection command. For example, to enable logging of the example connection above, you would enter the following text in the **Start a Custom Connection** field:

```
rtserv2 -log=myfile integrity1
```

Custom INDRT2 (rtserv2) Connections Over a Serial Link



Note

Serial connections using **rtserv2** are not supported on Solaris.

To use a Custom Connection Method to establish an INDRT2 connection over a serial link, follow the steps listed at the beginning of “Custom INDRT2 (rtserv2) Connections Over an IP Network” on page 67. However, when entering a connection command, use the syntax provided next.

rtserv2 [-log=*filename*] -serial *device* [*baud_rate*]

where:

- `-log=filename` enables logging of communications between **rtserv2** and your target.
- `-serial` specifies a serial connection.
- `device` specifies the serial port device to use when connecting to the target. `device` can be the name of a device or the path to a device. For example, `ttys0` and `/dev/ttys0` are equivalent.

- `baud_rate` sets the serial port communication speed. The default baud rate is 9600.



Note

You can also issue the above connection command from the Debugger's command pane, where it must be preceded by the **connect** command.

For more information, see “Custom Connection Methods” on page 47.

Example 4.3. Establishing a Serial Connection

To establish a serial connection on Windows, you might enter the following text in the **Start a Custom Connection** field:

```
rtserv2 -serial com1 9600
```

On Linux, you might enter:

```
rtserv2 -serial /dev/ttyS0 9600
```

Automatically Establishing Run-Mode Connections

The Debugger allows you to define an INDRT2 or INDRT connection, called a *run-mode partner* that it will automatically establish when you download and run an INTEGRITY kernel. The INTEGRITY kernel must be run via a freeze-mode connection to a GHS simulator or GHS hardware debug solution (Green Hills Probe or SuperTrace Probe). After INTEGRITY has booted, the run-mode partner is established in the same Debugger window as the freeze-mode connection.

Run-mode partnering puts certain measures in place to prevent INDRT2/INDRT connections from being closed prematurely. If you halt a freeze-mode connection or step through source code while in freeze mode, the run-mode partner becomes inaccessible: you cannot attach to or otherwise control tasks, nor can you read or write registers or memory. In addition, host I/O calls issued over run mode will not be serviced. (You may be able to continue browsing source code in tasks that you are already attached to.) These measures reduce the chance that the Debugger will erroneously attempt to communicate with **rtserv2** or **rtserv** while the target is halted via the freeze-mode connection.

If you kill, restart, or disconnect from a freeze-mode connection, the Debugger automatically disconnects from any run-mode partner set on the same target.



Note

Run-mode partnering is only supported if you are debugging an INTEGRITY target and if you are connected to a GHS simulator or to one of the GHS hardware debug solutions (Green Hills Probe or SuperTrace Probe) via a freeze-mode connection. The run-mode partner requires an Ethernet connection to the target, and is not triggered by INTEGRITY until after the target has an IP address. (If your target is using DHCP, the connection is not triggered if the target cannot communicate with the DHCP server.)

The following limitations apply to run-mode partnering:

- Automatic establishment of the INDRT2/INDRT connection is aborted if you are stepping through kernel startup code or if any breakpoints are set on the freeze-mode connection when MULTI initializes the INDRT2/INDRT connection.
- If you are running a pre-INTEGRITY-10 kernel, the INDRT connection is only automatically established if the Idle Task gets to run—that is, if your system has at least some idle time and the processor is not being fully scheduled by the kernel.

See also “Troubleshooting” on page 73.

Setting a Run-Mode Partner

The **Set Run-Mode Partner** dialog box appears automatically the first time you download and run your INTEGRITY kernel using a freeze-mode Connection Method. Select an existing INDRT2/INDRT Connection Method in the drop-down list, or create a new INDRT2/INDRT Connection Method by clicking the **Create a new Connection Method** (📎) button. The INDRT2/INDRT connection is automatically established when your INTEGRITY kernel boots.

To modify the run-mode partner setting for your freeze-mode Connection Method, perform one of the following actions:

- Select a freeze-mode connection in the target list, and choose **Target** → **Set Run-Mode Partner**, or enter **set_runmode_partner** in the command pane. (For information about the **set_runmode_partner** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) Select or create a run-mode Connection Method.
- Use the **Connection Editor** to edit the freeze-mode Connection Method. Select the **INTEGRITY** or **Connection** tab, and enter the name of an existing INDRT2/INDRT Connection Method in the text field labeled **Run-Mode Partner Connection**. For information about the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 44.

The INDRT2/INDRT connection you specify is automatically established when your INTEGRITY kernel boots.

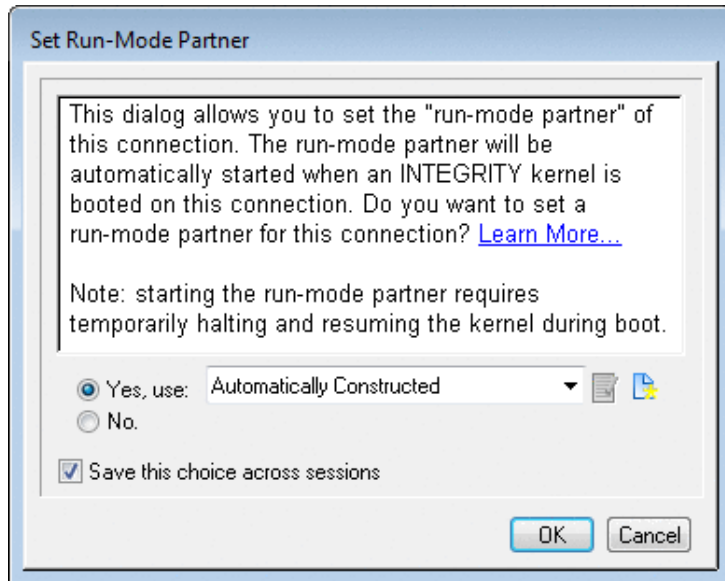


Tip

If you lose your INDRT2/INDRT connection, you can enter the command **connect -restart_runmode** in the Debugger command pane to try to reconnect. For more information, see the **connect** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

The Set Run-Mode Partner Dialog Box

The **Set Run-Mode Partner** dialog box allows you to configure the automatically established INDRT2/INDRT connection for your current freeze-mode connection. For information about accessing the **Set Run-Mode Partner** dialog box, see “Setting a Run-Mode Partner” on page 70.



The **Set Run-Mode Partner** dialog box contains the following options:

- **Yes, use** — Specifies the INDRT2/INDRT Connection Method that the Debugger automatically attempts to establish when you boot an INTEGRITY kernel via the current freeze-mode connection. The drop-down list contains known INDRT2/INDRT Connection Methods and, for versions 10 and later of INTEGRITY, an additional entry labeled **Automatically Constructed**. When you select **Automatically Constructed**, the operating system itself attempts to tell the Debugger what address and Method to use to create an INDRT2 connection to the target.
- **No** — Disables the automatic establishment of an INDRT2/INDRT connection when you boot an INTEGRITY kernel via the current freeze-mode connection.
- **Save this choice across sessions** — If selected, the settings in the dialog box are associated with the current freeze-mode debug connection and used in future debug sessions. If cleared, the dialog box settings are used only until you exit MULTI, after which the freeze-mode connection reverts to whatever run-mode partner it had last (if any).

Disabling Automatically Established Run-Mode Connections

To disable run-mode partnering for a particular freeze-mode connection, perform the following steps:

1. In the target list, select the freeze-mode connection.
2. Select **Target** → **Set Run-Mode Partner**.
3. In the **Set Run-Mode Partner** dialog box, select **No**.
4. Optionally select **Save this choice across sessions**.
5. Click **OK**.

Alternatively, you can select the freeze-mode connection in the target list and enter **set_runmode_partner -none** in the Debugger command pane. For information about the **set_runmode_partner** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Troubleshooting

You may encounter the following problems while debugging via a simultaneous freeze-mode connection and an INDRT2/INDRT (run-mode) connection to the same target, regardless of whether or not the INDRT2/INDRT connection was created automatically via the run-mode partner functionality, or whether or not the freeze-mode and INDRT2/INDRT connections were created within the same Debugger process.

- Host I/O calls made on a freeze-mode connection can cause an INDRT2/INDRT connection to the same target to become unresponsive for the duration of the host I/O call. Note that some host I/O calls (for example, large block reads from, or writes to, host files) can take a long time for the Debugger to process. Also note that some host I/O calls, such as reads from standard input, may block indefinitely, waiting for you to type input in the I/O pane. To avoid this problem, do not write code that makes freeze-mode host I/O calls.
- Host I/O calls made on an INDRT2/INDRT connection can cause your program to hang. To remedy this, disconnect from the INDRT2/INDRT connection.
- Breakpoints that are set on a freeze-mode connection can cause the INDRT2/INDRT connection to the same target to become unresponsive until

the freeze-mode connection is resumed. This applies regardless of whether hitting the breakpoint leaves the target halted, as in the case of a regular software breakpoint, or whether it resumes the target, as in the case of a conditional breakpoint whose condition is false. To avoid this problem, remove all freeze-mode breakpoints before initiating the INDRT2/INDRT connection.

- A single instance of the MULTI Debugger can deadlock between a freeze-mode connection and an INDRT2/INDRT connection to the same target, even though various safeguards have been introduced to prevent this problem in common cases. If you encounter this problem, after a short delay (configurable via MULTI's `SERVERTIMEOUT` system variable), you may see a dialog box saying **Server message timed out. Terminate Connection?** with buttons labeled **Continue** and **Terminate**. To regain control of the Debugger and freeze-mode connection, click **Terminate**. You may also be able to re-initiate the INDRT2/INDRT connection request after this. To avoid this problem, launch a separate instance of the MULTI Debugger to connect to the target via the INDRT2/INDRT Connection Method.

Using the run-mode partner functionality is preferable to manually establishing an INDRT2/INDRT connection. However, if you must manually establish an INDRT2/INDRT connection, do so several seconds after booting your kernel via the freeze-mode connection. For example, wait for the INTEGRITY kernel banner to appear on your target's serial port before connecting via a run-mode debug server such as **rtserv** or **rtserv2**.

Connecting to Multiple ISIM Targets

By default, the INTEGRITY simulator (ISIM) accepts INDRT2 connections on the default UDP socket port of 2220. Only one ISIM instance on a given host can listen on a particular port. To connect to multiple ISIM instances on the same host, some of the ISIM instances must listen on different ports.

For example, to create an ISIM connection that listens on port 3330 instead of port 2220, first configure a connection to the simulator:

1. In the **Connection Chooser**, click the **Create a New Connection Method**  button.
2. For the connection type, select **INTEGRITY Simulator (isim)** and click **Create**. The **Connection Editor** will open.

3. Select the **Debug** tab. In the **Other Options** field enter:

```
-host_indrt_port 3330
```

Then configure an INDRT2 (**rtserv2**) connection to work with the simulator connection you just set up:

1. Create a custom INDRT2 connection in the following format:

```
rtserv2 [-log=filename] localhost[:portnumber]
```

2. Select your INDRT2 connection in the **Connection Chooser**, and click the **Edit the selected Connection Method** button (.
3. In the **Connection Editor**, select the **Connection** tab and enter `localhost:3330` in the **Target Name or Address** field.

When using socket emulation for TCP/IP communications on an ISIM target, those emulated socket ports may also conflict with other ISIM instances or with the host's reserved ports or running services. These ports can also be remapped. For more information, see the documentation about ISIM socket port remapping in the *INTEGRITY Development Guide*.

Chapter 5

INDRT (rtserv) Connections

Contents

Introduction to rtserv and INDRT	78
Building in Run-Mode Debugging Support	79
Connecting Overview	80
INDRT Connection Methods	80
Connecting with rtserv over the ARM Debug Comm Channel	88

INDRT is a software interface provided by Green Hills that facilitates run-mode debugging of the INTEGRITY real-time operating system (version 5). INDRT and **rtserv**, the debug server that supports INDRT connections, are installed automatically when you install a distribution of the MULTI IDE that supports INDRT connections.

This chapter supplements the general target connection information in Chapter 3, “Connecting to Your Target” on page 39 with specific information for INDRT connections.

For more information about run-mode connections, including a caveat to simultaneously debugging multiple targets in run mode, see “Establishing Run-Mode Connections” on page 578.

Introduction to **rtserv** and INDRT

Debug servers such as **rtserv** enable communication between the host and the target. During a debugging session, the MULTI Debugger sends **rtserv**, a host-resident debug server, requests to read registers, write to memory, etc. In turn, **rtserv** sends the debugging requests to the target. You can use the **rtserv** debug server to connect to a running INTEGRITY system (version 5) to perform run-mode debugging.

rtserv enables advanced features such as dynamic downloading of virtual AddressSpaces, and use of tools such as the **Profile** window. When you are connected to **rtserv**, the target list displays all the tasks in the system. Any user task in the target list can be selected and individually debugged.

Communication Media

An INDRT (**rtserv**) connection can be established over an IP network (typically Ethernet) or a serial link. A BSP typically provides drivers to support an Ethernet or serial device.

The following are general recommendations for INDRT (**rtserv**) connections:

- Use a fast and reliable IP network (such as Ethernet) whenever possible for better performance.
- For BSPs that support only one serial port, the port cannot be used for debugging when it is being used for diagnostics. If the serial port is already in use, close

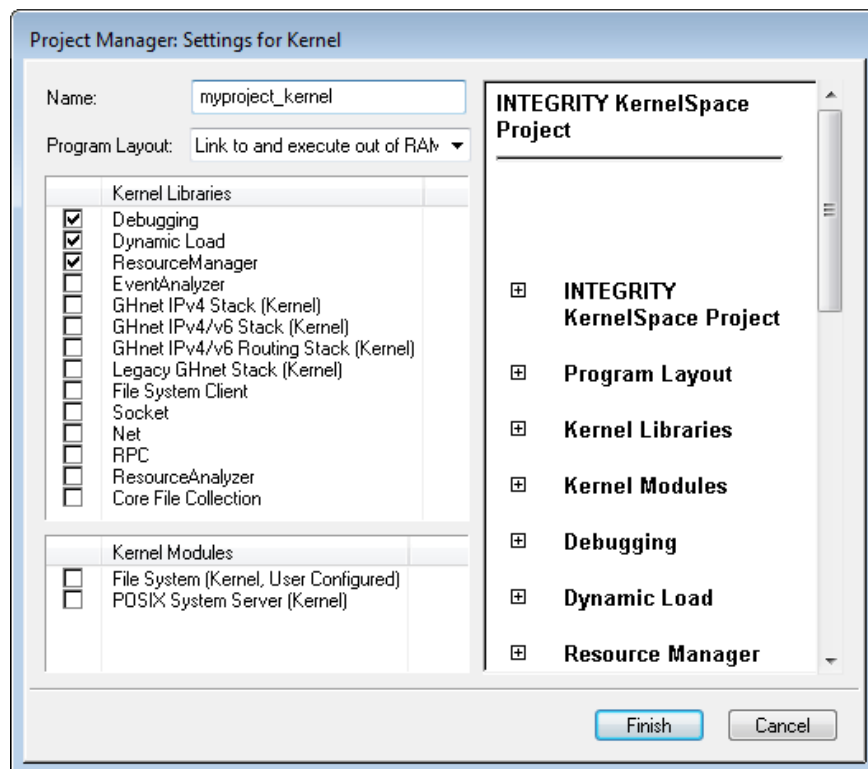
the terminal program used to view the serial port output before establishing your run-mode connection.

- The baud rate at which a particular BSP communicates over a serial port varies. For details, see the documentation that came with your BSP.

Building in Run-Mode Debugging Support

To enable run-mode debugging support, an INTEGRITY kernel program must be linked with a debug library supplied by Green Hills. To include this library in an existing kernel or monolith project:

1. In the Project Manager, locate the **.gpj** file for the kernel within the project (by default, **myproject_kernel.gpj**).
2. Right-click the file, and select **Configure**.
3. In the **Settings for Kernel** window that appears, select the **Debugging** check box.



Connecting Overview

You can establish a debug server connection in any of the following ways. Each is discussed in more detail in the section referenced.

- Graphically configure a Connection Method. For a set of steps to follow, see “Connecting to rtserv via the Connection Organizer and Connection Editor” on page 84. For reference information that supplements the general information in Chapter 3, “Connecting to Your Target” on page 39, see the next section.
- Use a custom connection command. See “Using Custom INDRT (rtserv) Connection Methods” on page 86.
- Set up a run-mode partner. See “Automatically Establishing Run-Mode Connections” on page 69.

INDRT Connection Methods

To help you connect to your target quickly and easily, MULTI allows you to create and save Connection Methods that correspond to your particular host and target systems and your desired debugging options.

For general instructions that explain how to create and use Connection Methods, see Chapter 3, “Connecting to Your Target” on page 39. The information in the following sections supplements the instructions provided there with information that is specific to INDRT connections.

Using the INDRT (rtserv) Connection Editor

In addition to the generic fields that appear on all **Connection Editors** for Standard Connection Methods (see the documentation about the Connection Editor in the *MULTI: Configuring Connections* book), the **INDRT (rtserv) Connection Editor** includes **Connection**, **Advanced**, and **Debug** tabs that provide settings and options specific to your target and host operating systems.

When the **Connection Editor** is first displayed after you create a new Connection Method, the settings and options are set to default values. Settings and options that are not available on your host operating system may appear dimmed. Some of the fields may require user input before the Connection Method can be used.

Each field is described in the following sections.

INDRT (rtserv) Connection Settings

The screenshot shows the 'INDRT (rtserv) Connection Editor' dialog box. It has three tabs: 'Connection', 'Advanced', and 'Debug'. The 'Connection' tab is active. Inside, there are two radio buttons. The first, 'Ethernet / IP Connection', is selected. Below it is a text field for 'Target Name or Address:' and a 'TFTP Load Directory:' text field with a 'Choose...' button. The second radio button, 'Serial Connection', is unselected. Below it are two dropdown menus for 'Serial Port' and 'Baud Rate', both showing '<default>'.

Ethernet/IP Connection

Sets your desired connection type as Ethernet/IP. This radio button is mutually exclusive with the **Serial Connection** button. If you select an Ethernet/IP connection (this is the default), the following fields will also be available:

- **Target Name or Address** — Specifies the host name or IP address of your target. You must specify a host name or IP address to create a valid Ethernet/IP connection.
- **TFTP Load Directory** — Specifies a load directory for dynamic downloading via TFTP. This is the directory to which **rtserv** will copy a file before requesting a download. Enter the name of your desired TFTP load directory or click **Choose** to browse to it. The specified directory must be accessible by the TFTP server.

Note: If you need to connect to your target using a non-standard UDP port number (that is, not 2220), you must use a Custom Connection Method. See “Using Custom INDRT (rtserv) Connection Methods” on page 86 for instructions.

Serial Connection

Specifies that a serial connection to the target should be used. This radio button is mutually exclusive with the **Ethernet/IP Connection** button. If you select a serial connection, the following option fields will also be available:

- **Serial Port** — Specifies which host serial port to use for your serial INDRT connection. In the event that you do not specify a port, the default serial port for each supported host operating system is listed below:
 - Windows — **COM1**
 - Linux — **/dev/ttyS0**
 - Solaris — **/dev/ttya**
- **Baud Rate** — Specifies the serial port communication speed. The default baud rate is 9600.

INDRT (rtserv) Advanced Settings

Connection Advanced Debug

☒ BSD Exclusive Serial Port Access

☐ Always Use TFTP for Load

**Warning**

Use this tab carefully, since changing the advanced options from their default settings can cause problems with your connection.

BSD Exclusive Serial Port Access

Solaris only

Enables exclusive serial port access.

By default, the serial port is opened in exclusive, or locked, mode. If you are connecting through a terminal server, clear this box to disable exclusive serial port access.

Always Use TFTP for Load

Forces **rtserv** to use TFTP for downloads, even when the system defaults to a different download method (for example, host I/O).

INDRT (rtserv) Debug Settings

Connection Advanced **Debug**

☐ Log Host-Target Debug Output

☒ StdErr / StdOut

☐ File

rtserv.log Choose...

Other Options:



Warning

Do not change the settings on the **Debug** tab unless you are instructed to do so by Green Hills Technical Support.

Log Host-Target Debug Output

Enables logging of all communications between **rtserv** and your target. Logging is disabled by default.

StdErr/StdOut

File

Allows you to specify the destination for host-target debug output. These fields will be dimmed unless **Log Host-Target Debug Output** is selected.

If you choose **StdErr/StdOut**, host-target debug output will be directed to the console. This setting is not supported on Windows hosts. To log debug output on Windows, select **File**.

If you select **File**, host-target debug output will be directed to the file you specify in the text field. You may enter a filename directly into this text field, or click **Choose** to browse the file system.

Other Options

Allows you to add other optional arguments directly to the command used for connecting. You should only use this field if directed to do so by Green Hills Technical Support.

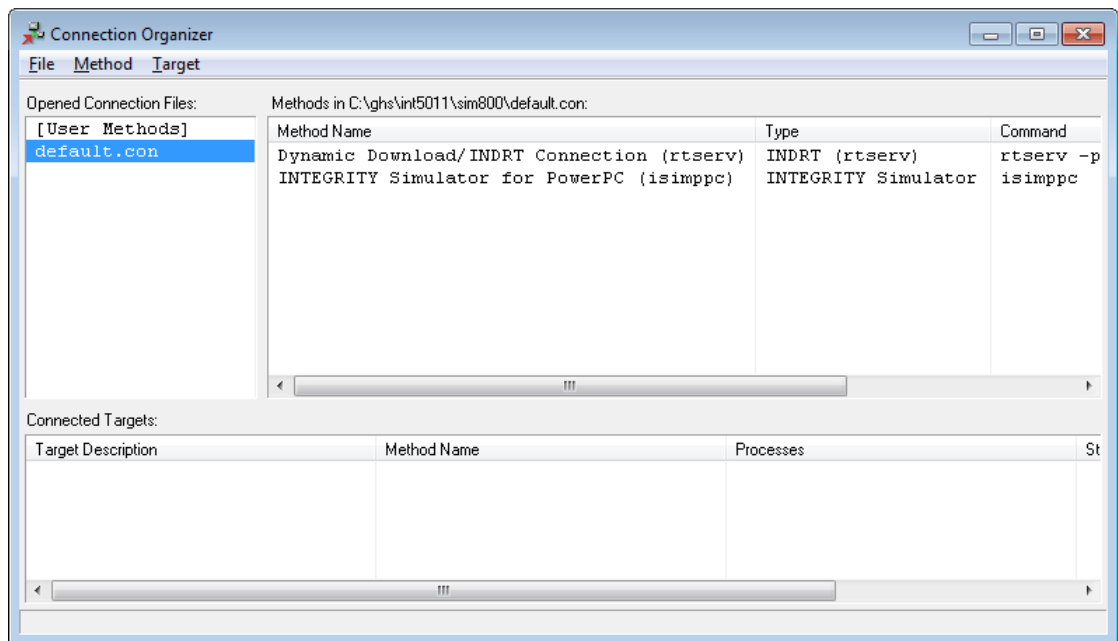
Connecting to rtserv via the Connection Organizer and Connection Editor

This section provides a practical set of steps that you can follow to connect to your INTEGRITY target. This set of steps, which uses the **Connection Organizer** and **Connection Editor**, outlines one of the many possible ways in which you can connect. For information about other ways to connect to your target, see Chapter 3, “Connecting to Your Target” on page 39.

1. In the MULTI Project Manager, open the Top Project that came with your BSP, or, if you already have an existing project for your BSP, open its Top Project.

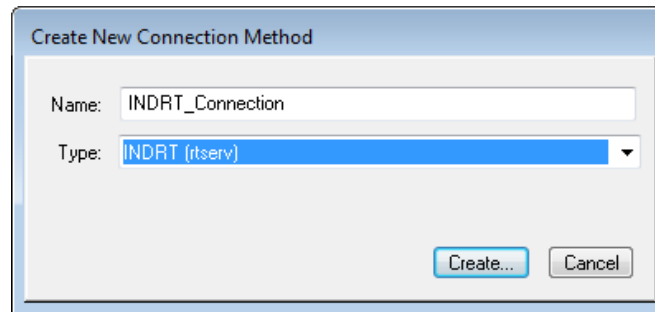
Each BSP contains example connections that you can customize for your own use. These same connections are used by the **Project Wizard** for every project created for that BSP.

2. In the MULTI Project Manager, select **Connect** → **Connection Organizer**.
3. In the **Opened Connection Files** list, select **default.con**.



4. Modify or create a connection:
 - To modify an existing connection (recommended) — Double-click the **Dynamic Download/INDRT Connection** to open the **Connection Editor**.

- To create a new connection — Select **Method** → **New**. In the **Create New Connection Method** dialog box, specify a **Name**, and select **INDRT (rtserv)** as the **Type**. Click **Create** to open the **Connection Editor**.



5. The **INDRT (rtserv) Connection Editor** includes **Connection**, **Advanced**, and **Debug** tabs to set options specific to your target and host operating systems. When you first open the **Connection Editor**, the options are set to default values. Some fields may require user input. For a description of each field, see “INDRT (rtserv) Connection Settings” on page 81, “INDRT (rtserv) Advanced Settings” on page 82, and “INDRT (rtserv) Debug Settings” on page 83.
6. After customizing the connection, click **OK** to return to the **Connection Organizer**.
7. Ensure that you are running a properly configured INTEGRITY kernel (see “Building in Run-Mode Debugging Support” on page 79).
8. Right-click the **Dynamic Download/INDRT Connection**, and select **Connect to Target**.

Using TFTP

In releases of INTEGRITY prior to version 5, TFTP was used for fast dynamic downloads. In INTEGRITY 5, fast downloads are accomplished via the built-in INDRT protocol, and TFTP is no longer necessary. However, there may be situations where you want to force the use of TFTP. To do so:

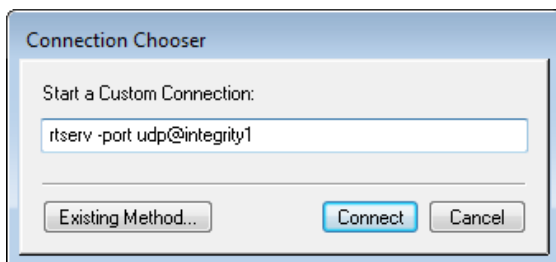
1. On the **Connection** tab of the **Connection Editor**, set the **TFTP Load Directory**.
2. On the **Advanced** tab, select **Always Use TFTP for Load**.
3. Click **OK** or **Apply**.

Using Custom INDRT (rtserv) Connection Methods

You can create a Custom INDRT (**rtserv**) Connection Method by manually entering a connection command into the **Connection Chooser** instead of using the graphical **Connection Editor**.

To establish an INDRT connection using a Custom Connection Method:

1. Open the **Connection Chooser**. One way to do so is to click the **Connect** button () in the MULTI Project Manager.
2. In the **Connection Chooser**, click **Custom**.
3. In the **Start a Custom Connection** field, enter a connection command with appropriate options.



- To establish an IP connection to a board running INTEGRITY with a BSP that supports Ethernet, enter a command similar to the following.

```
rtserv -port udp@integrity1
```

In this example, *integrity1* is the host name that the system administrator has set up for the board.

- To establish a serial debugging connection, enter the following command (port selection may vary):

```
rtserv com1 9600 (Windows)
```

```
rtserv /dev/ttya 9600 (Solaris)
```

- To establish a debugging connection over the ARM debug comm channel using the Green Hills Probe or Slingshot as a communications channel, simply connect as if over an IP network to the IP address of the host that is running **mpserv**, after configuring **mpserv** and the target system

appropriately. For more details, see “Connecting with rtserv over the ARM Debug Comm Channel” on page 88.

Always Connect Mode

rtserv accepts the **-alwaysconnect** option, which enables always connect mode. A Custom Connection Method must be used to specify this option (it is not accessible from the **Connection Organizer** GUI).

In always connect mode:

- An **rtserv** session is always established, even if the user-specified target is not up and running yet.
- The target list provides an indication that a target is not up yet with a task entry named **NOT CONNECTED**.
- When the target does come up, **rtserv** automatically connects (the target is polled periodically).
- While one or more targets are in the **NOT CONNECTED** state, debugging of the connected targets is slower because of the background polling of the unconnected targets.

This mode is useful when you want to establish an **rtserv** session without necessarily knowing or caring whether the target is up yet (or if you tend to forget to boot a target first).



Note

This is simply a method for delaying connection. Targets that go down do not revert to the **NOT CONNECTED** status.

Connecting with rtserv over the ARM Debug Comm Channel

To enable forwarding of **rtserv** communications over the ARM debug comm channel using the Green Hills Probe, run the **mpserv** debug server with a special option as follows:

```
-runmode_port 2220
```

This instructs **mpserv** to forward **rtserv** packets between the ARM debug comm channel and UDP port 2220 (the standard INDRT target connection port). To connect with **rtserv** over this port, specify a connection using the host's IP address.

Debugging performance over the ARM debug comm channel is adequate for most debugging tasks, but dynamic downloading of applications can take a long time to complete.

The ARM debug comm channel interface is further described in the *INTEGRITY Development Guide*.

Chapter 6

Configuring Your Target Hardware

Contents

Installing Your Target Hardware	90
Configuring Your Target Hardware for Debugging	90
Specifying Setup Scripts	98

This chapter describes how to set up your target hardware for use with MULTI. This chapter is not relevant if you are connecting to a simulated target.

Installing Your Target Hardware

Before you can configure your embedded target for use with MULTI, you must install your hardware and any necessary software. Installation details depend on your particular system. For detailed instructions, see your hardware documentation and the chapter in the *MULTI: Configuring Connections* book that discusses your debug server.

Configuring Your Target Hardware for Debugging

Before beginning your first debugging session, you should ensure that your target board is configured properly. For many targets, you will need to run a board setup script to initialize your target before downloading and running a program. This script, along with required linker directives files that provide MULTI with information about your target's memory map, are typically generated for you when you create a new Top Project using the **Project Wizard** and Project Manager.

To create an example “Hello World” project for your target, follow the steps provided in Chapter 1, “Creating a Project” in the *MULTI: Managing Projects and Configuring the IDE* book. If you can build and download the example (see “Quick Start: Building and Running Hello World” in Chapter 1, “Creating a Project” in the *MULTI: Managing Projects and Configuring the IDE* book), the default board setup script and linker directives file configured your target board properly and no additional configuration is required. However, if you are using custom hardware, or if you experience problems when downloading the program, you may need to create or customize the setup script and/or customize the linker directives file in use. The following sections provide customization guidelines and diagnostics that you can use to make sure that these target resources are configured correctly.

Customizing MULTI Board Setup Scripts

A board setup script is a file that MULTI uses to initialize your target before downloading and debugging a program. The default board setup script (if any) for your target is copied in by the Project Manager when you create a new project.

Setup scripts in MULTI 6 use the following conventions and commands:

- The MULTI scripting conventions described in Chapter 1, “Using MULTI Scripts” in the *MULTI: Scripting* book.
- MULTI Debugger commands described in the *MULTI: Debugging Command Reference* book. You can find an overview of useful board setup script commands in “Useful Commands for MULTI Board Setup Scripts” on page 94.
- Debug server commands listed in the *MULTI: Configuring Connections* book or, if you are using a Green Hills Debug Probe, in the documentation about probe commands in the *Green Hills Debug Probes User's Guide*. You must prefix debug server commands with the MULTI Debugger **target** command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book).



Note

Green Hills Monitor targets usually do not require a setup script.



Note

The following sections assume that you are using a MULTI board setup script (**.mbs**). If you want to use a legacy setup script (**.dbs**) generated by MULTI 4 or older, see “Specifying Setup Scripts” on page 98 and the documentation about Green Hills debug server scripts and commands in the *MULTI: Configuring Connections* book.

To edit a MULTI board setup script to make it suitable for your system:

1. Obtain your board and processor's documentation. You will need it to gather information about memory resources, registers, interrupts, etc.
2. Double-click the default setup script in your target resources project to open it in an editor. If there are multiple setup scripts, choose the one containing the name of the debug server that supports your specific debugging interface. For example, if you are using the **mpserv** debug server, which supports Green Hills Debug Probe connections, the default setup script file is **mpserv_standard.mbs**.

3. Determine whether your board can initialize itself, and then proceed as follows:
 - If your target does not have a valid ROM image that initializes the target upon reset, skip to the next step.
 - If your target has a valid ROM image that initializes the target upon reset, comment out the contents of the MULTI board setup script you are editing by adding two forward slashes (//) to the beginning of each line. Replace the script with the command sequence shown in the following (or with an equivalent command sequence).

```
// Reset and halt the board
reset
// Let the ROM image run the target
c
// Give the ROM image 3 seconds to set up the board
wait -time 3000
// Halt the board to get ready for debugging
halt
```



Note

You may need to alter the **wait** command, depending on how long your board takes to set itself up (for information about this command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book).

If you need to initialize your board further, continue to the next step. Otherwise, save the setup script and skip to “Customizing Linker Directives Files” on page 98.

4. Verify that your setup script begins with a command that resets the target, such as the MULTI Debugger **reset** command or the debug server **target tr** command. (For information about these commands, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book and the documentation about probe commands in the *Green Hills Debug Probes User's Guide*.)

Ensure that your target is halted before continuing initialization, or the state of the target might be unpredictable.

5. Disable any interrupt sources that can disrupt the setup or destabilize the board's memory. (For example, if you are using a PowerPC 860 processor, disable the watchdog timer to prevent it from interrupting your target board setup.) The following steps provide a general procedure for disabling interrupt sources:
 - a. Determine whether your processor has any interrupt sources that might disturb your debugging session.
 - b. Using your processor's documentation, determine which registers affect interrupt sources. Then determine the values those registers must have to disable the interrupt sources.
 - c. Enter the commands necessary to configure your memory resources into your setup script. See “Useful Commands for MULTI Board Setup Scripts” on page 94.
6. Configure your target's memory controller based on your board's memory resources. If your memory controller and memory resources are already properly configured, skip to the next step. The following are general steps for configuring memory using a setup script:
 - a. Determine what memory resources your board has by answering the following:
 - How fast and how big is the board's memory, and where do you want to map it?
 - Does the board have SRAM? If so, where is it?
 - Does the board have DRAM? If so, where is it, and where is the DRAM controller for it?
 - Does the DRAM controller need refresh timing information or knowledge of any special modes the DRAM chips may have, such as Synchronous DRAM?
 - Does the board require a peripheral memory base register to access memory controllers or other on-chip peripherals?
 - b. If your processor requires you to set up the base register before you can access your memory controllers or other on-chip peripherals, set the base register.
 - c. Using your processor's documentation and memory resources, determine which memory-related registers you must set. Additionally, determine what values those registers must have to properly configure your memory resources.

- d. Enter the commands necessary to configure your memory resources into your setup script (for more information, see “Useful Commands for MULTI Board Setup Scripts” on page 94).
7. Save your setup script. For information about specifying and running the script, see “Specifying Setup Scripts” on page 98.

Testing Individual Commands

If you have connected MULTI to your target, you can use the MULTI Debugger's command pane to confirm the success or failure of each command individually instead of trying to debug an entire setup script.

Some commands cannot be tested individually because they must be executed within a certain time period in relation to other commands. In this case, put the relevant commands into a small script and run the script from the Debugger's command pane using the `<` command, or type them in the same command line, separated by semicolons (`;`). (For information about the `<` command, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.)

Useful Commands for MULTI Board Setup Scripts

You can find complete documentation for all MULTI Debugger commands in the *MULTI: Debugging Command Reference* book. However, only a small subset of these commands are needed for most setup scripts. The following table outlines these commands.

addhook
Adds a hook to a Debugger action.
c
Continues a stopped target.
clearhooks
Removes hooks.
eval
Evaluates an expression without printing the result.

halt
Halts the target.
memread
Performs a sized memory read from the target, and prints the result.
memwrite
Performs a sized memory write to the target.
reset
Resets the target.
target
Transmits commands directly to the debug server. Use this command before a debug server command. For example, to pass the tr Green Hills Debug Probe command to mpserv using the Debugger, type:
<pre>target tr</pre>
To pass the output of a MULTI command to a debug server command, use the following syntax:
<pre>target command %EVAL{multi_command}</pre>
wait
Blocks command processing.
<pre>\$register = value</pre>
Sets a register named <i>register</i> on your target board to <i>value</i> . For example:
<pre>> \$ivor0 = 0x10</pre>
<pre>\$register.field = value</pre>
Sets a field in <i>register</i> to <i>value</i> . For example:
<pre>> \$CPSR.F = 1</pre>
If the name you specify for <i>register</i> is not a named register, MULTI creates a new variable using the name provided. For example, the following command creates a new variable called <code>\$FOO</code> and sets its value to 6:
<pre>> \$FOO = 6</pre>
You can also use C-style expressions for complex memory manipulation. For example:
<pre>> *((unsigned int *) 0x8000) = 0x10</pre>

Example MULTI Board Setup Script

This example script sets up the Motorola MBX for running programs. Even if you are not using this board, this example contains techniques that may be useful when you write a script for your hardware.

```
// delay slow
// Reset the processor
target rst
// set the internal register space to 0xff000000
$immr=0xff000000
// turn off the watchdog timer --- disable always
memwrite 4 0xff000004 0xffffffff88
// disable the cache
$dc_cst=0x04000000
// The board is configured to run at 40 MHz
// init set_clock=1 cpu_speed=40000000
// delay fast

// Setup memory
memwrite 4 0xFF00017C 0xCFAFC004
memwrite 4 0xFF000168 0x00000000

...

memwrite 2 0xFF00017A 0x0200
```

Testing Target Access

After you have created your new setup script, open your project in the MULTI Debugger. At the bottom of the Debugger window is the command pane, which you can use to send commands to MULTI. To run your project's default setup script:

- Type **setup** in the command pane.

After running this command, test that your target is correctly initialized by performing the diagnostics documented in the following sections.

Testing Register Access

To test your ability to access a register:

1. Select a general purpose register (for example, `r1` on many targets).

2. Read the register. For example:

```
> $r1
```

3. Write a different value to the same register. For example:

```
> $r1=0xdeadbeef
```

4. Read the register again and see if it has changed to the new value.

For information about testing the ability of a Green Hills Probe or SuperTrace Probe to access your target's registers, see the documentation about configuring target resources in the *Green Hills Debug Probes User's Guide*.

Testing Memory Access

To test your ability to access the target's memory:

1. If you have not run your setup script, enter **setup** in the Debugger's command pane.
2. Select a location in memory where you plan to download a program (for example, 0x8000).
3. Read the memory at this location by entering the following command:

```
> memread 4 0x8000
```

4. Write a different value to the same memory location:

```
> memwrite 4 0x8000 0xdeadbeef
```

5. Read the memory location again and see if it has changed to the new value. If it has, the debugging interface is successfully accessing your target's memory.



Note

You can also use the graphical **Memory Tester** to test your target memory. For more information, see Chapter 21, “Testing Target Memory” on page 511.

For information about additional tests you can perform if you using a Green Hills Probe or SuperTrace Probe, see the documentation about configuring target resources in the *Green Hills Debug Probes User's Guide*.

Customizing Linker Directives Files

A linker directives (**.ld**) file controls how the linker links your executable and loads it into memory. When you create a project for a target running stand-alone programs using the **Project Wizard**, it creates several linker directives files and places them in the target resources project (**tgt/resources.gpj**). The linker links any project you add to your Top Project using one of these files, depending on that project's **Program Layout** setting.

By default, the **Project Wizard** sets the **Program Layout** setting for new stand-alone program projects to **Link to and Execute out of RAM**.



Note

You can change the **Program Layout** for your project by right-clicking the program in the Project Manager and selecting **Configure**.

To edit your linker directives file:

1. In the Project Manager, double-click the linker directives file in your program's project to open it in an editor.
2. Modify and save the edited linker directives file.
3. Select the project (**.gpj**) file in the Project Manager and click  to rebuild it.

For more information, see the documentation about linker directives files in the *MULTI: Building Applications* book.

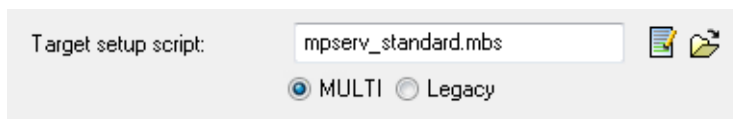
Specifying Setup Scripts

After you have a working setup script, you should run it prior to every download to ensure that your target is properly configured. The following sections explain how to specify and run board setup scripts depending on how you are connecting to your target and whether you are using a MULTI (**.mbs**) or legacy (**.dbs**) board setup script.

Using MULTI (.mbs) Setup Scripts When Connecting to Your Target

The method for specifying an **.mbs** setup script at the time of connection varies depending upon the procedure you use to connect MULTI to your target:

- To run an **.mbs** setup script every time you connect using a particular Standard Connection Method, specify the filename of the script in the **Target Setup script** field of the **Connection Editor** for the Connection Method and select the **MULTI** radio button immediately below the field.



If you are using a default Connection Method created by the **Project Wizard**, the necessary setup script file for your processor-board combination (if applicable) is specified automatically and the **MULTI** button will be selected.

- To run an **.mbs** setup script and connect to your target using a Custom Connection Method:
 - If you are editing the Custom Connection Method using the **Connection Editor**, specify the filename of the target setup script in the **Target Setup script** field.
 - If you are entering the connection command using the **Start a Custom Connection** field of the **Connection Chooser**, precede the debug server command with the option **setup=filename** (where *filename* is the **.mbs** setup script filename). Click **Connect** to continue.
- To run an **.mbs** setup script when connecting from the Debugger command pane, use the following syntax:

connect setup=filename.mbs dbserv [args]... [opts]...

where **filename.mbs** is the setup script filename, **dbserv** is the name of the debug server to be used, and **args** and **opts** are appropriate arguments for your debug server and target.

For more information about the **connect** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command

Reference” in the *MULTI: Debugging Command Reference* book and Chapter 3, “Connecting to Your Target” on page 39.

Using Legacy (.dbs) Setup Scripts When Connecting to Your Target



Note

Support for legacy (.dbs) setup scripts is deprecated and may be removed in a future release.

The method for specifying a legacy (.dbs) setup script at the time of connection varies depending upon the procedure you use to connect MULTI to your target. The various methods are:

- To run a legacy setup script every time you connect using a particular Standard Connection Method, specify the filename of the target setup script in the **Target Setup script** field of the **Connection Editor** for the Connection Method. Select the **Legacy** radio button immediately below the field.



- To run a .dbs setup script and connect to your target using a Custom Connection Method:
 - If you are editing the Custom Connection Method using the **Connection Editor**, include the **-setup filename.dbs** debug server option in the **Arguments** field.
 - If you are entering the connection command using the **Start a Custom Connection** field of the **Connection Chooser**, include the **-setup filename.dbs** debug server option in the command you enter and click **Connect**. For more information about connecting to your specific target this way, see the appropriate debug server chapter in the *MULTI: Configuring Connections* book.
- To run a .dbs setup script from the Debugger command pane, use the following syntax:

```
connect dbserv -setup filename.dbs [args]... [opts]...
```

where **filename.dbs** is the setup script filename, *dbserv* is the name of the debug server to be used, and *args* and *opts* are appropriate arguments for your debug server and target.

Running Setup Scripts Manually

In addition to running setup scripts as part of the connecting process, you can also run setup scripts manually at other times using any of the following methods:

- (MULTI **.mbs** scripts) Run your setup script file manually from the Debugger command pane using the **<** command. (For information about the **<** command, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.)
- (MULTI **.mbs** scripts) Use the MULTI **setup filename.mbs** command. If this command is used with a connection command, the setup script runs prior to downloading and debugging. The **setup** command can also be used without a specific script name if you are connected and specified a setup script when you connected. The script you specified for the connection is run if you issue the **setup** command with no specified file. For more information about the **setup** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- (Legacy **.dbs** scripts) Use the **target script filename.dbs** command from the Debugger command pane. (For more information, see the documentation about Green Hills debug server commands in the *MULTI: Configuring Connections* book.)

Early MULTI Board Setup Scripts with Debugger Hooks

Ordinarily, MULTI (**.mbs**) board setup scripts are run every time you download a program to your target, just before the download begins. However, in certain circumstances and for certain targets (such as multi-core boards), it is not appropriate to re-initialize the entire board every time you download a program to a given CPU on that board.

If you write a setup script with a comment on the first line containing the marker `MBS_OPT="early"`, the entire board setup script will be run once immediately

after you connect to your target instead of every time just before you download a program. The comment should look similar to the following:

```
// MBS_OPT="early"
```

When combined with the hook commands described in “Hook Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book, the "early" MULTI board setup script mechanism gives you fine-grained control over how and when MULTI will set up your target. For example, you can install hooks in your setup script to reinitialize the entire board upon reset by using reset hooks to cause certain initializations to take place before reset and certain others to take place afterwards.

```
addhook -before reset { /* Before reset work */ }
addhook -after reset -core 0 { /* After reset, core 0 work */ }
addhook -after reset -core 1 { /* After reset, core 1 work */ }
```

You can also add a hook to reset the board upon connecting, which causes your reset hooks to run whenever you connect to the target with MULTI.

```
addhook -after connect { reset }
```

Lastly, you might decide to reinitialize a selected subset of the board circuitry every time you download a program to one of the CPUs, while leaving the other CPUs alone.

```
addhook -before download -core 0 { /* Core 0's initialization commands */ }
addhook -before download -core 1 { /* Core 1's initialization commands */ }
```

Chapter 7

Preparing Your Target

Contents

Chapter Terminology	104
Associating Your Executable with a Connection	105
Preparing Your Target	108
Related Settings	115
Core File Debugging (Linux/Solaris only)	117

Before you can run or debug a program on your target, you must download it to the target's RAM, program it into the target's flash memory, or verify through the Debugger that it is already present on the target. This chapter describes how you can prepare your target to be debugged using one of these methods.

In addition, this chapter discusses how to open the Debugger on a core file that represents the state of your target at the time of a fatal signal. When you open the Debugger on a core file, MULTI automatically establishes a connection and prepares the target for you.

Chapter Terminology

This chapter uses terms that have specific meanings in the following sections. These terms are defined in the list below:

- Executable — Any executable; thread; or INTEGRITY application, module, or kernel that you can select from the target list for debugging. This definition only applies to this chapter.
- CPU — An entity that an executable runs on. When debugging in freeze mode, this is an actual CPU. When debugging in run mode, it is the CPU abstraction provided by the operating system, which in some cases corresponds to more than one actual CPU.
- Downloading — Writing the executable into RAM on your target.
- Flashing — Writing the executable into flash memory on your target.
- Verifying — Reading memory (either RAM or flash) from your target and comparing it to the contents of the executable, thus ensuring that the executable loaded into the Debugger is the same as the executable loaded onto your target.

For information about phrases that are used in the target list, see “Target Terminology” on page 18. For information about statuses that appear in the target list, see “The Status Column” on page 19.

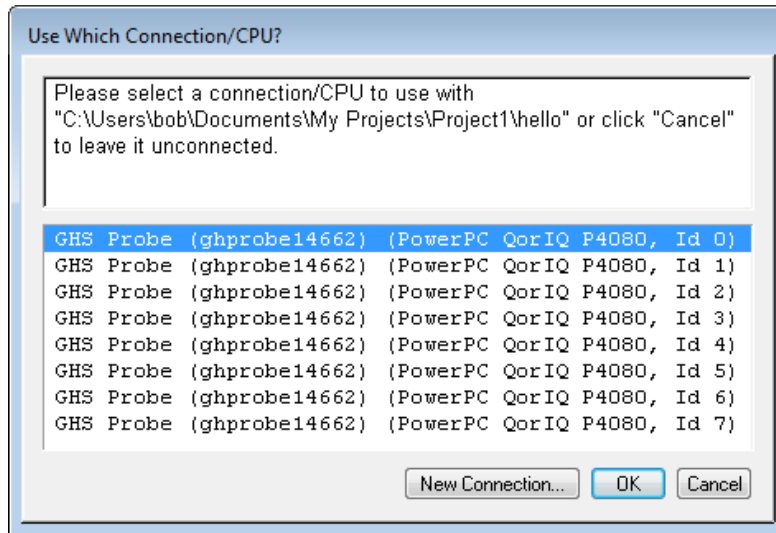
Associating Your Executable with a Connection

When you open an executable in the Debugger, the Debugger automatically associates it with a connection if one is available and applicable to the executable. Otherwise, the executable is not associated with any connection and appears in the target list under the heading **Unconnected Executables**. Before you are able to download, flash, or verify the executable, you must establish a connection to your target and associate the executable with the connection.

To do so, select the executable in the target list and then perform one of the following actions.

- Click the **Connect** button (). Using the **Connection Chooser** that appears, connect to a target that contains a CPU compatible with the executable.
- Select **Debug** → **Use Connection**. The submenu that appears lists compatible, currently active connections. If a connection appears dimmed, the current executable cannot be associated with that connection. Select an **Available** connection (if any) from the top of the menu, or select **Create New Connection**. (**Available** indicates that the connection can accept more executables. **Current** indicates that the connection is associated with the current executable. **Full** indicates that using the connection will cause another executable to stop using it. See “The Use Connection Submenu” on page 670.)
- In the Debugger command pane, enter the **change_binding bind** command. The **Connection Chooser** prompts you to establish a connection. For more information, see the **change_binding** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

After you perform one of the preceding operations, the executable is automatically associated with the connection if only one compatible CPU exists on the target you connected to. (This is usually the case.) If the selected executable is compatible with more than one CPU on the target, the **Use Which Connection/CPU?** dialog box appears. The following graphic is an example of the **Use Which Connection/CPU?** dialog box for a multi-core QorIQ P4080 target connected via a Green Hills Probe.



Select the connection that contains the correct CPU.

To disassociate the executable from the connection, select the executable and then perform one of the following actions:

- Click the **Disconnect** button () to disconnect from the target.
- Select **Debug** → **Use Connection** → **Stop Using Current Connection**.
- In the Debugger command pane, enter **change_binding unbind**. For more information, see the **change_binding** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

All the menu items listed in this section are also accessible from the shortcut menu that appears when you right-click an executable.

For more information about connecting, see Chapter 3, “Connecting to Your Target” on page 39. For information about how items are arranged in the target list after the executable is associated with the connection, see “The Target List Display” on page 17. For information about loading the executable on the target, see “Preparing Your Target” on page 108.

Updating MULTI 4 Target Connections

If you were a previous user of MULTI 4, you may need to convert your connections. Specifying a download, attach, or board setup connection mode for your target connection, as was required in MULTI 4, is not supported in MULTI 6. This section summarizes how to obtain results similar to those of:

- Selecting **Download**, **Attach**, or **Board Setup** in MULTI 4's **Connection Editor** or **Connection Chooser**
- Specifying `mode=download`, `mode=attach`, or `mode=boardsetup` in a MULTI 4 Custom Connection Method, **connect** command, or **-connect** command line option.



Tip

To remove the deprecated `mode=setting` argument from a MULTI 4 Connection Method, edit and save the connection in MULTI 6.

Download Mode (`mode=download`)

Downloading your program in MULTI 6 works in roughly the same way as in previous versions. You can open a Debugger window on your executable, connect to your target, and download your program to your target just as you could before. In MULTI 6, you do not have to specify that your connection is a download mode connection.

Attach Mode (`mode=attach`)

After connecting to your target, select **<Direct hardware access>** from the target list to get run control of your target, to read and write registers and memory, and to see the raw disassembly view of whatever your target is running (as on a newly opened MULTI 4 attach mode connection with no executable). You no longer have to specify that your connection is an attach mode connection.

If an executable that you would like to debug is already loaded onto your target, open your executable in the Debugger and do one of the following:

- Select **Debug** → **Prepare Target**, and specify **Program already present on target. Verify: Not at all.**

- Run the Debugger command **prepare_target -verify=none** in the Debugger command pane. For information about the **prepare_target** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Performing either of the preceding operations allows MULTI to assume that the program loaded on your target is the same as the executable you opened. In the target list, the **<Direct hardware access>** entry and your executable will merge together, allowing you to run, halt, and/or step your target with reference to your executable's source code in the source pane.

Board Setup Mode (mode=boardsetup)

Because MULTI 4's board setup mode is an extension of attach mode, you can connect to your target without an executable, and click **<Direct hardware access>** as described in “Attach Mode (mode=attach)” on page 107. To get some of the other effects of the deprecated board setup mode, try enabling no stack trace mode and memory sensitive mode and disabling automatic coherency checking—settings for all of which appear under **Debug → Debug Settings**.

If you are connected via a Green Hills Probe, you can also disallow access to memory (or specific areas thereof) with the **ma** command. For information about the **ma** command, see the documentation about probe commands in the *Green Hills Debug Probes User's Guide*.

Preparing Your Target

Before you can run or debug a program on your target, you must download it to the target's RAM, program it into the target's flash memory, or verify through the Debugger that it is already present on the target. To do so, select an executable in the target list and then perform one of the following operations:

- Perform a run-control operation such as stepping or running your program.
- Click the **Prepare Target** button (.
- Select the **Prepare Target** menu item from the **Debug** menu or the right-click menu.

- In the Debugger command pane, enter the **prepare_target** command. For more information and options to this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

If you have not connected to a target, the **Connection Chooser** prompts you to connect. If the selected executable is not automatically associated with the connection, the **Use Which Connection/CPU?** dialog box prompts you to pick a connection. For more information, see “Associating Your Executable with a Connection” on page 105.

After you perform one of the preceding operations, MULTI either opens the **Prepare Target** dialog box, which allows you to choose whether to download, flash, or verify the executable, or MULTI prepares your target for you by automatically executing one of these actions. For information about how MULTI determines which action to execute automatically, see “Program Types” on page 111.

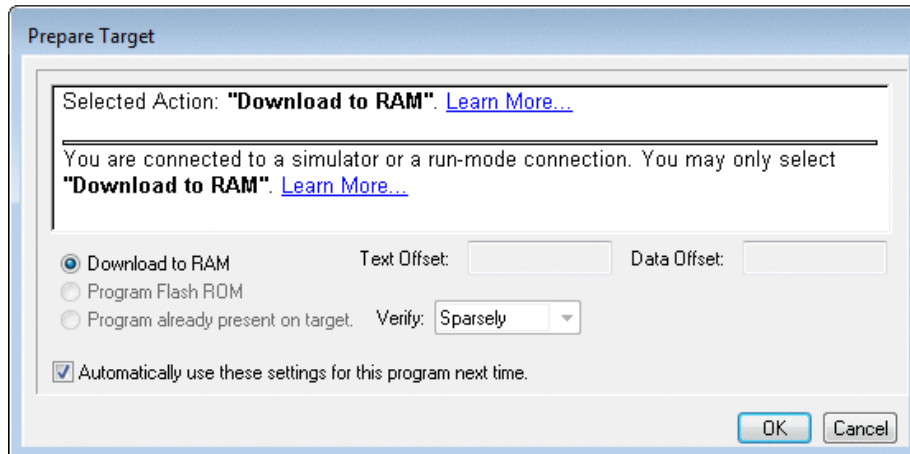
If you want to specify which action is performed (that is, you do not want MULTI to automatically download, flash, or verify the executable), select the **Prepare Target** menu item from the **Debug** menu or the right-click menu, or pass the **-ask** option to the **prepare_target** command. If you prepare the target by another means and at least one of the following items is true, MULTI automatically executes an action:

- Only one action is appropriate and MULTI does not require input (such as the text offset of a PIC program).
- The last time you prepared the target for the selected program, the option **Automatically use these settings for this program next time** was selected in the **Prepare Target** dialog box, and the program's memory layout has not changed since then. (This option is selected by default.)

For more information about the **Prepare Target** dialog box, see the next section.

The Prepare Target Dialog Box

The **Prepare Target** dialog box contains options for specifying that MULTI downloads the executable, flashes the executable, or verifies the presence of the executable on the target.



Each option in the **Prepare Target** dialog box is listed below:

- **Download to RAM** — Writes the executable into RAM on your target. For more information, see “Downloading Your Executable” on page 114.
 - **Text Offset** — Allows you to specify where the text section will be located when the executable is downloaded. This option is only available for programs using position-independent code. For more information, see “Downloading Your Executable” on page 114.
 - **Data Offset** — Allows you to specify where the data section will be located when the executable is downloaded. This option is only available for programs using position-independent data. For more information, see “Downloading Your Executable” on page 114.
- **Program Flash ROM** — Writes the executable into flash memory on your target. For more information, see “Flashing Your Executable” on page 114.
- **Program already present on target. Verify** — Specifies that the executable is already present in your target's memory. Depending on the **Verify** option that you choose from the drop-down menu, MULTI may check to ensure that the executable is on your target. For more information, see “Verifying the Presence of Your Executable” on page 114.

- **Automatically use these settings for this program next time** — Remembers the **Prepare Target** dialog box settings for this executable. If the settings remain applicable, MULTI uses them automatically and does not open the **Prepare Target** dialog box unless you request otherwise. For more information, see “Preparing Your Target” on page 108.

The options that are available in the **Prepare Target** dialog box vary according to the item you are debugging. For more information, see the next section.

When you are finished making your selections, click **OK**. MULTI prepares your target.

Program Types

MULTI understands a number of general program types, and is able to determine the action (download, flash, or verify) that should either be selected by default in the **Prepare Target** dialog box or automatically executed (for more information, see “Preparing Your Target” on page 108). The following sections provide specific information about how MULTI determines a program's type, what each type means, and what action MULTI defaults to for each type.



Note

If you are using an instruction set simulator, MULTI may determine that **Download to RAM** is the only action possible, regardless of what program type is detected. In this situation, all other options in the **Prepare Target** dialog box are disabled, and even programs built to load from ROM (that is, ROM run and ROM copy programs) should be “downloaded.”

RAM Download Programs

If MULTI is unable to locate certain special symbols that indicate the program is ROM run or ROM copy (see the following sections), it determines that the program is a RAM download program. In this case, MULTI expects that all sections marked as allocated in the ELF file should be present on the target.

The **Prepare Target** dialog box defaults to **Download to RAM** for programs of this type. No other option is generally applicable, though if the program has already

been downloaded to the target, it may be possible to verify that this is the case instead of re-downloading. If you are debugging a native target or a Dynamic Download INTEGRITY application, downloading is the only action possible. For information about downloading, see “Downloading Your Executable” on page 114. For information about verifying, see “Verifying the Presence of Your Executable” on page 114.

Only RAM download programs can have position-independent code or data, and if MULTI determines that either or both of those is present, the **Text Offset** and/or **Data Offset** text fields are available for input and must be filled in. For more information, see “Downloading Your Executable” on page 114.

ROM Run Programs

A ROM run program is stored to ROM and runs solely out of ROM (using RAM only for stack and heap space).

MULTI determines that a given program has this type if the special symbols `__ghs_rombootcodestart` and `__ghs_rombootcodeend` exist, and the symbols `__ghs_rambootcodestart` and `__ghs_rambootcodeend` do not exist. (Note that MULTI also searches for and uses various other special symbols, such as `__ghs_romstart`, `__ghs_romend`, `__ghs_ramstart`, and `__ghs_ramend`, to improve the debugging experience.)

If a program is of type ROM run, MULTI expects that the sections marked as allocated in the program's ELF file should be loaded into the target's ROM. This is generally achieved by programming the flash ROM on the target.

The **Prepare Target** dialog box defaults to **Program Flash ROM** for programs of this type. Any of the **Verify** options may also be appropriate if the program has already been programmed into the target's flash ROM. Downloading is generally not applicable, unless the program is being run on a simulator (see “Program Types” on page 111). For information about flashing, see “Flashing Your Executable” on page 114. For information about verifying, see “Verifying the Presence of Your Executable” on page 114.

ROM Copy Programs

ROM copy programs are initially located in ROM, but copy themselves to RAM during startup, and then execute partially or completely from RAM.

MULTI determines that a given program has this type if the linker-inserted symbols `__ghs_rombootcodestart`, `__ghs_rombootcodeend`, `__ghs_rambootcodestart`, and `__ghs_rambootcodeend` exist. (Note that MULTI also searches for and uses various other special symbols, such as `__ghs_romstart`, `__ghs_romend`, `__ghs_ramstart`, and `__ghs_ramend`, to improve the debugging experience.) If the special symbol `__ghs_after_romcopy` exists, MULTI is able to restore software breakpoints in the RAM image of the program after the ROM copy is completed.

If a program is of type ROM copy, MULTI expects that the sections marked as allocated in the program's ELF file should be loaded into the target's ROM. This is generally achieved by programming the flash ROM on the target.

The **Prepare Target** dialog box defaults to **Program Flash ROM** for programs of this type. Any of the **Verify** options may also be appropriate if the program has already been programmed into the target's flash ROM. Downloading is generally not applicable, unless the program is being run on a simulator (see “Program Types” on page 111). For information about flashing, see “Flashing Your Executable” on page 114. For information about verifying, see “Verifying the Presence of Your Executable” on page 114.

Unknown Programs

If MULTI is unable to determine anything about a program, the default action is **Download to RAM**. For information about downloading, see “Downloading Your Executable” on page 114.

When MULTI cannot detect any program information, it is generally the case that something is wrong with the program or its debug information.

Downloading Your Executable

To download your executable to your target's memory, select **Download to RAM** in the **Prepare Target** dialog box. This is the most commonly selected option for debugging embedded programs with MULTI.

If a board setup script is associated with the target connection, it is executed immediately before the executable is downloaded.

When the **Download to RAM** action is available and MULTI detects that your program has been linked with position-independent code and/or data, the **Text Offset** and/or **Data Offset** fields become available. These fields set the `_TEXT` and `_DATA` system variables, which allow you to specify where the text and data sections will be located when they are downloaded. If MULTI fails to detect that a program contains position-independent code or data, you can manually set the offsets using `_TEXT` and `_DATA`. For more information, see “System Variables” on page 310.

Flashing Your Executable

To flash your executable to ROM, select **Program Flash ROM** in the **Prepare Target** dialog box. The **MULTI Fast Flash Programmer** appears. This window allows you to write a memory image from the host to flash memory on the target. For more information, see Chapter 22, “Programming Flash Memory” on page 539.

After you have flashed your executable to ROM, reset your target and perform any other operations that are required for booting code from flash, such as interacting with the ROM monitor.



Note

If you flash your executable to ROM by selecting **Program Flash ROM** and **Automatically use these settings for this program next time**, clicking the **Restart** (↺) or **Prepare Target** button (⬇) or issuing an equivalent command reflashes your target.

Verifying the Presence of Your Executable

To specify that your executable is already present in the target's memory, select **Program already present on target. Verify**. Depending on the **Verify** option you

choose from the drop-down menu, MULTI may check to ensure that the contents of target memory match the contents of the executable program file.

The following list describes the **Verify** options:

- **Sparsely** — Verifies a few bytes at the beginning, middle, and end of all downloaded non-data sections that cannot be written to. The `.text` section is an example of one such section. Because certain sections of memory, such as `.bss`, `.data`, and `.heap`, may be written to during program execution, you can expect them to differ from the executable program file. When you specify this option, MULTI does not check these sections.

This option halts your target if it is running.

- **Completely** — Verifies (in entirety) all downloaded non-data sections that cannot be written to. This may take a long time.

This option halts your target if it is running.

For information about downloaded sections, see the preceding bullet point.

- **Not at all** — Assumes, but does not verify that the contents of target memory match the contents of the executable program file. This option does not halt your target.

Related Settings

Memory Sensitive Mode

To enable memory sensitive mode in the Debugger, select **Debug** → **Debug Settings** → **Memory Sensitive**. You can also select this mode by setting the system variable `$_VOLATILE=1`. For more information about system variables, see “System Variables” on page 310.

Memory sensitive mode is used when debugging targets whose memory can change in ways unexpected to the Debugger or while examining memory-mapped I/O to avoid unexpected memory references. Some ways that a target could cause such unexpected changes are self-modifying code and references to dual-port memory or memory-mapped I/O that can be changed in ways asynchronous to the program execution.

Memory sensitive mode has the following effects:

- Memory in areas where executable code resides is read from the target. When not in memory sensitive mode, MULTI assumes that executable code does not change and uses the contents of the executable file when displaying disassembly at such addresses.
- MULTI avoids reading more memory than necessary by only reading the memory locations requested and by using the specified access size. When not in memory sensitive mode, MULTI may cache some of the target memory by reading 64-byte blocks, even if only a small part of memory is actually needed.
- Only one target instruction in the Debugger pane is read and displayed at the current program location. If you want to expand the allowed range in which the Debugger can display target machine instructions, set the system variable `$_VOLATILEDISPMAX` to the desired maximum number of instruction bytes to display.

The MULTI commands **memread** and **memwrite** can also be used to perform precise memory references, even when not in memory sensitive mode. For information about these commands, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

No Stack Trace Mode

To enable no stack trace mode in the Debugger, select **Debug** → **Debug Settings** → **No Stack Trace**.

No stack trace mode disables call stack viewing and related functionality. You can also select this mode by setting the system variable `$_NOSTACKTRACE=1`.

Call stacks and local variables on a call stack are not displayed because generating a call stack could cause unexpected memory references in the case where the stack pointer is either uninitialized or points to a nonstandard stack frame. As a consequence of this, certain debugging features are also disabled. You can perform instruction single-stepping, set and hit breakpoints, and start the target executing. Other debugging functionality, such as stepping over function calls, returning from function calls, and source-level single-stepping, is not supported in this mode. Once

the target's stack is correctly set up, you can enable stack traces and the related Debugger features.

No stack trace mode is typically used in combination with memory sensitive mode (“Memory Sensitive Mode” on page 115) although it can be selected independently.

Core File Debugging (Linux/Solaris only)

Core file debugging allows you to perform static analysis of your target. The target's state is dumped into a core file when the program running on the target encounters a fatal signal.



Note

The state of the target is not dumped into a core file if you have disabled core dumps or if your program is being run under the control of a debugger.

To perform core file debugging, you can run one of the following commands from the command line, or you can enter the Debugger command **debug** or **new** in the MULTI command pane. Both the command line and Debugger commands specify the program to be debugged and its core image. When you open the Debugger on a core file, MULTI automatically establishes a connection and prepares the target.

If you are debugging a native Linux program, use the following command to start the MULTI Debugger from the command line:

```
multi program_name -C core_file
```

where:

- *program_name* is the name of the crashed program you want to debug.
- **-C** is a MULTI command line option. See Appendix C, “Command Line Reference” on page 721.
- *core_file* is the Linux/Solaris core image of the program to be debugged.

You can also use one of the following Debugger commands to open the MULTI Debugger on a core file:

- **debug *program_name core_file***

- **new** *program_name* *core_file*

See the **debug** and **new** commands in Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

Once you are attached to the process, you can view the call stack and locate the procedure that caused the dump to occur. All the operations for inspecting data (reading process registers, the stack, local and global variables, and whatever portions of memory were dumped into the core file) are available.

Part II

Basic Debugging

Chapter 8

Executing and Controlling Your Program from the Debugger

Contents

Starting and Stopping a Program	122
Single-Stepping Through a Program	123
Using Breakpoints and Tracepoints	124
Software and Hardware Breakpoint Editors	139
The Breakpoints Window	146
The Breakpoints Restore Window	152

This chapter explains how to use the MULTI Debugger to run programs and control and monitor processes on embedded and simulated targets. You can examine program details at different points during execution by halting the process manually, single-stepping through the program, and setting breakpoints.



Note

This chapter focuses primarily on how to use MULTI Debugger features through the GUI interface. However, you can also perform most of the procedures described by issuing Debugger commands in the command pane of the Debugger window. In some situations, using Debugger commands provides more specific control or more options. For a full description of the Debugger commands, which are grouped by function, see the *MULTI: Debugging Command Reference* book.

Starting and Stopping a Program

Before you can run a program, you must be connected to an embedded target, a simulator, or a native debug server; your executable must be associated with a connection; and your target must be prepared. If you are not connected to a target or simulator and you try to run a program, the **Connection Chooser** appears and prompts you to connect. For more information, see Chapter 3, “Connecting to Your Target” on page 39. In many cases, your executable is automatically associated with a connection and your target automatically prepared after you connect. If these things are not automatically done, MULTI prompts you to do them. For more information, see Chapter 7, “Preparing Your Target” on page 103.

The simplest way to run a program is to click  on the MULTI Debugger toolbar. The process executes and runs until it terminates successfully, encounters a serious error, is halted manually, or hits a breakpoint.

To halt a process manually, click . For instructions about using breakpoints to stop a process, see “Using Breakpoints and Tracepoints” on page 124. Click  to restart a halted process from the location where it stopped.

For an overview of the information you can view about your program and target during execution or while the process is halted, see “Viewing Program and Target Information” on page 168.

**Note**

You can also control execution of a program—sometimes more precisely—using Debugger commands. For more information, see Chapter 13, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

Single-Stepping Through a Program

Before you can single-step through a program, you must meet the same prerequisites that are listed in “Starting and Stopping a Program” on page 122.

The  and  buttons allow you to step through your program, executing one statement at a time. Both buttons execute the next single statement. The  button steps into function calls. It follows the execution of every individual instruction, even when functions are called. Conversely, the  button steps over function calls. It does not step into called functions. If the program halts within a called function, click  to step out of the called function.

You can also single-step through your program using Debugger commands. For more information, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

**Note**

When the Debugger stops on a conditionally not-executed instruction, the Debugger dims both the instruction and the PC pointer. (The PC pointer is the arrow marked with the word **STOPPED**.) In source display mode, a source instruction is dimmed if the current machine instruction that the Debugger is stopped on, as well as any remaining machine instructions that make up the current source statement, are conditionally not executed. In assembly display mode, the Debugger dims the instruction it is currently stopped on if the instruction is conditionally not executed. Conditionally not-executed instructions are only available on certain architectures, such as ARM.



Note

The Debugger generally completes a source-line single-step when it reaches the first instruction of a source line. The code for certain kinds of loops, such as `do {...} while` loops and infinite loops, may be generated with a backwards branch to the beginning of the loop. If debug information indicates that such a loop appears by itself on a single source line, the Debugger stops stepping after traversing the branch—even though the program may not have finished executing the entire loop—because the first instruction of the loop is also the first instruction of the source line. This is true even if the loop appears by itself on a line because of compiler or linker code transformations such as macro expansion, inlining, or code factoring. To advance past such loops with a single command, consider using the **c** or **cb** commands with an address expression. For information about these commands, see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

Using Breakpoints and Tracepoints

Breakpoints halt a process and allow you to examine your code and the state of your target at various points during program execution. You can specify conditions that trigger the breakpoint, and you can specify one or more commands to run after the breakpoint halts the process. There are two main types of breakpoints:

- *Software breakpoints* — Can be set on any executable line of code in RAM. Your process halts when it hits the software breakpoint. For more information, see “Working with Software Breakpoints” on page 128.
- *Hardware breakpoints* — Can be set on data memory locations or on executable lines of code located in RAM or ROM. When set on data memory locations, the hardware breakpoint halts your process if it reads or writes data to the specified location. Hardware breakpoint support varies by target type. For more information, see “Working with Hardware Breakpoints” on page 133.

Breakpoints can be either temporary or permanent. MULTI deletes temporary breakpoints after your process hits them. MULTI does not automatically delete permanent breakpoints.

You can easily set both software and hardware breakpoints by clicking the breakdots (•) that appear in the Debugger source pane. For more information, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 125.

On targets that use shared objects, MULTI supports breakpoints that are set in shared object code. For more information, see “Working with Shared Object Breakpoints” on page 136.

On some targets, you can also use tracepoints to collect debugging data without halting your process. For more information, see Chapter 24, “Non-Intrusive Debugging with Tracepoints” on page 559.



Note

The MULTI Debugger also supports Debugger Notes, which allow you to attach notes to any line of code. Unlike breakpoints, Debugger Notes do not halt your process or execute commands. For more information, see Chapter 10, “Using Debugger Notes” on page 173.

Breakdots, Breakpoint Markers, and Tracepoint Markers

A breakdot (•) is a small dot located directly to the left of any line of source code that corresponds to an executable instruction. A breakdot indicates that you can set one or more of a software breakpoint, hardware breakpoint, or tracepoint on the line marked with the breakdot.

By default, most breakdots in the source pane are green. But you may also encounter blue, red, or gray breakdots in the following situations:

- *Blue breakdots* (•) — Indicate source lines for which the compiler generated reordered code. For example, if you step through a function with blue breakdots at the top, you will see that the program counter starts at the first green breakdot and then goes backwards to the blue breakdots before continuing on to the rest of the green breakdots. This can happen when the compiler delays the initialization of variables for a function.
- *Red breakdots* (•) — Indicate source lines that the linker removed during link-time optimization. For example, the linker may use subroutine calls and tail merges to remove redundant segments of code from object files. Because this code has been removed, you cannot set breakpoints on these source lines.

**Note**

Link-time optimization is not available for all target processors. For more information, see the *MULTI: Building Applications* book for your target processor family.

- *Gray breakdots* (*) — Indicate source lines for which the compiler has merged the code associated with the line into another source line. This can happen when a particular source statement appears multiple times within a function. Because the code has been merged to a different source line, you cannot set breakpoints on a line with a gray breakdot. If a breakpoint is set at a source line where the code has been merged, it will be hit anytime the merged code is executed.

To set a software breakpoint on any source or assembly line marked by a breakdot, click the breakdot. To set a hardware breakpoint or a tracepoint on targets that support them, right-click any breakdot and select the appropriate action from the shortcut menu that appears.

When you set a breakpoint or tracepoint, you replace the breakdot with a breakpoint or tracepoint marker. The following table describes these markers.

Breakpoint Marker	Type of Breakpoint
	Software breakpoint
	Software breakpoint with conditions
	Software breakpoint with a bell activated
	Software breakpoint with a command list
	Software breakpoint with a breakpoint count greater than 1
	Jump breakpoint
	Any-task breakpoint (for information about any-task breakpoints in run mode, see “Any-Task Breakpoints” on page 587; for information about any-task breakpoints in freeze mode, see “Working with Freeze-Mode Breakpoints” on page 618)
	Task-specific breakpoint in freeze mode (for more information, see “Working with Freeze-Mode Breakpoints” on page 618)

Breakpoint Marker	Type of Breakpoint
	Group breakpoint for synchronized halts (for more information, see “Group Breakpoints” on page 588)
	Hardware breakpoint
	Tracepoint

If you set multiple software breakpoints on a source line, MULTI displays the marker for the enabled breakpoint that was most recently set or modified. If none of the breakpoints on the line are enabled, MULTI displays the marker for the disabled breakpoint that was most recently set or modified.

Positioning your mouse pointer over a breakpoint or tracepoint marker displays the properties of the breakpoint or tracepoint. Clicking an active breakpoint or tracepoint marker clears the breakpoint or tracepoint and changes the marker back to a breakdot. Clicking an inactive breakpoint or tracepoint marker enables the breakpoint or tracepoint. For information about clearing and enabling multiple breakpoints and tracepoints set on a single source line, see “Multiple Breakpoints and Tracepoints on a Single Line” on page 138.

Inactive breakpoints and tracepoints have gray markers. To disable a breakpoint or tracepoint, right-click the breakpoint or tracepoint marker and select **Disable Breakpoint**, **Disable Hardware Breakpoint**, or **Disable Tracepoint**. To re-enable a disabled breakpoint or tracepoint, click the gray marker. Middle-clicking a marker toggles the breakpoint or tracepoint between active and inactive status.

For more information about software breakpoints, see “Working with Software Breakpoints” on page 128. For more information about hardware breakpoints, see “Working with Hardware Breakpoints” on page 133. For more information about tracepoints, see Chapter 24, “Non-Intrusive Debugging with Tracepoints” on page 559.

Viewing Breakpoint and Tracepoint Information

The MULTI Debugger provides several ways for you to view information about the breakpoints and tracepoints you have set:

- **Tooltips** — In the source pane, positioning your mouse pointer over a breakpoint or tracepoint displays the properties of that breakpoint or tracepoint.
- **Breakpoints window** — The **Breakpoints** window displays and allows you to configure software breakpoints and, for targets that support them, hardware breakpoints, tracepoints, and shared object breakpoints. To open this window, do one of the following:
 - Click the **Breakpoints** button ().
 - In the Debugger command pane, enter the **bpview** command. For information about the **bpview** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.
 - Select **View → Breakpoints**.

Software breakpoints, hardware breakpoints, and tracepoints are displayed on separate tabs of the **Breakpoints** window. For more information, see “Viewing and Managing Software Breakpoints” on page 131, “Viewing and Managing Hardware Breakpoints” on page 135, and “The Tracepoints Tab of the Breakpoints Window” on page 571. For a comprehensive description of the columns, buttons, and shortcut menus available in the **Breakpoints** window, as well as the actions you can perform from this window, see “The Breakpoints Window” on page 146.

- **Breakpoint list commands** — You can enter breakpoint commands such as **B** to print information about specific breakpoints, or all breakpoints, to the command pane. For more information about the **B** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Working with Software Breakpoints

Software breakpoints halt your process when it reaches a specified line of code and meets the conditions you have specified. When the process is stopped, you can inspect the state of your program and track down problems. For an overview of the

types of information you can view about your program and target, see “Viewing Program and Target Information” on page 168. You can also specify that MULTI runs a list of commands when your process hits a breakpoint.

**Note**

You cannot set software breakpoints in your target's ROM because software breakpoint instructions cannot be inserted into read-only memory. However, some targets support hardware breakpoints, which can be used for debugging ROM. For more information, see “Working with Hardware Breakpoints” on page 133.

MULTI denotes a software breakpoint in the source pane with a small red stop sign (STOP). To set a software breakpoint, click a breakdot; the breakpoint marker replaces the breakdot. To move a software breakpoint, right-click the breakpoint marker and select **Move Breakpoint** (for more information, see “Moving Software Breakpoints” on page 132). To remove a breakpoint, click it; the breakpoint marker reverts to a breakdot.

**Note**

You can also set and delete breakpoints using breakpoint commands. For a full description of these commands, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

If you set a breakpoint by clicking a breakdot, the breakpoint halts your process every time the process reaches the line of code marked with the breakpoint. After you have created a breakpoint, you can use the command pane or the **Software Breakpoint Editor** to set parameters that associate conditions and commands with the breakpoint. If you set any of these additional features, the breakpoint's stop sign icon contains a symbol within it. Even if a breakpoint has more than one property, only one of these icons is displayed. To view these unique breakpoint icons, see the table in “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 125. For more information about the software breakpoint parameters you can set, see “Creating and Editing Software Breakpoints” on page 130.

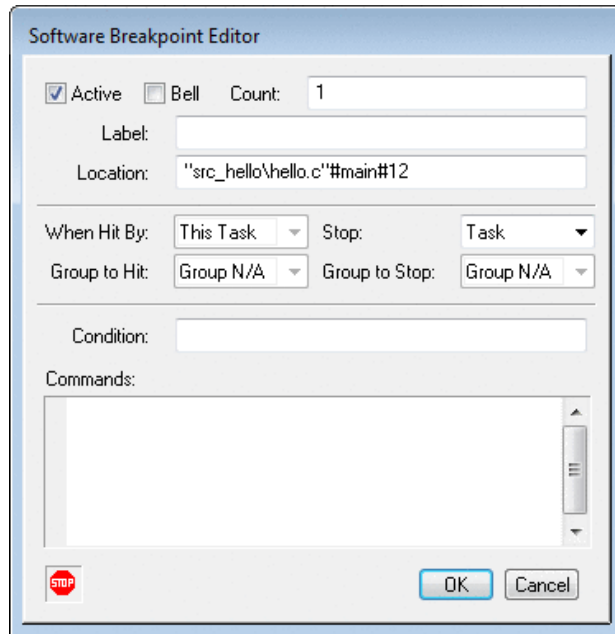
Creating and Editing Software Breakpoints

You can set a basic software breakpoint quickly and easily:

- In the source pane, click a breakdot (■). For information about breakdots, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 125.
- In the Debugger command pane, enter the **b** command. If you do not specify a location, MULTI sets the breakpoint at the location of the current line pointer. For more information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

After you have created a software breakpoint, you can add conditions and command lists to it by using the **Software Breakpoint Editor**. To open a **Software Breakpoint Editor**, do one of the following.

- In the Debugger source pane, right-click the breakpoint's stop sign icon and choose **Edit Breakpoint** from the shortcut menu.
- Click the **Breakpoints** button (■) to open the **Breakpoints** window, select the **Software** tab, and double-click the breakpoint you want to edit. For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 128.
- In the Debugger command pane, enter the **editswbp** command. For more information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.



You can set and modify all the basic properties of a software breakpoint by using the fields in this window, which are described in “Software and Hardware Breakpoint Editors” on page 139.

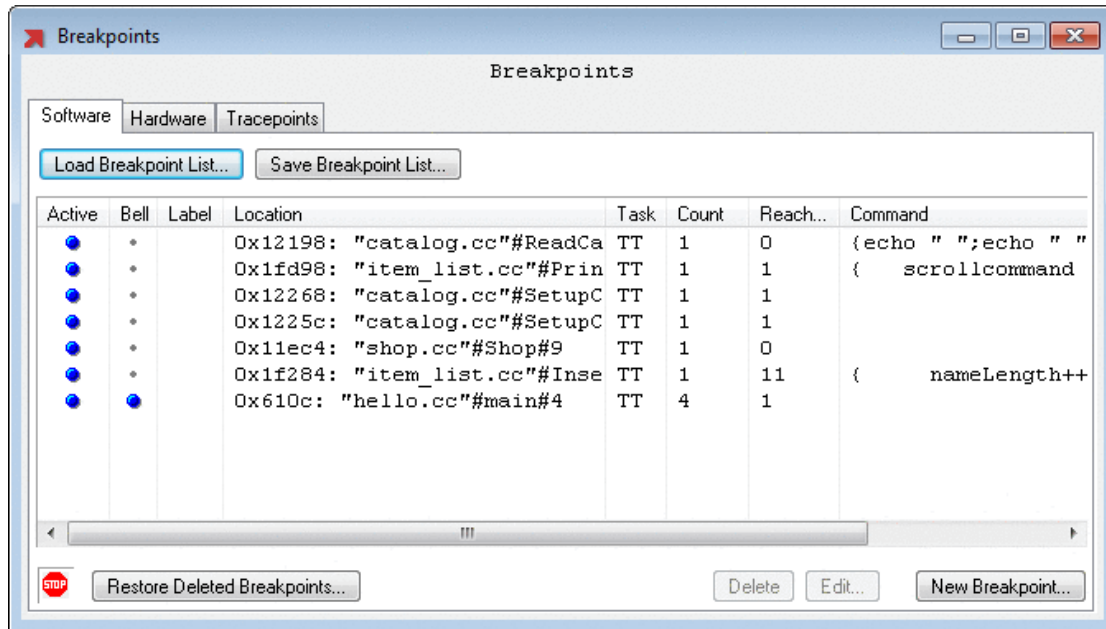


Note

You can also set software breakpoints by entering breakpoint commands in the Debugger command pane. For detailed information about these commands, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing and Managing Software Breakpoints

To view all software breakpoints for a program and access them for editing, use the **Software** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button (🛑) located on the toolbar. (For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 128.)



The **Software** tab contains a list of your program's software breakpoints and their properties. The information displayed for each breakpoint corresponds to the breakpoint properties displayed in the **Software Breakpoint Editor**. For a description of the columns and buttons that appear in the **Breakpoints** window and the actions and shortcuts available from the window, see “The Breakpoints Window” on page 146.

Moving Software Breakpoints

You can move software breakpoints easily. To move a software breakpoint, right-click the breakpoint marker and select **Move Breakpoint**. A dialog box prompts you to either click the source line where you want to move the breakpoint or cancel the operation. **MULTI** signifies the source line where your mouse is currently positioned by enclosing that line's breakdot with a small square. Click one of these locations (either on the source line or on the breakdot itself) to move the breakpoint to that line. The following list provides caveats for moving breakpoints.

- You cannot move a breakpoint out of the current function in which it resides.
- In source-only mode (**View** → **Display Mode** → **Source Only**), you cannot move a breakpoint from its original location if multiple breakpoints occur on the original source line. You must change the display mode to an assembly

mode—either **View → Display Mode → Interlaced Assembly** or **View → Display Mode → Assembly Only**. Then move the individual breakpoint using the same method as outlined previously.

- You cannot move a breakpoint to a source line that already contains a breakpoint.

Working with Hardware Breakpoints

On some targets, MULTI can set a small number of hardware breakpoints in your program. MULTI denotes a hardware breakpoint in the source pane with a small purple box (■). As with software breakpoints, you can set hardware breakpoints on your program's source or assembly lines. However, unlike software breakpoints, hardware breakpoints can be set in ROM because MULTI does not need to modify target memory to set them. You can also set hardware breakpoints on data memory locations, so that your process stops when it reads from or writes to those memory locations.

The number of hardware breakpoints you may set varies from target to target, but it is usually fewer than four. Additionally, even targets that support hardware breakpoints may not support all hardware breakpoint capabilities. For information about how your target handles hardware breakpoints if you are using a Green Hills Probe or SuperTrace Probe, see the documentation about target-specific hardware breakpoint support in the *Green Hills Debug Probes User's Guide*.

Creating and Editing Hardware Breakpoints

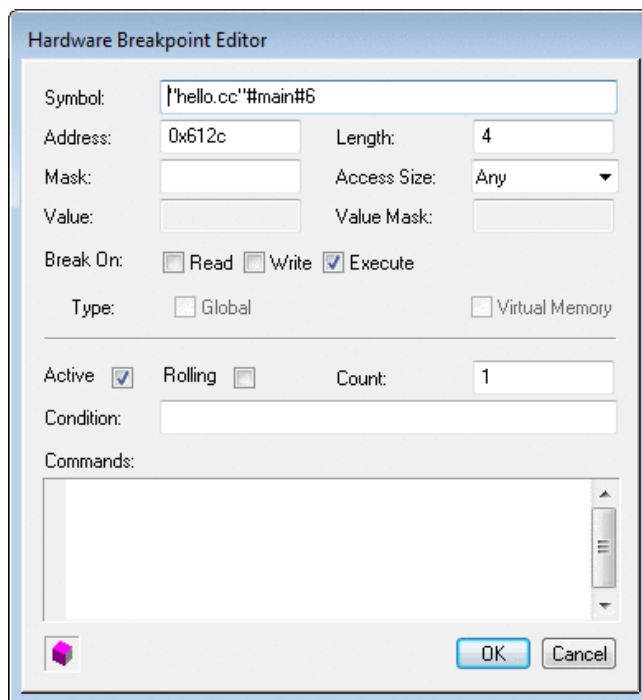
If you are connected to a target that supports hardware breakpoints, you can set a hardware breakpoint in any of the following ways.

- In the Debugger source pane, right-click a green breakdot (■) and choose **Set Hardware Breakpoint**.
- In the Debugger source pane, right-click a global or static variable and choose **Set Hardware Breakpoint**.
- In the Debugger command pane, enter the **hardbrk** command. For more information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

- On the **Hardware** tab of the **Breakpoints** window, click the **New HW Breakpoint** button.

After you have created a hardware breakpoint, you can add conditions and command lists to it using the **Hardware Breakpoint Editor**. To open a **Hardware Breakpoint Editor**, do one of the following:

- In the Debugger source pane, right-click the breakpoint's cube icon in the source pane and choose **Edit Hardware Breakpoint** from the shortcut menu.
- In the Debugger source pane, right-click a global or static variable that has a hardware breakpoint set on it, and choose **Edit Hardware Breakpoint**.
- Click the **Breakpoints** button (), select the **Hardware** tab, and double-click the breakpoint you want to edit.
- In the Debugger command pane, enter the **edithwbp** command. For information about the **edithwbp** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.



You can set and modify all the basic properties of a hardware breakpoint by using the fields in this window, which are described in “Software and Hardware Breakpoint Editors” on page 139.



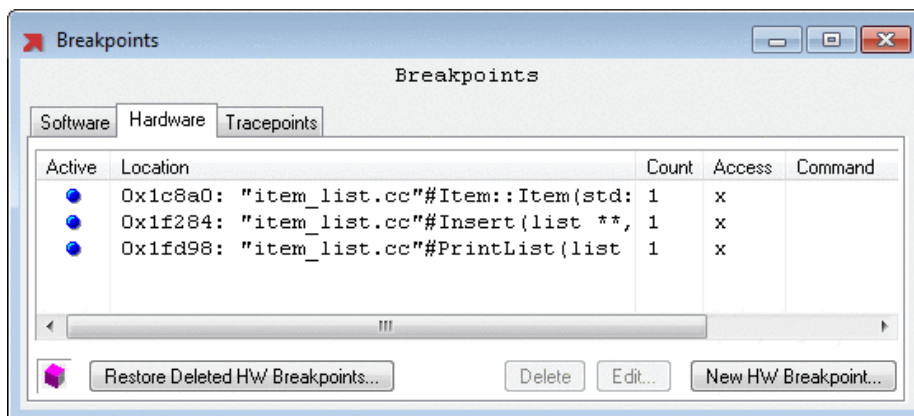
Note

You can also set hardware breakpoints by entering breakpoint commands in the Debugger command pane. For detailed information about these commands, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing and Managing Hardware Breakpoints

To work with hardware breakpoints, you must be connected to a debugging target that supports them. If you are using a Green Hills Probe or SuperTrace Probe, see the documentation about target-specific hardware breakpoint support in the *Green Hills Debug Probes User's Guide*.

To view all hardware breakpoints and access them for editing, use the **Hardware** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button (ST) located on the toolbar. (For other ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 128.)



The **Hardware** tab contains a list of all the hardware breakpoints in your program. For a description of the columns and buttons that appear in the **Breakpoints** window and the actions and shortcuts available from the window, see “The Breakpoints Window” on page 146.

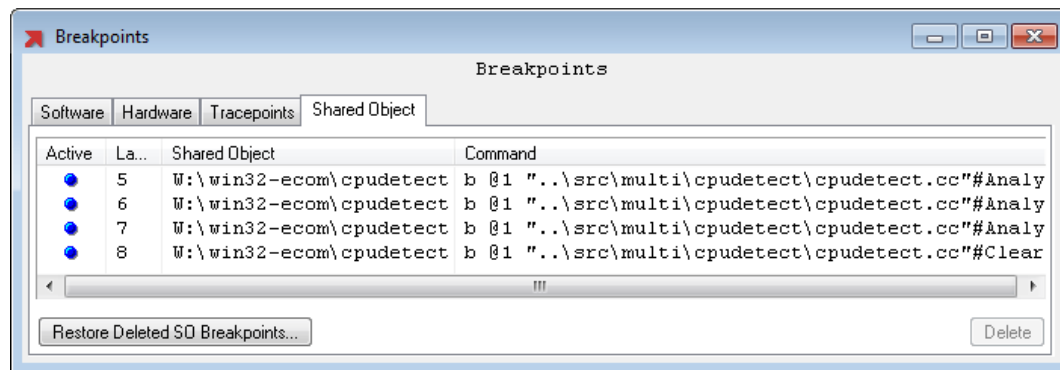
Working with Shared Object Breakpoints

When a shared object is loaded, breakpoints work normally. However, after the shared object is unloaded, you cannot set new software breakpoints in the object's code. Instead, MULTI only provides the ability to list and delete these breakpoints and to toggle them on and off. When the shared object is next loaded, MULTI restores the breakpoints, including any modifications you have made to them via the **Shared Object** tab.

Viewing and Managing Shared Object Breakpoints

When a shared object containing breakpoints is removed from the target process at run time, any breakpoints set in the shared object are shown in the **Shared Object** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button (🔍) located on the toolbar. (For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 128.)

The **Shared Object** tab is only visible when breakpoints from an unloaded shared object are available for display.



The **Shared Object** tab contains a list of all the shared object breakpoints in your program. For a description of the columns and buttons that appear in the **Breakpoints** window and the actions and shortcuts available from the window, see “The Breakpoints Window” on page 146.



Note

To use the Debugger's command pane to list shared object breakpoints, enter the **B** command. See the **B** command in Chapter 3, “Breakpoint

Command Reference” in the *MULTI: Debugging Command Reference* book.

Restoring Deleted Breakpoints

MULTI allows you to view and restore breakpoints that have been removed during the current MULTI debugging session. This is especially useful if you accidentally delete a breakpoint that took some time to set up (for example, if you created a breakpoint that is associated with a series of commands).

To view deleted breakpoints that you can restore, perform one of the following actions:

- In the Debugger command pane, enter the **dz -list** command.
- Open the **Breakpoints Restore** window by:
 - Entering the **dz -gui** command in the Debugger command pane, or by
 - Clicking the **Restore Deleted Breakpoints**, **Restore Deleted HW Breakpoints**, or **Restore Deleted SO Breakpoints** button in the bottom-left corner of the **Breakpoints** window.

From the **Breakpoints Restore** window, click the **Software** tab to view restorable software breakpoints, the **Hardware** tab to view restorable hardware breakpoints, and the **Shared Object** tab to view restorable shared object breakpoints.

To restore deleted breakpoints, perform one of the following actions:

- In the Debugger command pane, enter the **dz** command with appropriate arguments.
- Open the **Breakpoints Restore** window by using one of the methods detailed earlier in this section. From the **Breakpoints Restore** window, click the **Restore** button to restore selected breakpoints.
- In the Debugger source pane, navigate to the line where the breakpoint you want to restore was set, and right-click the breakpoint on that line. In the shortcut menu that appears, select **Restore Deleted Breakpoint** to restore the software breakpoint(s) last deleted from the line, or click **Restore Deleted Hardware Breakpoint** to restore the hardware breakpoint(s) last deleted from the line.

For complete usage information for the **dz** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book. For more information about the **Breakpoints Restore** window, see “The Breakpoints Restore Window” on page 152.

Advanced Breakpoint Topics

Multiple Breakpoints and Tracepoints on a Single Line

If some combination of software breakpoints, hardware breakpoints, and tracepoints are all set on the same source line and you click the marker to the left of the source line, *all* software breakpoints are enabled or cleared independently of hardware breakpoints and tracepoints. Similarly, *all* hardware breakpoints are enabled or cleared independently of software breakpoints and tracepoints. The same principle applies to tracepoints. Breakpoints and tracepoints are enabled or cleared based on the following criteria:

- If all software breakpoints, hardware breakpoints, or tracepoints are disabled, MULTI enables all software breakpoints, hardware breakpoints, or tracepoints, respectively.
- If at least one software breakpoint, hardware breakpoint, or tracepoint is enabled, MULTI clears all software breakpoints, hardware breakpoints, or tracepoints, respectively.

For example, suppose a single source line contains multiple software breakpoints, some of which are enabled and some disabled; one hardware breakpoint that is enabled; and one tracepoint that is disabled. If you click the marker to the left of the source line, MULTI clears all the software breakpoints, clears the hardware breakpoint, and enables the tracepoint.

Breakpoint Limitations

- Full source-level debugging is not possible within procedure prologues and epilogues, so MULTI does not display source-level breakpoints within these regions. For more information, see “The Call Stack Window and Procedure Prologues and Epilogues” on page 390.

- If you halt a process at an instruction where a software breakpoint is set, the software breakpoint may not be hit. This means that any operation associated with the breakpoint is not executed. For example, any commands set to run when the breakpoint is hit will not run; if the breakpoint is a jump breakpoint, it will not jump to the specified location; etc.
- Setting, removing, enabling, or disabling a breakpoint requires communication between the Debugger and your target. While this communication is quick, it is not instantaneous. As a result, setting or enabling a breakpoint on a running target may result in the breakpoint not being hit. Similarly, removing or disabling a breakpoint on a running target may result in the breakpoint being hit. In the second scenario, the Debugger may report `Stopped by unknown trap/breakpoint`.

This uncommon behavior is most likely to occur if you modify a breakpoint located in a tight, currently executing loop.

Software and Hardware Breakpoint Editors

The following table describes the fields and options available in the **Software Breakpoint Editor** and the **Hardware Breakpoint Editor**. The options are ordered alphabetically in the table below. Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

Editor Field	Effect
Access Size	Specifies a size for the Value and Value Mask fields (if available). This option is available in the Hardware Breakpoint Editor .
Active	Activates the breakpoint. To disable the breakpoint, clear this box. By default, breakpoints are active. When a breakpoint is active, the process stops when it hits the breakpoint and it meets the conditions of the breakpoint. When a breakpoint is inactive, it has no effect on the process. Disabling a breakpoint does not delete the breakpoint or its parameters. See also the tog command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor .

Editor Field	Effect
Bell	Enables a bell that beeps each time the process hits the software breakpoint. By default, the bell is disabled. To enable the bell, check this box. If the breakpoint does not stop the process, MULTI does not beep when it reaches the breakpoint, even if the breakpoint's bell is activated. Situations when this could occur include the following: • The breakpoint is disabled. • The breakpoint is conditional, and its condition evaluated to false. • The breakpoint is associated with a command that continues execution of the program. This option is available in the Software Breakpoint Editor.
Break On	Specifies when your process stops. Click the memory access type that represents when your process should stop. The access types are: • Read — Your process stops whenever it reads from memory at the hardware breakpoint location. • Write — Your process stops whenever it writes to memory at the hardware breakpoint location. • Execute — Your process stops whenever it fetches instructions from memory at the hardware breakpoint location. You should not set Read and/or Write in conjunction with Execute. This option is available in the Hardware Breakpoint Editor.
Commands	Specifies one or more commands to run every time your process hits the breakpoint. You can enter multiple commands by inserting a semicolon between commands or by entering each command on a separate line in the Commands text box. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.

Editor Field	Effect
Condition	Specifies the condition(s) under which the breakpoint halts your process. To set conditions for a breakpoint, enter a language expression in this field. While the condition remains false (or zero), the breakpoint does not stop your process. If your process hits the breakpoint when the condition is true (or not zero), the process stops. For software breakpoints and hardware breakpoints set on executable lines of code, the condition is evaluated in the context where the breakpoint appears. For hardware breakpoints set on data memory locations, MULTI evaluates the condition in a global context, so you can only use global variables here. Even when the condition is false, the breakpoint briefly interrupts the process so that MULTI can evaluate the condition. Therefore, a code segment containing a conditional breakpoint executes more slowly than one without such a breakpoint, even when the condition is false. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.
Count	Specifies the breakpoint's count, which is the number of times the process must hit the breakpoint before MULTI halts the process. The default breakpoint count is 1. If the breakpoint's count is 1, the process stops when it hits the breakpoint. If the count is greater than 1, the process does not stop until it has hit the breakpoint the number of times specified in this field, after which the process halts every subsequent time it hits the breakpoint. Each time the process hits the breakpoint, the breakpoint count is decremented until it reaches 1. For more information about breakpoint counts, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.
Group to Hit	Identifies what task group must hit the software breakpoint for the breakpoint to halt the process. This option is only available if your target supports task groups and you set the field When Hit By to Any Task in Group. This option is available in the Software Breakpoint Editor.

Editor Field	Effect
Group to Stop	Specifies what task group halts when a process hits the software breakpoint. The Group to Stop option is only available if you set the Stop field to Task Group on a target that supports task groups. The Group to Stop option is available in the Software Breakpoint Editor.
Label	Specifies a string you can use to refer to the software breakpoint later. Breakpoint labels are optional, but can be useful when you enter breakpoint commands in the command pane. For information about breakpoint commands, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. To assign a breakpoint label, type a word or name in the text field. Breakpoint labels cannot contain spaces or special characters. For more information about using breakpoint labels with commands, see Chapter 1, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor.
Length	Specifies the size of memory your process accesses at the specified breakpoint address. Your process hits a hardware breakpoint only if it accesses Length number of bytes at the specified address. For example, if Length is 2 bytes, then your process hits the breakpoint when a <code>short</code> value is written to the breakpoint address, but does not hit the breakpoint when a <code>char</code> or <code>int</code> value is written to the same address. If Length is greater than the maximum hardware breakpoint access size supported by your target and your target supports range hardware breakpoints, Length specifies a range of addresses where any access will hit the hardware breakpoint. This option is available in the Hardware Breakpoint Editor.
Location	Specifies the location of a breakpoint in your program's code. This is a required property of all breakpoints, so this field must contain a valid address expression. For more information, see “Using Address Expressions in Debugger Commands” in Chapter 1, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.

Editor Field	Effect
Mask	Specifies the address bits that MULTI uses when comparing the current address to the hardware breakpoint address. To compare addresses, MULTI does the following: 1. Performs a bitwise AND operation on the mask and the hardware breakpoint address. 2. Performs a bitwise AND operation on the mask and the current address. 3. Compares the results of these two operations. If the results are equal and the Value and Value Mask fields are not set, MULTI triggers the hardware breakpoint. If the Value and Value Mask fields are set, MULTI verifies them before triggering the hardware breakpoint. For example, suppose: • You define mask as: 0xFF0 (1111 1111 0000) • You define the hardware breakpoint address as: 0xCD6 (1100 1101 0110) • You do not define the Value or Value Mask Because the bitwise AND operations cause the last 4 bits of the current address and the breakpoint address to be 0000, MULTI triggers the hardware breakpoint when your process accesses memory between addresses 0xCD0 (1100 1101 0000) and 0xCDF (1100 1101 1111). The default mask is 0xFFFFFFFF, which specifies that the current address must be identical to the specified breakpoint address. This option is available in the Hardware Breakpoint Editor.
Rolling	Allows MULTI to delete the breakpoint if MULTI needs the breakpoint's resources. This option is not selected by default. This option is available in the Hardware Breakpoint Editor.

Editor Field	Effect
Stop	Specifies what stops when a process hits the software breakpoint. The available settings for this option are: • Task — The breakpoint stops the task that hits that breakpoint. • System — The breakpoint stops the whole system. • Task Group — The breakpoint stops all the tasks in the group specified by the Group to Stop field. If your target does not support these breakpoints, this setting is disabled and cannot be modified. This option is available in the Software Breakpoint Editor.
Type	Specifies a type for the hardware breakpoint. The available settings for this option are: • Global — Specifies that the hardware breakpoint can be hit by any task in the address space in which it is set. This setting is only applicable if you are using a run-mode connection to debug INTEGRITY (version 10 or later) or VxWorks. • Virtual Memory — Instructs INTEGRITY to use a virtual memory breakpoint to simulate the hardware breakpoint. This setting is only applicable if you are using a run-mode connection to debug INTEGRITY (version 10 or later). For more information, see the hardbrk command's global and vm options in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Hardware Breakpoint Editor.
Value	Compares the specified value with the value located at the hardware breakpoint address, when the process has accessed memory at that address. If the specified value, when masked with Value Mask is equal to the value located at the hardware breakpoint address, the hardware breakpoint triggers. (See the following Value Mask description.) MULTI verifies Value and Value Mask after verifying the address and Mask. This setting is only applicable on certain targets and debug servers. If you set this field and it is not applicable in your current environment, MULTI either returns an error or the system ignores the setting. When used for reads and writes less than 32 bits in size, the behavior of this field is target dependent. The behavior may also vary between big and little endian targets. This option is available in the Hardware Breakpoint Editor.

Editor Field	Effect
Value Mask	Specifies the value bits that MULTI uses when comparing the current value to the hardware breakpoint value. To compare values, MULTI does the following: 1. Performs a bitwise AND operation on the value mask and the hardware breakpoint value. 2. Performs a bitwise AND operation on the value mask and the current value of memory at the hardware breakpoint location. 3. Compares the results of these two operations. If the results are equal, MULTI triggers the hardware breakpoint. The default value mask is 0xFFFFFFFF, which specifies that the current value must be identical to the specified breakpoint value. MULTI verifies Value and Value Mask after verifying the address and Mask. This setting is only applicable on certain targets and debug servers. If you set this field and it is not applicable in your current environment, MULTI either returns an error or the system ignores the setting. When used for reads and writes less than 32 bits in size, the behavior of this field is target dependent. The behavior may also vary between big and little endian targets. This option is available in the Hardware Breakpoint Editor.
When Hit By	Identifies what must hit the software breakpoint for the breakpoint to halt the process. You can specify that the process stops when one of the following hits the breakpoint: • <code>This Task</code> — A specific task • <code>Unattached Tasks</code> — Any unattached task • <code>Attached Tasks</code> — Any attached task • <code>Any Task</code> — Any task • <code>Any Task in Group</code> — Any task in the group specified in the Group to Hit field If your target does not support these breakpoint conditions, this setting is disabled and cannot be modified. This option is available in the Software Breakpoint Editor.

The Breakpoints Window

The following sections describe the columns, buttons, and shortcut menus available in the **Breakpoints** window, as well as the actions you can perform from this window. Only the **Software**, **Hardware**, and **Shared Object** tabs are covered here. For information about the **Tracepoints** tab, see “The Tracepoints Tab of the Breakpoints Window” on page 571.



Note

Most of the actions you can perform from the **Breakpoints** window, you can also perform by entering commands in the Debugger command pane. For more information, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Breakpoints Window Columns

The following table describes breakpoint properties and how they are displayed in the **Breakpoints** window.

Columns vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The columns are ordered alphabetically in the following table.

Column	Description
Access	The memory access type, which causes your process to trigger the hardware breakpoint. Access types are: • r — (read) Your process stops whenever it reads from memory at the hardware breakpoint location. • w — (write) Your process stops whenever it writes to memory at the hardware breakpoint location. • rw — (read/write) Your process stops whenever it either reads from or writes to memory at the hardware breakpoint location. • x — (execute) Your process stops whenever it fetches instructions from memory at the hardware breakpoint location. This column is available on the Hardware tab.

Column	Description
Active	A large blue dot (●) indicates the breakpoint is active. A small gray dot (*) indicates the breakpoint is inactive. To toggle between the two states, click the dot. This column is available on the Software, Hardware, and Shared Object tabs.
Bell	A large blue dot (●) indicates the bell is active. A small gray dot (*) indicates the bell is inactive. To toggle between the two states, click the dot. This column is available on the Software tab.
Command	On the Software and Hardware tabs: the commands that run every time your process hits the breakpoint. On the Shared Object tab: the command that restores each shared object breakpoint when the shared object is loaded again.
Count	If MULTI is counting the number of times the process hits the breakpoint (this is the common case): the breakpoint's current count, which indicates how many more times the process must hit the breakpoint before the process halts. If the count is 1, the process halts every time the target hits the breakpoint. If the target is counting the number of times the process hits the breakpoint (for example, if you are using INTEGRITY 10 with a run-mode connection): the behavior of this column is undefined. This column is available on the Software and Hardware tabs.
Label	The label (if any) of the breakpoint. Breakpoint labels cannot contain spaces or special characters. This column is available on the Software and Shared Object tabs.
Location	The location of the breakpoint in your program's code. For more information, see “Using Address Expressions in Debugger Commands” in Chapter 1, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book. This column is available on the Software and Hardware tabs.
Reached	If MULTI is counting (this is the common case): the number of times your process has hit the breakpoint. If the target is counting (for example, if you are using INTEGRITY 10 with a run-mode connection): the behavior of this column is undefined. This column is available on the Software tab.

Column	Description
Shared Object	The shared object that each breakpoint is set in. This column is available on the Shared Object tab.
Task	Identifies what must hit the breakpoint for the breakpoint to halt your process, and also what tasks halt when your process hits the breakpoint. This column is available on the Software tab.

Breakpoints Window Buttons

In addition to displaying information about all breakpoints, the **Breakpoints** window contains buttons that make it easy for you to do things such as create, modify, and load breakpoints. Some operations can be performed on multiple breakpoints simultaneously.

Buttons vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The buttons are ordered alphabetically in the following table.

Button	Action
Delete	Deletes the selected breakpoint(s). This button is available on the Software , Hardware , and Shared Object tabs.
Edit	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. For more information about setting software breakpoint parameters, see “Creating and Editing Software Breakpoints” on page 130. For more information about setting hardware breakpoint parameters, see “Creating and Editing Hardware Breakpoints” on page 133. This button is available on the Software and Hardware tabs.
Load Breakpoint List	Opens a Load Breakpoints dialog box where you can choose to load the file in which you previously saved breakpoints. (See Save Breakpoint List below.) Note: The Debugger may not be able to load group breakpoints. This button is available on the Software tab.

Button	Action
New Breakpoint or New HW Breakpoint	Creates a new breakpoint at the location of the current line pointer and opens a Software Breakpoint Editor or Hardware Breakpoint Editor on the new breakpoint. This button is available on the Software and Hardware tabs.
Restore Deleted Breakpoints or Restore Deleted HW Breakpoints or Restore Deleted SO Breakpoints	Opens the Breakpoints Restore window, which allows you to view and restore breakpoints that have been removed during the current MULTI debugging session. For more information about this window, see “The Breakpoints Restore Window” on page 152. This button is available on the Software , Hardware , and Shared Object tabs.
Save Breakpoint List	Opens a Save Breakpoints dialog box where you can choose a file in which to save your program's currently set breakpoints. You can reload the file later. Note that the Debugger may not be able to reload group breakpoints. (See Load Breakpoint List above.) This button is available on the Software tab.

Breakpoints Window Mouse and Keyboard Actions

The following table describes the mouse and keyboard actions you can perform on items located in the **Breakpoints** window. Possible actions vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The actions are ordered alphabetically in the following table.

Action	Result
Click a breakpoint	Displays the breakpoint's location in the Debugger, if the breakpoint is set in source code. This action is possible on the Software and Hardware tabs.
Double-click a breakpoint or Press Enter	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. This action is possible on the Software and Hardware tabs.

Action	Result
Click the Active dot	Toggles the state of the selected breakpoint between active and inactive. See also the tog command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This action is possible on the Software , Hardware , and Shared Object tabs.
Click the Bell dot	Toggles the state of the bell between enabled and disabled. If enabled on a specific breakpoint, the bell beeps every time the process stops at that breakpoint. This action is possible on the Software tab.
Right-click a breakpoint	Opens a shortcut menu. Available menu options are described in the following table. This action is possible on the Software , Hardware , and Shared Object tabs.
Press Delete	Deletes the selected breakpoint(s). This action is possible on the Software , Hardware , and Shared Object tabs.

Breakpoints Window Shortcut Menu

The following table describes the menu options that appear when you right-click a breakpoint in the **Breakpoints** window. Menu options vary according to the type of breakpoint clicked (software, hardware, or shared object). The menu options are ordered alphabetically in the following table.

Menu Item	Action
Beep on Selected Breakpoints	Enables the breakpoint bell for the selected breakpoints. A beep sounds when the process stops at these specified breakpoints. This menu item is available from the Software tab shortcut menu.
Delete Selected Breakpoints or Delete Selected Shared Object Breakpoints	Deletes the selected breakpoint(s). This menu item is available from the Software , Hardware , and Shared Object tab shortcut menus.

Menu Item	Action
Disable Selected Breakpoints or Disable Selected Shared Object Breakpoints	Inactivates all the selected breakpoints so that the process does not stop when it hits them. See also the tog command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This menu item is available from the Software, Hardware, and Shared Object tab shortcut menus.
Do not Beep on Selected Breakpoints	Disables the breakpoint bell for the selected breakpoints. No sound is emitted when the process stops at these specified breakpoints. This menu item is available from the Software tab shortcut menu.
Edit Selected Breakpoint	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor, which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. This menu item is available from the Software and Hardware tab shortcut menus.
Enable Selected Breakpoints or Enable Selected Shared Object Breakpoints	Activates all the selected breakpoints so that the process stops when it hits them. See also the tog command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This menu item is available from the Software, Hardware, and Shared Object tab shortcut menus.
New Software Breakpoint or New Hardware Breakpoint	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor, which you can use to create a new software or hardware breakpoint with the properties that you specify. This menu item is available from the Software and Hardware tab shortcut menus.
Show In Debugger	Displays the source code where the selected breakpoint is set. This menu item is available from the Software and Hardware tab shortcut menus.

The Breakpoints Restore Window

The following sections describe the columns, buttons, and shortcut menus available in the **Breakpoints Restore** window. For general information about restoring breakpoints, see “Restoring Deleted Breakpoints” on page 137.



Note

Most of the actions you can perform from the **Breakpoints Restore** window, you can also perform by using the **dz** Debugger command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Breakpoints Restore Window Columns

The following table describes how properties of deleted breakpoints are displayed in the **Breakpoints Restore** window.

Columns vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The columns are ordered alphabetically in the following table.

Column	Description
Command	The command that was used to create the deleted breakpoint. This column is available on the Software , Hardware , and Shared Object tabs.
Label	An identifier for use with the dz command (see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book). This column is available on the Shared Object tab.
Location	The source location (if applicable) and address that the breakpoint existed at. This column is available on the Software , Hardware , and Shared Object tabs.
Shared Object	The shared object that the deleted shared object breakpoint was set on. This column is available on the Shared Object tab.

Breakpoints Restore Window Buttons

In addition to displaying information about breakpoints that have been deleted, the **Breakpoints Restore** window contains buttons that make it easy for you to restore or permanently remove deleted breakpoints. The buttons available in the **Breakpoints Restore** window are described in the following table.

Button	Action
All	Selects all breakpoints listed on the current tab.
None	Deselects all breakpoints listed on the current tab.
Clear	Clears all selected breakpoints from the current tab. (Once cleared, the breakpoints can never be restored.) This button corresponds to the dz-clear command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Restore	Restores all selected breakpoints listed on the current tab. This button corresponds to the dz command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Breakpoints Restore Window Shortcut Menu

The following table describes the menu options that appear when you right-click a deleted breakpoint entry in the **Breakpoints Restore** window. Menu options vary according to the type of breakpoint clicked (software, hardware, or shared object). The menu options are ordered alphabetically in the following table.

Menu Item	Action
Clear Selected Breakpoints	Clears all selected breakpoints from the current tab. (Once cleared, the breakpoints can never be restored.)
or	
Clear Selected Shared Object Breakpoints	This menu item corresponds to the dz-clear command and is available from the Software , Hardware , and Shared Object tab shortcut menus.

Menu Item	Action
Restore Selected Breakpoints or Restore Selected Shared Object Breakpoints	Restores all selected breakpoints listed on the current tab. This menu item corresponds to the dz command and is available from the Software , Hardware , and Shared Object tab shortcut menus.
Show Location In Debugger	Jumps to the location in the Debugger source pane where the deleted breakpoint was set. This menu item is available from the Software and Hardware tab shortcut menus.

For complete information about the **dz** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Chapter 9

Navigating Windows and Viewing Information

Contents

Navigating and Searching Basics	156
Navigating, Browsing, and Searching the Source Pane	162
Viewing Program and Target Information	168

This chapter describes how to navigate and browse MULTI Debugger windows.

Navigating and Searching Basics

The following sections explain features and procedures common to most MULTI Debugger windows.

Using the Scroll Bar

- You can either use the scroll bar or enter the **scrollcommand** command to scroll through your program. For more information about this command, see Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- For information about navigating the source pane, see “Navigating, Browsing, and Searching the Source Pane” on page 162.

Infinite Scrolling

In two situations, normal scrolling behavior is replaced by *infinite scrolling*. In this mode, the thumb in the vertical scroll bar stays fixed in the middle of the scroll bar, does not reflect the relative position of the display, and cannot be dragged to new locations. The scroll thumb's appearance is also different. On Windows, the Debugger sets the thumb at its minimum size. On Linux/Solaris, the Debugger replaces the thumb with a diamond. You can still click above or below the thumb or use the arrow keys to scroll up or down. Infinite scrolling occurs in the following situations:

- When the Debugger displays only assembly code in the source pane. See “Source Pane Display Modes” on page 23.
- In a **Memory View** window. See Chapter 15, “Using the Memory View Window” on page 323.

Incremental Searching

You can perform incremental text searches in most MULTI debug windows. (In some windows, you must select something before you can search.) To start an

incremental forward search, press **Ctrl+F**. To start an incremental backward search, press **Ctrl+B**. When you start a search, `srch` appears on the left side of the window's status bar.

After you press **Ctrl+F** or **Ctrl+B** and start typing, MULTI begins searching the active window for the string of characters and highlights the first match it finds. If subsequent keystrokes do not match the first selection, MULTI continues to search the active window for an exact match. The search string appears to the right of `srch` in the status bar. In most windows, the search starts with the text that the window currently displays. In the source pane, the search begins at the current line pointer.

To find the next or previous search match, press **Ctrl+F** or **Ctrl+B** again. Incremental searches wrap around the entire text buffer of the window you are searching. When the Debugger reaches the end or the beginning of the window buffer, it beeps to indicate that it is about to wrap.

To end a search, press **Esc** or **Enter**.

You can switch the search mode for a single search from case-insensitive to case-sensitive by typing uppercase characters in the search string. To change the default case sensitivity for all incremental searches in the current session, enter the **chgcse** command in the Debugger command pane. For more information about this command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

The following table summarizes the shortcuts and keystrokes available for incremental searching.

Keyboard Shortcut	Effect
Ctrl+F	Starts an incremental forward search or, if you have already started a search, advances to the next matching pattern. If you have not started searching and you press Ctrl+F twice, you resume the last search.
Ctrl+B	Starts an incremental backward search or, if you have already started a search, jumps backward to the previous matching pattern. If you have not started searching and you press Ctrl+B twice, you resume the last search.
Ctrl+U (while in search mode)	Resets the search pattern.

Keyboard Shortcut	Effect
Esc or Enter (while in search mode)	Ends the search. In source pane searches, the last match remains selected.
<i>any_character</i> (while in search mode)	Adds a character to the search pattern.
Backspace (while in search mode)	Deletes the last character in the search pattern.



Note

You can also enter search commands in the command pane or use the **Source Pane Search** dialog box to perform incremental searches in the source pane. For more information, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book and “Using the Source Pane Search Dialog Box” on page 167.

Example 9.1. Searching for a String

If the source pane of an active Debugger window contains the following text:

```
this is a search string.
```

and you start a search by pressing **Ctrl+F** and typing the letter *i*, the Debugger highlights the character *i* in the word *this*.

If you then type *s*, the Debugger highlights the two characters *i* and *s* in the word *this*.

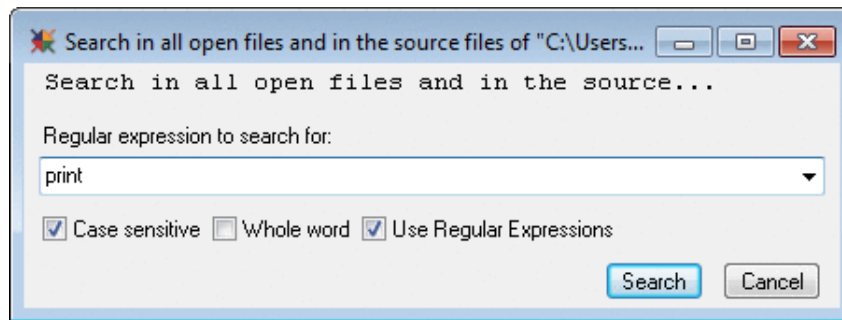
To jump to the next occurrence of the pattern *is*, press **Ctrl+F** again. The Debugger selects the word *is*.

To search next for the pattern *in*, press **Backspace** to reset the search string to *i*, and then type *n*. The Debugger highlights the characters *i* and *n* in *string*.

To stop the search, press **Enter**. The *in* in *string* remains selected.

Searching in Files

The MULTI Debugger supports full-text searching of open files and, if debugging information is available, of all the files that make up a program. To search your source files, select **Tools** → **Search in Files**. A Search in Files dialog box appears.



Enter your search string in the text box or use the drop-down list to select recent search strings. You can also select any of the following optional search criteria.

- **Case sensitive** — The search only finds text that matches the case of the search string exactly. If this box is cleared, the search ignores case when searching for a match. By default, this box is checked.
- **Whole word** — The search only finds text that contains the search terms as words. This means that the matching string must be preceded by a non-word character and followed by a non-word character, where word characters are letters, digits, and the underscore. For example, if you select this check box, a search for `ice` does not match `slice` or `ice__`, but it does match `ice-9`. This box is cleared by default.
- **Use Regular Expressions** — The search treats the text you enter as a regular expression. If this box is cleared, the search treats the text you enter as a fixed string. By default, this box is checked.

After you have entered the search string and set the search criteria, click **Search** to perform the search. A **Search in Files Results** window opens to show the search progress. For a description of this window, see “Viewing Search in Files Results” in Chapter 4, “Editing Files with the MULTI Editor” in the *MULTI: Managing Projects and Configuring the IDE* book.

**Note**

To access the same Search in Files capability without using a GUI interface, use the **grep** command. For more information about this command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

**Note**

The Search in Files capability runs the BSD **grep** utility. A copy of BSD **grep** is installed along with the MULTI IDE. However, BSD **grep** is not part of MULTI and is not distributed under the same license as MULTI. For more information about the license under which BSD **grep** is distributed, refer to the file **bsdgrep.txt**, which is located in the **copyright** subdirectory of the IDE installation directory. For information about the search expression format that BSD **grep** uses, refer to the OpenBSD `re_format(7)` man page.

Selecting, Cutting, and Pasting Text

In most MULTI windows, you can select and copy text using your mouse and common keyboard shortcuts. Some windows, however, do not support text selection, copying, and/or pasting. For more information about whether a specific window supports text selection and modification, use the index to find the main discussion of that window in this book. Conventions for selecting, copying, and pasting text differ in the source and command panes of the main Debugger window. For more information, see “Selecting, Copying, and Pasting Text in the Main Debugger Window” on page 161.

The following table describes general conventions for selecting, cutting, and pasting text in MULTI Debugger windows.

To	Do this
Select text	Use your mouse pointer to highlight text.
Copy text	Select text, and press Ctrl+C .
Paste text	Copy text, click in the spot where you want to paste the text, and press Ctrl+V .
Deselect text	Click in an area of the window without any text.

**Note**

On Windows, the copy and paste functions use the Windows clipboard, so you can copy and paste text among any applications that use the Windows clipboard.

On Linux/Solaris, you can copy and paste your selection to other X Window System applications that support pasting, such as `xterms`. However, different applications may behave differently, so consult the reference manual for your specific system. Generally, when you select text, `MULTI` automatically copies it. Middle-click to paste automatically copied text. Text cannot be selected in two windows at the same time.

Selecting Items from Lists

In those windows containing lists of items that you can select, the following conventions usually apply:

- To select an item, single-click the row containing that item. This highlights the row.
- To select multiple, non-adjacent items/rows, hold the **Ctrl** key while clicking each item or row.
- To select a range of adjacent rows, click the first row, hold the **Shift** key, and then click the last row in the range.

Selecting, Copying, and Pasting Text in the Main Debugger Window

Conventions for selecting, copying, and pasting text in the source and command panes differ from the conventions used in other Debugger windows. In the source pane, your selection automatically expands to include a whole entity, such as a word or expression. For example, to select a word in the source pane, simply click anywhere within the word. To select all the text occurring between a pair of parentheses, including the parentheses themselves, drag your mouse over one of the parenthesis markers and release.

When selecting text in the source pane, you may find that the Debugger performs a command as soon as you release the mouse button. To prevent the Debugger from

issuing a command, hold down the **Ctrl** key while selecting text. This also prevents the Debugger from attempting to auto-complete your selection. Alternatively, you can configure Debugger key bindings to prevent the Debugger from issuing commands. See “Customizing Keys and Mouse Behavior” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

When you select text in the source pane, the Debugger automatically copies your selection. To paste the selection into the command pane, middle-click in the command pane. To enable this behavior on Windows, select **Config** → **Options** → **MULTI Editor** tab, and check **Allow middle click to paste**. In general, this behavior is similar to selecting and pasting procedures on Linux/Solaris.



Note

On Windows, the automatic copy that the Debugger makes at the time of selection only allows you to paste to the command pane. To copy text and paste it into another window, highlight the text and press **Ctrl+C** to copy it. Then press **Ctrl+V** to paste it into another window.

Navigating, Browsing, and Searching the Source Pane

You can use the following shortcuts, buttons, commands, and menu items to navigate through your program code in the source pane. For information about scrolling, see “Using the Scroll Bar” on page 156.

To	Do (one of) the following
Scroll up by one screen	• Press PageUp.
Scroll down by one screen	• Press PageDown.
Scroll up by one line	• Press Shift+UpArrow.
Scroll down by one line	• Press Shift+DownArrow.
View the code one level higher on the call stack	• Press Ctrl++ (plus sign). • Click • Select View → Navigation → UpStack.

To	Do (one of) the following
View the code one level lower on the call stack	• Press Ctrl+ (minus sign). • Click . • Select View → Navigation → DownStack.
View the procedure where the process is stopped	• Click . • In the command pane, enter the E command. For information about the E command, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Select View → Navigation → Current PC.
View the first procedure higher on the call stack that has source code (for example, if you are stopped inside a library function with no source code and you want to return to viewing your program)	• In the command pane, enter the uptosource command. For information about the uptosource command, see Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Select View → Navigation → UpStack To Source.
Navigate to a specific file, procedure, line number, breakpoint, or other location	• In the command pane, enter the e command followed by an address expression. For information about the e command, see Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • On the navigation bar, use the File Locator to navigate to a file. See “Using the File Locator” on page 164. • On the navigation bar, use the Procedure Locator to navigate to a procedure. See “Using the Procedure Locator” on page 165. • Select View → Navigation → Goto Location and enter an appropriate location in the dialog box.

To	Do (one of) the following
Return to a location you recently viewed in the source pane	- In the command pane, enter the indexprev and indexnext commands. For information about these commands, see Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. - On the navigation bar, use the Back (↶) and Forward (↷) history buttons.

Using the File Locator

The File Locator (File: `src_trace\trace_demo.c` ▾) shows the name of the file currently displayed in the Debugger and allows you to navigate quickly to other files in your program. The following table summarizes what you can do with the File Locator.

To	Do this
View the full name and path of the displayed file	Move the cursor over the File Locator. The file path appears in a tooltip.
Navigate to another source file in your program	Type the name of the file in the File Locator and press Enter. If the name you type is of a valid source file that was compiled into your program, the Debugger navigates to the file and displays it in the source pane. If the name you type does not match a filename exactly, the Debugger displays a file whose name begins with the name that you typed, or if there is more than one match, it displays a list of files. If you do not remember the exact name of the file you are looking for, use a wildcard (see “Wildcards” on page 303). For example, if you know that a file's name contains the word <code>sort</code>, enter <code>*sort*</code> in the File Locator to browse a list of all the files in your program that contain the word <code>sort</code> in their names. If only one file in your program matches the pattern that you type, the Debugger displays that file immediately. You can use any number of wildcard characters in your search. Wildcard searching is not case-sensitive.
View a file recently displayed in the source pane	Click the arrow on the right side of the File Locator to open a drop-down list of recently viewed files. The File Locator sorts the files according to when you last viewed them, with the currently displayed file at the top of the list. To display a file in the source pane, select it in the list.

To	Do this
Browse a list of all the files in your program	Do one of the following: • In the command pane, enter the browse files command. For information about the browse command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Click the arrow on the right side of the File Locator and select Browse all source files in program from the list that appears. • Select Browse → Files. Performing any of the above actions opens a Source Files with Procedures window. To display a file in the source pane, click the name of the file in the list.

See also “Browsing Source Files” on page 230.

Using the Procedure Locator

The Procedure Locator (Proc: ▼) shows the name of the procedure currently displayed in the Debugger and allows you to navigate quickly to other procedures in your program. The following table summarizes what you can do with the Procedure Locator.

To	Do this
View the full name and signature of the displayed procedure	Move the mouse pointer over the Procedure Locator. The full name and signature appears in a tooltip. Full signature display is only available for C++ procedures. The return type of the procedure is not included in the signature display.

To	Do this
Navigate to another procedure in your program	Type the name of the procedure in the Procedure Locator and press Enter. If the name you type is of a valid procedure that was compiled into your program, the Debugger navigates to the procedure and displays it in the source pane. If the name you type does not match a procedure name exactly, the Debugger displays a procedure whose name begins with the name that you typed, or if there is more than one match, it displays a list of procedures. If you do not remember the exact name of the procedure you are looking for, use a wildcard (see “Wildcards” on page 303). For example, if you know that a procedure's name contains the word <code>sort</code>, enter <code>*sort*</code> in the Procedure Locator to browse a list of all the procedures in your program that contain the word <code>sort</code> in their names. If only one procedure in your program matches the pattern that you type, the Debugger displays that procedure immediately. You can use any number of wildcard characters in your search. Wildcard searching is not case-sensitive.
View a procedure recently displayed in the source pane	Click the arrow on the right side of the Procedure Locator to open a drop-down list of recently viewed files. The Procedure Locator sorts the procedures according to when you last viewed them, with the currently displayed procedure at the top of the list. If more than one procedure is visible in the source pane, the name of the procedure pointed to by the blue current line pointer is displayed.
Browse a list of all the procedures in the displayed file	Click the arrow on the right side of the Procedure Locator and select Browse procedures in current file from the list that appears. The Procedures: <i>current_file</i> window opens. To display a procedure in the source pane, click the name of the procedure in the list.
Browse a list of all the procedures in your program	Do one of the following: • In the command pane, enter the browse procs command. For information about the browse command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Click the arrow on the right side of the Procedure Locator and select Browse procedures in program from the list that appears. • Select Browse → Procedures. Performing any of the above actions opens a Procedures Browse window. To display a procedure in the source pane, click the name of the procedure in the list.

For more information about browsing procedures, see “Browsing Procedures” on page 220.

Using Navigation History Buttons

The Debugger keeps a history of the source locations you have viewed. You can use the **Back** (↶) and **Forward** (↷) buttons to display source code you have recently viewed in the source pane. These buttons function similarly to the back and forward buttons found in Web browsers. If you click and hold either of these buttons, you can select from a short list of previously visited source locations.

See also the **indexnext** and **indexprev** commands in Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.

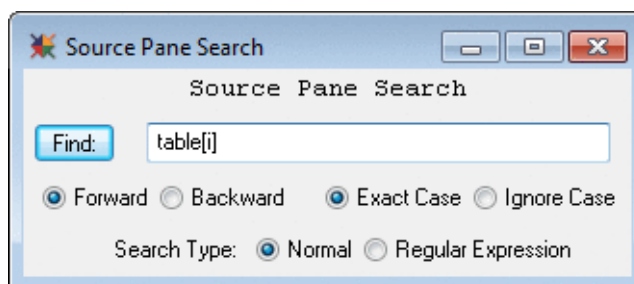
Using the Source Pane Search Dialog Box

The **Source Pane Search** dialog box provides a simple way for you to search your code quickly and easily. To open the **Source Pane Search** dialog box, do one of the following.

- In the command pane, enter the **dialogsearch** command. For information about the **dialogsearch** command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.
- From the main Debugger window, select **Tools** → **Search**.
- Use the **Ctrl+Shift+F** keyboard shortcut.

To search:

1. Enter the text you are searching for in the **Find** field.



2. Use the radio buttons to specify the direction of the search (**Forward** or **Backward**), the case sensitivity of the search (**Exact Case** or **Ignore Case**), and the type of search (**Normal** or **Regular Expression**). The default settings are **Forward**, **Exact Case**, and **Normal**.

3. Click **Find**.

MULTI highlights the first match of the search string in the source pane. To find the next match, click **Find** again. The search wraps at the end of the source file.



Note

You can also search the source pane incrementally by using simple keyboard shortcuts (see “Incremental Searching” on page 156) or by entering the **fsearch** or **bsearch** commands in the command pane. For information about these and about other commands that allow you to search from the command pane, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing Program and Target Information

In addition to viewing information in the source pane, you can use stand-alone windows to view more detailed information about your program and the state of your target during program execution. The following sections provide a brief introduction to the types of information you can view, and in some cases, edit.

Viewing Information in Stand-Alone Windows

The MULTI Debugger includes view windows you can use to examine aspects of your source code, debugging information, and target memory information. The following table briefly describes these windows and how to open them from the main Debugger window. For information about the main Debugger window, see Chapter 2, “The Main Debugger Window” on page 11. Each window listed in the following table is described in more detail later in this book.

For general information about navigating MULTI Debugger windows, see “Navigating and Searching Basics” on page 156.

\$locals\$ window

Displays information about variables local to the current function. If the current function is a C++ instance method, the window also displays information about the `this` pointer.

The current function is the function where the program counter (PC) pointer is located. If the PC pointer moves to a new function, either by running to a new function or by viewing a different call stack frame, the content of the **\$locals\$** window changes to display local variables for that function. For information about the PC pointer, see “The Source Pane” on page 21.

To open this window, click the **Locals** button ().

The **\$locals\$** window is a specialized Data Explorer window. For information about the Data Explorer, see Chapter 11, “Viewing and Modifying Variables with the Data Explorer” on page 183.

Breakpoints window

Displays and allows you to configure software breakpoints, hardware breakpoints, and tracepoints.

To open this window, click the **Breakpoints** button ().

For more information about the contents and features of this window, see “The Breakpoints Window” on page 146.

Browse window

Allows you to browse information about procedures, global variables, source files, data types, and cross references.

To open this window, select **Procedures**, **Globals**, **Files**, or **All Types** from the **Browse** menu.

For more information, see “The Browse Window” on page 214.

Call Stack window

Lists the functions currently on the call stack. In addition to naming each function, this window provides the name and value of each argument.

To open the **Call Stack** window, click the **Call Stack** button ().

For more information, see “Viewing Call Stacks” on page 388.

Data Explorer window

Displays a variable, its type, and its current value.

To open a Data Explorer window, double-click a variable in the source pane. The variable may be of any type, whether an integer, structure, array, or class. If the Data Explorer contains a pointer, double-clicking the pointer opens a new Data Explorer window and displays the variable pointed to. This feature makes it easy to follow linked lists and other complex data structures.

For more information about the Data Explorer window, see Chapter 11, “Viewing and Modifying Variables with the Data Explorer” on page 183. For more information about viewing variables, see “Viewing Variables” on page 297.

Graph View window Displays an include file dependency graph. To open the Graph View window, select Browse → Includes. For more information, see “The Graph View Window” on page 247.
Memory Test Wizard Helps you perform memory test operations. To open this window, select Target → Memory Test. For more information, see Chapter 21, “Testing Target Memory” on page 511.
Memory View window Displays memory content information in various formats. To open this window, click the Memory button (). For more information, see Chapter 15, “Using the Memory View Window” on page 323.
MULTI Editor Allows you to modify source files. To use the Editor to view and edit source code for a function, double-click the function name in the Debugger's source pane. The MULTI Editor opens on the function. (If you set another editor as your default, that editor opens on the function.) For more information about the MULTI Editor, see Chapter 4, “Editing Files with the MULTI Editor” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book.
Note Browser window Allows you to associate Debugger Notes with any line of source or assembly code. To open this window, select View → Debugger Notes. For more information, see Chapter 10, “Using Debugger Notes” on page 173.
OSA Explorer Displays information about a multitasking operating system. To open this window, click the OSA Explorer button (). For more information, see “The OSA Explorer” on page 609.
PathAnalyzer Trace-enabled targets only Displays called functions graphically. To open this window, select TimeMachine → PathAnalyzer. For more information, see “The PathAnalyzer” on page 424.

Process Viewer**Linux/Solaris only**

Lists processes on your target.

To open this window, enter the **top** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Profile window

Reports performance, function, graph, and coverage analysis profiling information.

To open this window, select **View** → **Profile**.

For more information, see “The Profile Window” on page 361.

Register Information window

Displays detailed information, including bitfields and documentation, about a specific register.

To open this window, enter the **regview** command followed by a register name. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information, see “The Register Information Window” on page 270.

Register View window

Displays current register values.

To open this window, click the **Registers** button ().

For more information, see Chapter 13, “Using the Register Explorer” on page 253.

Serial Connection Chooser window

Allows you to interact with a serial terminal.

To open this window, select **Tools** → **Serial Terminal** → **Make Serial Connection**.

For more information, see Chapter 27, “Establishing Serial Connections” on page 643.

Task Manager for *target* window**Debug servers that support multitasking debugging only**

Displays the current status of tasks running on an operating system.

To open this window, select **View** → **Task Manager**.

For more information, see “The Task Manager” on page 580.

Tree Browser

Displays information about classes, static calls, and file calls in the target program.

To open this window, select **Classes**, **Static Calls**, or **File Calls** from the **Browse** menu.

For more information, see “The Tree Browser Window” on page 239.

Viewing Information in the Command Pane

In addition to using the specialized, stand-alone windows described in the previous section to view program information, you can also view some information in the command pane of the main Debugger window, as described next.

- To view information about a variable, pointer, or structure, click its name in the source pane. When you click an item in the source pane, the resulting selection is evaluated as an expression by the **MULTI *examine*** command, which may perform macro expansion in its evaluation. For information about the **examine** command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book. See also Chapter 14, “Using Expressions, Variables, and Procedure Calls” on page 291.

For information about viewing variables, see “Viewing Variables” on page 297. When you click a pointer, the command pane also displays the value of the object pointed to. When you click a structure, the command pane displays the whole structure, with every structure or class element labeled.

- To evaluate an expression, type it into the command pane. For more information about evaluating and working with expressions, see Chapter 14, “Using Expressions, Variables, and Procedure Calls” on page 291.



Tip

You can also use Debugger commands to print a variety of program and target information to the command pane. For detailed information, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Chapter 10

Using Debugger Notes

Contents

Creating and Editing Debugger Notes	174
Organizing Debugger Notes Into Groups	177
Viewing Debugger Notes	177

This chapter describes how to create, edit, and access Debugger Notes.

Debugger Notes allow you to associate notes with any line of source or assembly code. You can quickly jump to any part of your program where you have set a Note. It is also possible to use breakpoints to mark parts of your program that are of particular interest. Debugger Notes, however, have the following advantages over breakpoints when marking locations:

- You can set Debugger Notes on source lines that do not have code associated with them, such as comments.
- You can simultaneously edit multiple Debugger Notes.
- You can organize Debugger Notes into groups.
- If you rebuild your program, Debugger Notes are not deleted—even when they are no longer in a valid location. In addition, if the line of source code where you set the Debugger Note moves, the Debugger Note relocates automatically.
- You can quickly display and navigate to any Debugger Note by pressing the **F12** key.

Creating and Editing Debugger Notes

Debugger Notes can contain any kind of text and can be associated with any line of code. The line numbers of lines that contain Debugger Notes are shaded gray.

To set a Debugger Note:

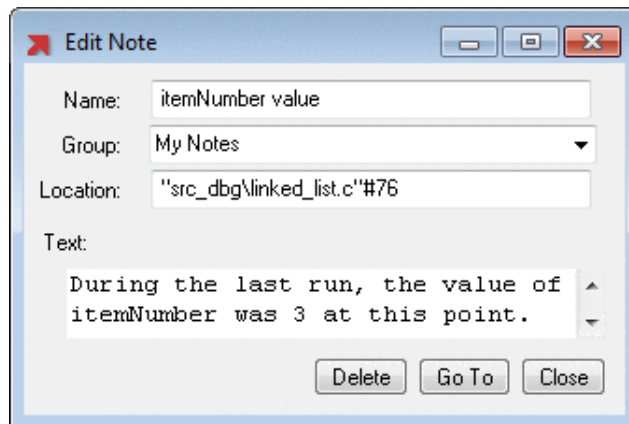
- Right-click a line in the source pane and select **Create Note**, or
- Enter the **noteedit** command in the Debugger command pane. For information about this command, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

Editing a Single Debugger Note

You can modify a single pre-existing Debugger Note with the **Edit Note** window. To open this window, do one of the following:

- Double-click the shaded gray area in the line number column.
- Right-click the shaded gray area and select **Edit Note**.

- Enter the **noteedit** command in the Debugger command pane. For more information about this command, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.



The following table explains each property you can set from this window.

Name	Assigns a name to the Debugger Note. The Note Browser and Note listings display this name. If you do not specify a name when you create a new Note, MULTI uses a brief version of the Note location as the default name.
Group	Specifies the name of the group where the Debugger Note is located. A Debugger Note must always exist in exactly one group. To put the Note in a group that already exists, select a group from the drop-down list. To create a new group, type the new group name in the Group field.
Location	Specifies the location of the Debugger Note. This may be either an address expression or a source line. For information about using an address expression to specify a location, see “Using Address Expressions in Debugger Commands” in Chapter 1, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book.
Text	Assigns text to the Note.

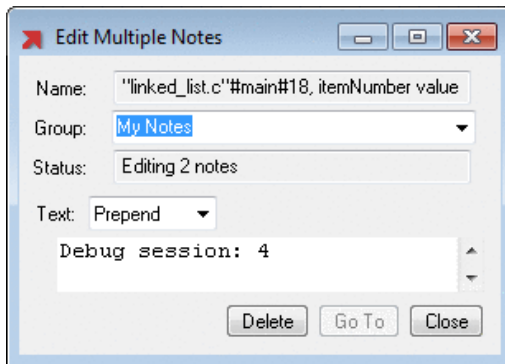
The buttons in this window allow you to perform the actions listed in the following table.

Delete	Deletes the Note displayed in the window. A dialog box asking you to confirm the deletion appears.
Go To	Navigates to the displayed Note.
Close	Closes the window, saving any changes you made to the Note. If the Note location is not valid, a warning message appears.

Editing Multiple Debugger Notes

You can select and modify multiple Notes with the **Edit Multiple Notes** window. This window provides a quick way to simultaneously edit the group or text of multiple Notes. To open this window, perform the following steps:

1. Select **View** → **Debugger Notes** or enter the **noteview** command to open the **Note Browser**. For information about the **noteview** command, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.
2. Select **List** from the **Style** drop-down box.
3. Highlight more than one Debugger Note, right-click, and select **Edit** from the menu that appears.



In the **Edit Multiple Notes** window, use the **Text** drop-down list to set whether text should be appended or prepended to the selected Notes. Enter the desired text in the field underneath the **Text** drop-down. When you click the **Close** button, the text you entered is added to the text of all the Notes listed in the **Name** field.

Removing Debugger Notes

To delete a Note, do one of the following:

- Enter the **notedel** command followed by an address expression or the number of the Debugger Note. For information about the **notedel** command, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.
- Right-click the shaded area and select **Remove Note**.

Organizing Debugger Notes Into Groups

You can organize Debugger Notes into groups to make it easier to work with a large number of them. A Debugger Note group is simply a named collection of Debugger Notes. You can view Note groups in the **Note Browser**, and you can easily delete all Notes in a group or move them to another group. For information about the **Note Browser**, see “Viewing Debugger Notes” on page 177.

Each Debugger Note you create always exists in exactly one group. If no groups exist when you create a Note, MULTI creates a new group called `<default>` to hold the Note. This group becomes the *active group*, which indicates that any Note you create is put into this group by default.

To add a Note to a group that is not the active group, you must either:

- Change the active group by clicking  in the **Note Browser**, or
- Enter the command **noteedit -group group_name**. For information about the **noteedit** command, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing Debugger Notes

To display the text stored in a Note, hover over the shaded area; the text appears in a tooltip. Alternatively, single-click the shaded line number area, and the text appears in the command pane.

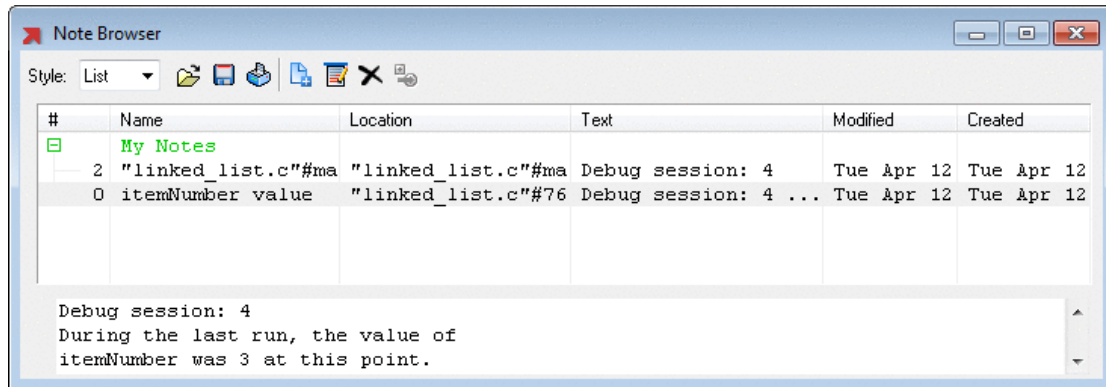
To list all your program's Debugger Notes in the command pane, enter the **notelist** command. To list all your program's Debugger Notes in the **Note Browser**, select **View → Debugger Notes** or enter the **noteview** command. For information about these commands, see Chapter 7, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

The next section contains details about the **Note Browser**.

The Note Browser

The **Note Browser** displays your Notes in a list or in full report style.

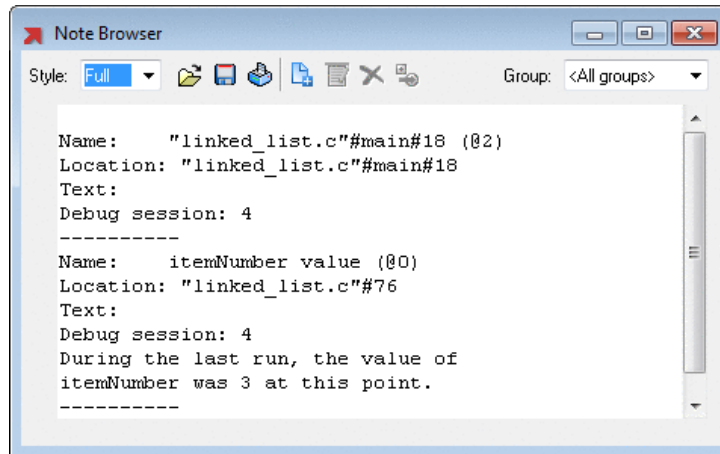
The list style display presents all Debugger Notes in a list format in which you can expand or collapse groups.



To view the list style display, select **List** from the **Style** drop-down box. To edit multiple Notes and to directly modify a group name, use the list style display. For more information, see “Editing Multiple Debugger Notes” on page 176.

In this display, **MULTI** highlights the active group in green to indicate that it adds Notes to this group by default. Clicking a Note navigates to that Note in the Debugger window.

The full report style display presents all Debugger Notes in report format. Full style display shows the Note's name, location, and text. This display allows you to search easily through Note text by using the incremental search capability (**Ctrl+F** and **Ctrl+B**). For more information, see “Incremental Searching” on page 156.



To view the full style display, select **Full** from the **Style** drop-down box.

In this display, the **Edit** button () is always disabled. The **Group** field, located in the upper-right corner of the window, determines whether the Notes from all groups are displayed or only those Notes from a particular group. The **Group** field also determines whether the **Set Active Group** button () is enabled. Choose from the **Group** drop-down list according to your preferences.

The following table describes the buttons available in the **Note Browser** window.

Button	Action
	Opens a previously saved Debugger Note list from the file you choose. For more information, see the notestate command in Chapter 7, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Saves the current list of Debugger Notes to the file you choose. For more information, see the notestate command in Chapter 7, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Prints the current view.
	Creates a new Debugger Note at the line selected in the Debugger window.
	Allows you to edit the selected Note(s) or group. This button is disabled if you select both a Note and a group. It is also disabled in full style display. For more information, see the noteedit command in Chapter 7, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Deletes the selected Notes and/or groups. For more information, see the notedel command in Chapter 7, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Action
	Sets the active group, to which Notes are added by default if you do not specify a group. This option is disabled if you select the active group.

Part III

Viewing Debugging Information and Program Details

Chapter 11

Viewing and Modifying Variables with the Data Explorer

Contents

Opening a Data Explorer	184
The Data Explorer Window	185
Viewing Multiple Items in a Data Explorer	188
Updating Data Explorer Variables	189
Freezing Data Explorer Variables	189
Types of Variable Displays in a Data Explorer	190
Changing How Variables are Displayed in a Data Explorer	195
Modifying Variables from a Data Explorer	198
Configuring Data Explorers	198
Data Explorer Messages	200
Data Explorer Menus	201

The Data Explorer is a graphical tool that displays variables of any type in a number of different formats. You can modify the values of variables from a Data Explorer.

Opening a Data Explorer

To open a Data Explorer, do one of the following:

- Double-click a variable in the Debugger source pane.
- Double-click a variable in a preexisting Data Explorer. This opens a new Data Explorer on the double-clicked variable and leaves the original Data Explorer open and unchanged.
- Enter the **view** command in the Debugger's command pane, where:
 - **view** *expr* [, *expr*]... — Opens a Data Explorer that displays the specified expression(s) *expr*.
 - **view** *type* — Opens a Data Explorer that displays the type *type*.
 - **view** **address* — Opens a Data Explorer that displays the contents of the given location in memory. You must enter an asterisk (*) before the address name.
 - **view** \$locals\$ — Opens a Data Explorer that displays all local variables. If the current procedure is a C++ instance method, the `this` pointer is displayed as well. This command is equivalent to the **localsview** command and the **Locals** button (🔍) located in the Debugger. For information about the **localsview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
 - **view** *number*:\$locals\$ — Opens a Data Explorer that displays local variables for the procedure located *number* levels up the stack. If the procedure is a C++ instance method, the `this` pointer is displayed as well.

For more information about the **view** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

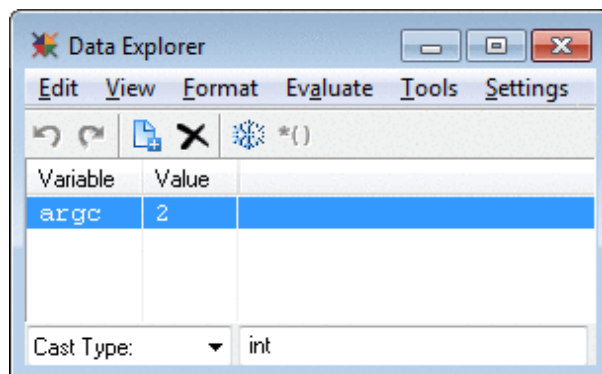
If a Data Explorer is already open when you perform any of the preceding actions, the variables you specified appear in the existing Data Explorer.

To close a Data Explorer, press **Ctrl+Q** or select **Edit → Close**. Enter the **viewdel** command to close all Data Explorers associated with the active Debugger window. This command also closes **Browse**, **Register View**, **Memory View**, **Call Stack**, and **Breakpoints** windows. For more information about the **viewdel** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The Data Explorer Window

The Data Explorer window displays information about your program variables. The format of this information depends on what type of variables you are viewing and on a number of configuration and formatting options you can modify. For information about modifying configuration options, see “Changing How Variables are Displayed in a Data Explorer” on page 195 and “Data Explorer Menus” on page 201.

A sample Data Explorer follows:



By default, the Data Explorer displays two columns. The **Variable** column displays the names of variables, types, members, or expressions in an expandable tree format. Numbers preceding variable names indicate call stack depth. For example, **2: myval** indicates that the variable **myval** is two stack levels up from the current program counter. The **Value** column displays values when applicable.



Note

For a description of all menu items available from the Data Explorer, see “Data Explorer Menus” on page 201.

The Data Explorer Toolbar

The Data Explorer toolbar contains the following buttons, from left to right:

- **Undo** () — Undoes your last action. Click **Undo** once for each action you want to undo. The **Undo** button appears dimmed when you reach the point at which no more actions can be undone.

This button is only available for certain operations. The most common operations you can undo are deleting variables from a Data Explorer, making arrays, casting variables to a different type, rerooting members, and dereferencing pointers.

- **Redo** () — Restores the last action you undid. Click **Redo** once for each action you want to restore. The **Redo** button appears dimmed when you reach the point at which no more actions can be restored.

This button is only available after you have undone certain operations. The most common operations you can redo are deleting variables from a Data Explorer, making arrays, casting variables to a different type, rerooting members, and dereferencing pointers.

- **Add Variable** () — Adds a specified variable to the Data Explorer. Specify the new variable in the dialog box that appears.
- **Delete** () — Removes the selected variable from the Data Explorer.
- **Freeze** () — Toggles the selected variable between frozen and unfrozen mode. When the variable is frozen, the Data Explorer does not automatically update it, and you cannot change its value.

Click the **Freeze** button again to reactivate the variable. When the variable is unfrozen, the Data Explorer updates it every time your process stops. For more information, see “Freezing Data Explorer Variables” on page 189.

- **Dereference Pointer** () — Displays the actual value, rather than the address, of the variable a pointer points to. This button is only available if the selected variable is a pointer.

By default, MULTI automatically dereferences pointers if the memory they point to seems safe to read (see the description of **Automatically Dereference Pointers** in “The Settings Menu” on page 208). However, even when automatic pointer dereferencing is enabled, MULTI automatically dereferences only a

single level of pointers. To manually dereference additional levels, or to dereference a pointer that MULTI does not automatically dereference, click this button. (You can also manually dereference pointers to structures by clicking the + button that appears to the left of their name.)

The Edit Bar

The edit bar allows you to change the type and value of a selected variable or define the bounds of a selected array. The edit bar is located at the bottom of the Data Explorer. The left side of the edit bar contains a drop-down menu where you can specify the action you want to perform. Descriptions of each action follow. The right side of the edit bar contains a text box (or two, depending on the action you select) where you can input new values.

- **Cast Type** — Casts the selected variable to the newly specified type. Specify a new type by entering a valid type in the text field. Press **Enter** to see your change take effect. If you enter an invalid type, your text appears red and no change occurs.

The **Undo** button () becomes available after you change a variable's type. Click the **Undo** button to return to the previous type.

The **Cast Type** entry is automatically selected when you click a variable in the **Variable** column.

- **Edit Value** — Edits the value of the selected variable. Define a new value by entering a valid number, string, or enumeration in the text field. Press **Enter** to see your change take effect. If you enter an invalid value, your text appears red and no change occurs.

The **Edit Value** entry is automatically selected when you click in the **Value** column.

- **Array Bounds** — Specifies the range of elements to display. In the text boxes, enter the indexes with which you want to begin and end the array. Press **Enter** to see your changes take effect.

The **Undo** button () becomes available after you change array bounds. Click the **Undo** button to return to the previous view.

The **Array Bounds** entry is automatically selected when you click an array in the **Variable** column.

Viewing Multiple Items in a Data Explorer

You can use the Data Explorer to view multiple variables or to view the same variables at different points during a process.

When you double-click a variable in the source pane or enter the **view** command followed by a variable, MULTI opens a Data Explorer displaying information about that variable. Information about each additional variable you specify—either by double-clicking in the source pane or by entering the **view** command—is added to the existing Data Explorer. For more information about the **view** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

If you double-click a variable in the source pane more than once or if you enter the **view** command to specify the same variable more than once, the variable appears in the same Data Explorer multiple times. You can freeze one instance of the variable and leave the other unfrozen to display the same variable at different points in the process. For information about freezing variables, see “Freezing Data Explorer Variables” on page 189.

To open a new Data Explorer on a variable listed in an existing Data Explorer, double-click the item in the Data Explorer. The original Data Explorer remains open and unchanged, and a new Data Explorer opens on the double-clicked item. The undo/redo history for the original Data Explorer is not transferred; the new Data Explorer starts with a new history.

You can drag variables from the Debugger's source pane or from another Data Explorer into an open Data Explorer. If you drag a variable from the source pane, MULTI copies it into the Data Explorer. If you drag a variable from another Data Explorer, MULTI moves the variable into the specified Data Explorer and removes it from its original location.

Updating Data Explorer Variables

The Data Explorer updates the values of its non-frozen variables each time your process stops. You can also force an update of non-frozen variables. In the Debugger command pane, enter the **update** command with no arguments. If necessary, the Data Explorer halts the process to update the data and then resumes the process.

You can also enter the **update interval** command to schedule periodic updates. By updating Data Explorer variables every *interval* seconds while your process is running, this command allows you to monitor the value of variables continually. To deactivate this update, re-enter the **update interval** command with the interval set to zero (0).



Note

The **update** command also attempts to refresh other view windows including the **Register View**, **Memory View**, and **Call Stack** windows. For more information about the **update** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Freezing Data Explorer Variables

You can freeze Data Explorer variables to compare the values of variables at different times. If you double-click a variable in the source pane more than once or if you enter the **view** command to specify the same variable more than once, the variable appears in the same Data Explorer multiple times (see “Viewing Multiple Items in a Data Explorer” on page 188). You can freeze one instance of the variable and leave the other unfrozen. If you freeze a child element (such as a member of a structure) in a Data Explorer, all children under the same parent will also be frozen.

To freeze a variable, do one of the following:

- Select the variable and click the **Freeze** button (.
- Select the variable and then select **View** → **Freeze “variable”**.

While the frozen variable is selected, the **Freeze** button () appears to be pushed down and a tick mark appears beside the **Freeze “variable”** menu item. The text for frozen variables is blue. You cannot update or edit frozen variables.

To reactivate a frozen variable, click the **Freeze** button again or select **View** → **Freeze “variable”** again. The Data Explorer updates the value of the variable to reflect the current state of the process and continues to automatically update it each time the process stops.



Tip

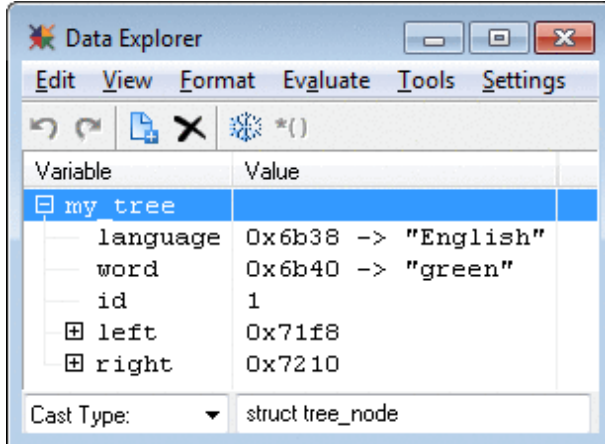
To freeze an original variable in an existing Data Explorer and open the same variable in another Data Explorer, select the variable and press **Ctrl+D**.

Types of Variable Displays in a Data Explorer

The following sections describe example Data Explorers for structures, linked lists, arrays, and C++ classes.

Displaying Structures

The following graphic displays a Data Explorer showing a structure named `my_tree`. It was generated with the `view my_tree` command.



In this example, the name of the displayed variable is `my_tree`, and its type is `struct tree_node`. This structure contains five fields, which each appear on separate lines: `language`, `word`, `id`, `left`, and `right`. These fields are in **Natural** format. The Data Explorer dereferences pointers to simple items and shows the value of the item pointed to.



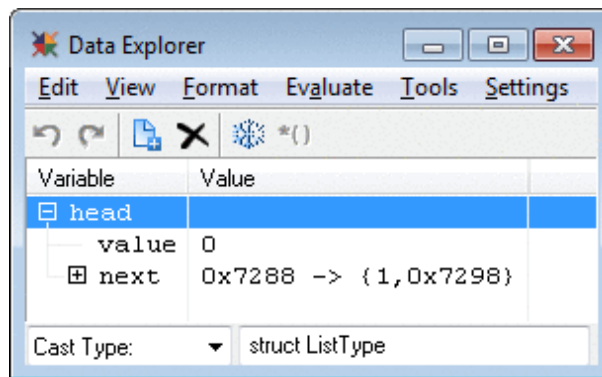
Note

In C++, the Data Explorer marks member variables stored by reference with an “@” preceding the address.

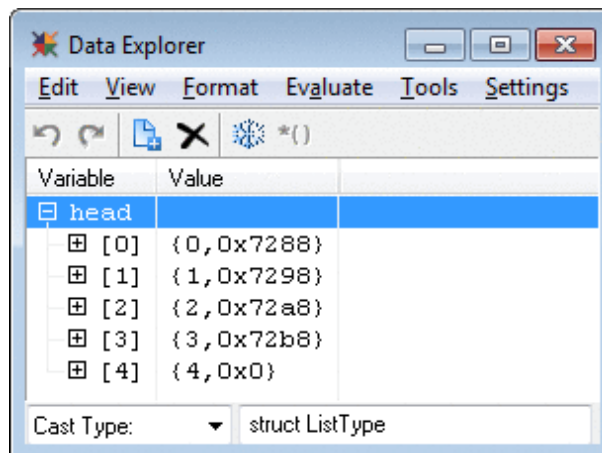
Displaying Linked Lists

The Data Explorer displays linked lists in two different formats. The graphic below represents a typical Data Explorer view for the following structure definition:

```
struct ListType {
    int value;
    struct ListType *next;
}
```



The second format—formatting a linked list as a container of elements—displays the list as if it were an array. As the following graphic demonstrates, this format greatly simplifies the display of these common structures.



To format a linked list as a container of elements, select a member or variable of the type you want to display, and select **View** → **View “variable” as Container**. This menu item is only available if you select a pointer to an aggregate type.

The **Expand As Container** dialog box opens. This dialog box gives you the following choices for how to display your data type:

- **Null-terminated list** — Displays the type as a C-style null-terminated list. A null-terminated list is a linked list that consists of a series of structures connected via *next* pointers and terminated with a NULL pointer. Choose the member that is the *next* pointer in the “**Next**” **Pointer** drop-down list, and click **OK**.
- **Circular list** — Displays the type as a C-style circular list. A C-style circular list is a linked list that consists of a series of structures connected via *next* pointers that form a loop. For the purpose of display, MULTI terminates the list when iteration returns to the initial node. Choose the member that is the *next* pointer in the “**Next**” **Pointer** drop-down list, and click **OK**.
- **Binary tree** — Displays the type as a tree of objects with two children, *left* and *right*. Choose the *left* and *right* pointers from the appropriate drop-down lists, and click **OK**. MULTI traverses the tree in order.

After you specify a visualization for the data type, MULTI displays all variables of that type as a container of elements. To display more element rows, select **View** → **Show More Elements**. To display fewer rows, select **View** → **Show Fewer Elements**. To display all element rows, select **View** → **Show All Elements**.

To stop viewing the type as a container, select a variable of that type in a Data Explorer, and then select **View** → **Stop Viewing as Container**. You may also temporarily disable the capability of viewing the type as a container. To do so, select a variable of that type in the Data Explorer and then select **Format** → **Format Container**, clearing the **Format Container** option.



Note

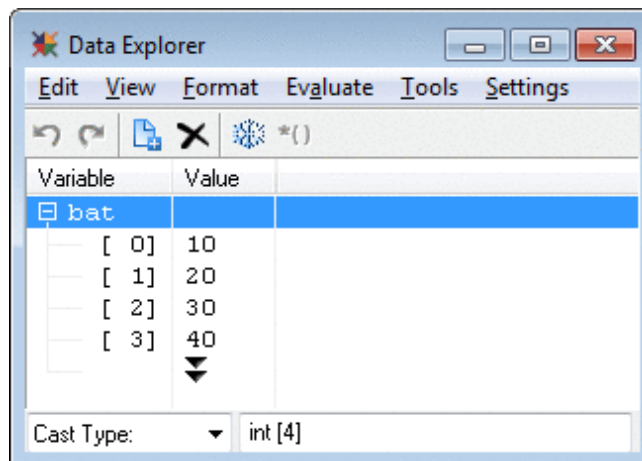
In addition to lists, you can also format other containers or structures as containers of elements. For more information, see Appendix E, “Creating Custom Data Visualizations” on page 741.

You can also use the **viewlist** command to display the elements in a linked list structure. Entering this command is helpful if you are trying to view a few elements in the middle of the linked list. However, the display this command creates often

takes up more space than the container display and can be redundant unless you select **View → Stop Viewing as Container** first. In addition, you must specify exactly how many elements you want to view, whereas you can simply select the **Show More Elements**, **Show Fewer Elements**, and **Show All Elements** menu items in the container display. For more information about the **viewlist** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Displaying Arrays

The following graphic displays an array in a Data Explorer. It was generated with the `view bat` command, where `bat` is an array of 4 integers.



The Data Explorer shows each element of an array on a separate line. The indices appear in the **Variable** column and the values of each element appear in the **Value** column.

To display a pointer or address type as an array, do one of the following:

- Select a variable in the Data Explorer and then select **Format → Make Array**. Each subsequent time you select **Format → Make Array**, the size of the array increases by ten (10).
- Select a variable in the Data Explorer and then select **Cast Type** from the edit bar. (If you select the variable from the **Variable** column, **Cast Type** is automatically selected.) Enter any valid type followed immediately by a number enclosed in square brackets.

For example, you might enter:

```
type[number]
```

where *type* is a valid type and *number* is any integer.

Press **Enter** to see your change take effect.

To view a C or C++ character pointer as an array, select the pointer, ensure that **Settings** → **Automatically Dereference Pointers** is enabled (the default), and then select **Format** → **View Alternate**. You may have to click the plus icon to see individual elements.

To change the size of a selected array, select one of the following:

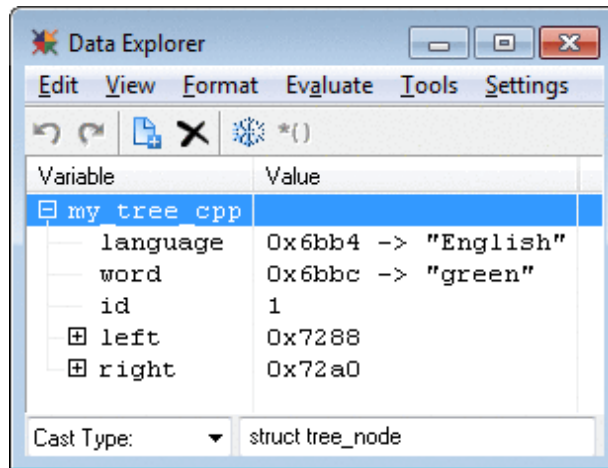
- **View** → **Show More Elements**
- **View** → **Show Fewer Elements**
- **View** → **Show All Elements**

You can also change the size of a selected array by editing the number that appears in square brackets when **Cast Type** is selected in the edit bar.

To change the bounds of a selected array, choose **Array Bounds** from the edit bar. (If you selected the array from the **Variable** column, **Array Bounds** is automatically selected.) Then enter the indices with which you want to begin and end the array. Press **Enter** to see your change take effect.

Displaying C++ Classes

The following graphic displays a Data Explorer showing C++ classes.



MULTI can display C++ class types, base class types, and virtual base class types in a Data Explorer. The Data Explorer displays static members of a class inside square brackets. It displays members of an anonymous union in an unnamed tree.

Changing How Variables are Displayed in a Data Explorer

You can modify how Data Explorers display variables by doing one of the following:

- Change the type used to display a variable. See “Changing the Type Used to Display a Variable” on page 195.
- View pointers to C++ base classes. See “Viewing Pointers to C++ Base Classes” on page 196.
- Change the base used to display a number. See the descriptions of the **Hex**, **Natural**, **Decimal**, **Binary**, and **Octal** menu items in “The Format Menu” on page 204.

Changing the Type Used to Display a Variable

To change the type used to display a variable in the Data Explorer, select the variable and then select **Cast Type** from the edit bar. (If you select the variable from the **Variable** column, **Cast Type** is automatically selected.) Specify the new type by

entering a valid type in the text field. Press **Enter** to see your variable cast to the newly specified type. Each change you make to the type in this way is independent of previous changes. If you enter an invalid type, your text appears red and no change occurs. You cannot change the type of a frozen variable.

To change a variable's type to an array of the current type, see “Displaying Arrays” on page 193.

Viewing Pointers to C++ Base Classes

In C++, a Data Explorer can cast a base class pointer to its derived class type. To enable this behavior, select the pointer and then select **Format** → **Cast To Derived**. With this option enabled, MULTI attempts to find the actual type of an object by matching the name of its virtual table to a known type.

An example follows.

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    virtual int af() {return a;}
};

class B : public A {
public:
    int b;
};

int main(int argc, char *argv[])
{
    A *ap;
    B b;

    ap = (A*)&b;
    ap->a = 10;

    cout << "ap->a: " << ap->a << endl;

    return 0;
}
```

If, in the preceding example, you open a Data Explorer on the variable `ap` after the statement `ap = (A*)&b;`, MULTI resolves the variable to a pointer to the `B` class. It does so by matching the virtual function table for `B` with the one from `ap`. To

recast the variable as a pointer to the base class (A), disable **Format** → **Cast To Derived**.

There are limitations to this capability. The base class must have a virtual function table or no comparison can be made. For example, if class A were defined as follows:

```
class A {
    public:
        int a;
};
```

and then you enable **Format** → **Cast To Derived**, the Data Explorer view does not change because MULTI cannot resolve the derived class. This limitation also applies to multiple inheritance. An example follows.

```
#include <iostream>
using namespace std;
class A {
    public:
        int a;
        virtual int af() {return a;}
};

class Z {
    public:
        int z;
};

class M : public A, public Z {
    public:
        int m;
};

int main(int argc, char *argv[])
{
    A *ap;
    Z *zp;
    M m;

    m.a = 10;

    ap = (A*)&m;
    zp = (Z*)&m;

    cout << "ap->a: " << ap->a << endl;
    cout << "zp->z: " << zp->z << endl;

    return 0;
}
```

In the preceding example, MULTI can determine that the variable `ap` points to an instance of class `M`. However, it cannot determine the actual type that the variable `zp` points to because class `Z` has no virtual function table.

Modifying Variables from a Data Explorer

To modify a variable's value, select the variable and then select **Edit Value** from the edit bar. (If you select the variable's value in the **Value** column, **Edit Value** is automatically selected.) Define the new value by entering a valid number, string, or enumeration in the text field. Press **Enter** to see your change take effect. If you enter an invalid value, your text appears red and no change occurs.

Configuring Data Explorers

You can configure settings for both individual Data Explorers and for all Data Explorers. The following sections describe how to do so.

Configuring Individual Data Explorer Dimensions

When you first open a Data Explorer, MULTI auto-sizes the window to a size appropriate for the displayed data, but within the minimum and maximum height and width values specified in your current configuration file. MULTI also automatically positions the column divider in the Data Explorer so that the **Variable** column fully displays the longest variable name, type name, member name, or expression.

You can manually resize a Data Explorer and/or column by dragging the window edges or the column divider. However, if you manually resize the window, the Data Explorer no longer auto-sizes to adjust for new data. If you want to use a Data Explorer that automatically resizes, you must close the current Data Explorer and open a new one.

You can also specify global Data Explorer dimensions in the **Options** window. For more information, see “Configuring Global Data Explorer Dimensions” on page 200.

Setting Global Options for Data Explorers

In addition to formatting individual Data Explorers, you can also set a number of MULTI IDE configuration options that affect the format and behavior of all variables in all Data Explorers.

Limiting the Complexity of Data Displayed

Depending on your target, you may want to minimize the amount of data MULTI reads from the target. MULTI provides configuration options to limit the complexity of information displayed in Data Explorers and to control how much data is read. After making configuration changes, issue the **update** command to see the effects of your changes. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The following list provides the configuration options.

- To specify a maximum length for the string representation of data in Data Explorers, enter the **configure formatStringMaxLength *num*** command in the Debugger command pane, where *num* is the maximum number of characters. When the accumulated data reaches this length, MULTI does not read any further data from the target unless you select a more detailed view of that data. The minimum value is 1024 bytes.
- To specify the maximum number of nested structure levels that Data Explorers display on one line, enter the **configure formatStringMaxDepth *num*** command in the Debugger command pane, where *num* is the maximum depth of the display. Past this maximum depth, the Data Explorer displays values of nested structures as The minimum value is one level.
- To set the maximum initial number of elements in a container, enter the **configure maxContainerDisplaySize *num*** command in the Debugger command pane, where *num* is the maximum number of elements. The default setting is 20.
- To set the number of elements added to a display when you expand a container, enter the **configure containerSizeincrement *num*** command in the Debugger command pane, where *num* is the number of elements to be added. The default setting is 10.

For more information about **formatStringMaxLength** and **formatStringMaxDepth**, see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book. For more information about **maxContainerDisplaySize** and **containerSizeIncrement**, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Configuring Global Data Explorer Dimensions

The global configuration options **Minimum initial size (WxH)** and **Maximum initial size (WxH)** affect the default dimensions of Data Explorers, while **Initial position (XxY)** affects the default positioning. To access these configuration options, select **Config** → **Options** → **Debugger** tab.

For more information, see “The Debugger Options Tab” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Data Explorer Messages

The Data Explorer may display any of the following messages.

Message	Meaning
...	There is too much data to show on this line. To expand the data in the current window, click the plus sign (+) that appears to the left of the variable. To show the data in a new Data Explorer, double-click the line.
Dead	The variable no longer exists in the current lexical scope. This usually happens when you have stepped past the last use of the variable. If the variable is assigned to a register, the Data Explorer shows the current value of the register along with this message, but the value is most likely meaningless.
Empty	The container has no elements to display.
NaN	Short for “not a number.” The value is not a legal representation of any number. This message applies to floating-point variable types.
No process	No process is currently being debugged, so the Data Explorer cannot display a value.

Message	Meaning
No symbols for this procedure	Debug symbol information does not exist for the current procedure. This message applies to local variables.
Optimized away	The variable does not exist because a compiler optimization removed it.
Original procedure not on stack	The original procedure, in which the variable was in scope, is no longer on the call stack. The Data Explorer only displays this message when Evaluate → In Context is enabled. For information about Evaluate → In Context , see “The Evaluate Menu” on page 206.
Process running	The process you are debugging is running, so the current value of the variable is unknown.
Not Initialized	The variable has not been assigned an initial value and could have a random value in memory. The Data Explorer shows the current value of the variable, but this value is most likely meaningless.
Unreadable memory	MULTI does not have read access to the memory location that a pointer or address type points to, or the memory location does not exist.
Unreadable/Unknown	TimeMachine was unable to reconstruct the value of the variable. For more information, see “Dealing with Incomplete Trace Data” on page 411.

Data Explorer Menus

The following sections describe all menu items available from the Data Explorer. Some menu items are context sensitive and are therefore unavailable in certain situations. Some of these menu items are also available via the Data Explorer's right-click menu.

If a toggle menu item is enabled, a check mark (Windows) or a dot (Linux/Solaris) appears to the left of the menu item. Unless otherwise stated, all descriptions of toggle menu items explain the behavior of the item when enabled.

The Edit Menu

The following table describes the menu items available from the Data Explorer's **Edit** menu.

Menu Item	Meaning
Add Variable	Adds a variable to the Data Explorer.
Remove “variable” from Window	Removes the selected variable from the Data Explorer.
Move “variable” to New Window	Moves the selected variable from its original location in an existing Data Explorer to a new Data Explorer.
Copy “variable” to New Window	Copies the selected variable to a new Data Explorer.
Close	Closes the Data Explorer. Equivalent to Ctrl+Q .

The View Menu

The following table describes the menu items available from the Data Explorer's **View** menu.

Menu Item	Meaning
Show	Opens a submenu that lists the following menu items: • Address (Hotkey: S) — Displays the Address column, which shows the addresses of the variables you are viewing. If you are viewing a structure, the field offsets also appear. By default, this item is disabled. • Type (Hotkey: T) — Displays the Type column, which shows the types of the variables you are viewing. By default, this item is disabled. • Variable — Displays the Variable column, which shows the names of the variables you are viewing. By default, this item is enabled. • Value — Displays the Value column, which shows the values of the variables you are viewing. By default, this item is enabled.
Toolbar	Shows the toolbar. By default, this item is enabled.

Menu Item	Meaning
Freeze “<i>variable</i>”	Freezes the selected variable. When the variable is frozen, the Data Explorer does not automatically update it, and you cannot change its display format or value. When the variable is unfrozen, the Data Explorer updates it every time your process stops. By default, this item is disabled. For more information, see “Freezing Data Explorer Variables” on page 189.
Dereference Pointer	Displays the actual value, rather than the address, of the variable a pointer points to. This option is only available if the selected variable is a pointer. By default, MULTI automatically dereferences pointers if the memory they point to seems safe to read (see the description of Automatically Dereference Pointers in “The Settings Menu” on page 208). However, even when automatic pointer dereferencing is enabled, MULTI automatically dereferences only a single level of pointers. To manually dereference additional levels, or to dereference a pointer that MULTI does not automatically dereference, select this option. (You can also manually dereference pointers to structures by clicking the + button that appears to the left of their name.) Equivalent to the hotkey P.
Show Register Info for “<i>variable</i>”	Opens the Register Information window. For more information, see “The Register Information Window” on page 270. This option is only available for local variables stored in a register.
View “<i>variable</i>” as Register	Displays <i>variable</i> as a register, or opens the Register Setup dialog box if <i>variable</i> does not match an existing register definition.
View “<i>variable</i>” as Container	Opens the Expand As Container dialog, which allows you to specify how to display variables of <i>variable's</i> type as containers. This option is only available for aggregate types or pointers to aggregate types.
Stop Viewing as Container	Stops viewing variables of <i>variable's</i> type as containers. This option only applies to containers previously displayed with View “<i>variable</i>” as Container (preceding).
Show More Elements	Displays 10 more rows of information in the Data Explorer. This menu item is only available if you select the name or address of an array, list, or container. Note: This option can be used to extend an array past its end. This causes target memory to be read past the end of the array and changes the size of the array for the purposes of the Data Explorer.
Show Fewer Elements	Displays 10 fewer rows of information in the Data Explorer. This menu item is only available if you select the name or address of an array, list, or container.

Menu Item	Meaning
Show All Elements	Displays rows of information for all elements in the Data Explorer. This menu item is only available if you select the name or address of a statically sized array, list, or container (but not a C pointer).

The Format Menu

The following table describes the menu items available from the Data Explorer's **Format** menu.



Note

The first five menu items, which control how numbers are displayed in a Data Explorer, pertain to all types except character pointer types. These string types are always displayed as quoted strings unless the menu items **View Alternate** and **Settings → Automatically Dereference Pointers** are enabled, in which case they are displayed as an array of characters.

Menu Item	Meaning
Hex	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 16. By default, this item is disabled. However, you can make it the default format for numbers and characters by selecting Config → Options → Debugger tab and checking Display all numbers/characters as hex. See also the note that precedes this table. Equivalent to the hotkey H.
Natural	Displays addresses of the selected variable, the selected variable's children, etc. in hexadecimal notation, characters in ASCII notation, and all other numbers in decimal notation. By default, this item is enabled. See also the note that precedes this table. Equivalent to the hotkey N.
Decimal	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 10. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey D.

Menu Item	Meaning
Binary	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 2. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey B.
Octal	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 8. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey O.
Pad Hex Values	Inserts zeros to the left of hexadecimal values. When you enable this item, the hexadecimal value has the same bit width as the displayed data. By default, this item is disabled. Equivalent to the hotkey X.
Make Array	Displays variables as arrays. For pointer and address types, this treats the pointer as an array such that the first element of the array is located at the address pointed to. For non-pointer types, the Data Explorer displays an array of the appropriate type starting at the variable's location. Each subsequent time you select Make Array, the size of the array increases by ten (10). C and C++ character pointer types must be in View Alternate mode to be viewed as arrays. Additionally, Settings → Automatically Dereference Pointers must be enabled (the default). See View Alternate (following). Equivalent to the hotkey A.
View Alternate	Displays data in an alternate way. For most types, the value is displayed in the currently selected format as well as in the alternate format. For example, an integer in natural format is displayed as <i>decimal (hexadecimal)</i>. If Settings → Automatically Dereference Pointers is enabled (the default), a character pointer, which is normally displayed as a string, is displayed as an array. All other pointers are treated as integers, which default to displaying in hexadecimal. By default, this item is disabled. Equivalent to the hotkey V.

Menu Item	Meaning
Cast to Derived	Determines the derived C++ class type of the current object and displays the object cast to that type. This item only applies to the display of C++ classes. For more information, see “Viewing Pointers to C++ Base Classes” on page 196. By default, this item is enabled. Equivalent to the hotkey I .
Show Member Functions	Displays class member functions in type view. By default, this item is enabled. Equivalent to the hotkey F .
Show Template Parameters	Displays the typedefs for template parameters. This option is only valid when you are viewing an instance of a type with template parameters in C++. By default, this item is disabled. Equivalent to the hotkey W .
Format Container	Displays Standard Template Library (STL) containers as if they were an array of elements. See “Displaying Linked Lists” on page 191. You can also define data descriptions for custom data types. See Appendix E, “Creating Custom Data Visualizations” on page 741. By default, this item is enabled. Equivalent to the hotkey E .

The Evaluate Menu

The **Evaluate** menu contains four menu items that control the context that is searched to locate the variables being displayed. These options are mutually exclusive; only one of them can be selected at a time. By default, **As Global** is used for all expressions involving only global variables, **By Address** is used if the expression is a static variable, and **In Context** is used for most others. MULTI detects which category the viewed variable falls under, and sets these options accordingly. You can change these options at any time, causing MULTI to re-evaluate the variable with the new setting.

The following table describes the menu items available from the Data Explorer's **Evaluate** menu.

Menu Item	Meaning
In Context	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression in the same context in which it first evaluated it. For example, if other procedures have been called since the Data Explorer was created, MULTI walks up the stack until it finds a stack frame with the procedure that was executing when the Data Explorer was created. It then evaluates the expression there. If MULTI cannot find such a stack frame, the Data Explorer displays an error. See also As Local (following). Equivalent to the hotkey C .
As Local	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression within the current procedure at the top of the call stack. See also In Context (preceding). Equivalent to the hotkey L .
As Global	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression, looking for variables in the global scope and ignoring all procedure scopes. This option is useful when an expression involves only global variables. Equivalent to the hotkey G .
By Address	Specifies that when the Data Explorer is updated, MULTI will use the last valid variable address to display data. MULTI does not re-evaluate the selected variable's expression in any context. This option is useful for examining the contents of local variables in memory before and after they are in scope. Equivalent to the hotkey R .

The Tools Menu

The following table describes the menu items available from the Data Explorer's **Tools** menu.

Menu Item	Meaning
Go to Definition of “variable”	Displays the source code, if available, for <i>variable's</i> definition in the source pane.
Go to Declaration of “variable”	Displays the source code, if available, for <i>variable's</i> declaration in the source pane. This is available only for global and static variables.

Menu Item	Meaning
Browse References of “variable”	Displays <i>variable</i> 's cross references, if any, in a Browse window. For more information, see “Browsing Cross References” on page 236.
Set Watchpoint on “variable”	Sets a watchpoint on <i>variable</i> . For more information, see the watchpoint command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Explore “variable”	Opens a Graph View window on <i>variable</i> . If data descriptions are available for <i>variable</i> 's type, MULTI traverses the data and displays a graph according to the data description. See Appendix E, “Creating Custom Data Visualizations” on page 741.
Memory View on “variable”	Opens a Memory View window in which you can view and modify memory contents. Initially, this window displays memory at the address of the selected variable. See also the memview command in “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Equivalent to the hotkey M .
Print	Opens a dialog box that allows you to print the contents of the Data Explorer.

The Settings Menu

The following table describes the menu items available from the Data Explorer's **Settings** menu.

Menu Item	Meaning
Automatically Dereference Pointers	Dereferences pointers viewed in the Data Explorer if the memory they point to seems safe to read. For character pointers, this option reads the string pointed to and displays it in the Data Explorer. Except for character pointer variables, the names of dereferenced variables begin with an asterisk (*). By default, this item is enabled. Equivalent to the configuration option derefPointer . For information about derefPointer , see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book.
Color Text Based on Evaluate Context	Applies different colors to local variables, global variables, variables evaluated in context, and variables evaluated by address. By default, this item is enabled.

Menu Item	Meaning
Default Large Variables to New Window	Opens a new Data Explorer for variables that have more than five rows of information. By default, this item is disabled.

Chapter 12

Browsing Program Elements

Contents

Browsing Program Elements Overview	212
The Browse Window	214
The Tree Browser Window	239
The Graph View Window	247

The MULTI Debugger provides several kinds of display windows that allow you to view program elements in different graphical formats. This chapter describes each of the following windows:

- The Browse window — Lists information about the selected object type in a single collapsible and expandable list. For general information about this window, see “The Browse Window” on page 214.
- The Tree Browser window — Displays information about the selected object type in collapsible and expandable lists. This window displays expanded information in new columns, thus offering a hierarchical view of the relationship among the items it displays. For general information about this window, see “The Tree Browser Window” on page 239.
- The Graph View window — Displays an object graph that shows the relationships between items. For general information about this window, see “The Graph View Window” on page 247.

Browsing Program Elements Overview

You can browse the following program elements in one or more of these windows:

- *Procedures* — Select **Browse** → **Procedures**. See “Browsing Procedures” on page 220.
- *Global variables* — Select **Browse** → **Globals**. See “Browsing Global Variables” on page 228.
- *Source files* — See “Browsing Source Files” on page 230.
 - *All* (all source files that contain procedures used in the program you are debugging) — Select **Browse** → **Files**.
 - *Includers* (all source files that directly include the current file) — Right-click an empty spot in the source pane and select **Browse Includers Of This File** from the menu that appears.
 - *Included files* (all files that the current file directly includes) — Right-click an empty spot in the source pane and select **Browse Files Included By This File** from the menu that appears.
 - *Include graph* (a graph of included files) — Select **Browse** → **Includes**. See “Browsing Includes” on page 247.

- *Data types* — Select **Browse** → **All Types**. See “Browsing Data Types” on page 233.
- *Cross references* (for a particular symbol in the program you are debugging) — In the source pane, right-click the symbol and select **Browse References** from the menu that appears. See “Browsing Cross References” on page 236.

Note that this menu option is only available if the program you are debugging was compiled with cross reference information. For information about how to do this, see the *MULTI: Building Applications* book for your target.

- *Class hierarchies* — Select **Browse** → **Classes**. See “Browsing Classes” on page 243.
- *Calls*:
 - *Static calls by function* (functions that call other functions, and functions that the procedure at the current line pointer calls) — Select **Browse** → **Static Calls**. See “Browsing Static Calls By Function” on page 244.
 - *Dynamic calls by function* (functions that were actually called during run time) — Select **Browse** → **Dynamic Calls**. See “Browsing Dynamic Calls by Function” on page 246.

Note that this menu option is only available if you collected profiling data about function calls (that is, call count data with call graph support enabled). See “Overview of Profiling Methods” on page 355.

- *Static calls by file* (source files whose functions are called from a particular source file) — Select **Browse** → **File Calls**. See “Browsing Static Calls By File” on page 245.



Note

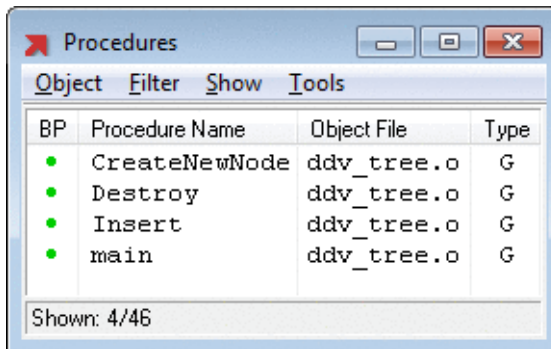
For most of these program elements and views, there are alternative ways to open the browsing tools. The following sections document each browsing tool, its views, and the ways you can access it.

The Browse Window

The Browse window is a graphical tool that you can use to view information about procedures, global variables, source files, data types, and cross references. The columns in the main portion of the window vary depending on what type of object you are viewing, but the four menus and the basic behavior of the window are always the same. The following section describes the aspects of the Browse window that remain constant. Later sections document how you can use the Browse window to view specific types of information.

Browse Window Basics

You can open a Browse window in multiple ways—each of which is described in more detail in the following sections. For most of the elements that you can view in a Browse window, you can simply select **Browse** → *program element* from the Debugger. For example, to open a Browse window similar to the one shown below, select **Browse** → **Procedures**.



Note

To browse cross references, files that include the current file, or files that the current file includes, right-click in the source pane and select the appropriate menu option from the menu that appears. For more information, see “Browsing Source Files” on page 230 and “Browsing Cross References” on page 236.

As with most MULTI windows, you can:

- Reorder the columns of a Browse window by dragging and dropping column headers.

- Sort the displayed data by clicking the relevant column header.
- Open a context-sensitive shortcut menu by right-clicking in the window.

Every Browse window includes one column known as the *primary column*. Unlike other columns, you can never hide the primary column, but you can change the content formatting via the **Show** menu. The primary column and available formatting options differ based on what type of object you view.

Every Browse window includes the following four menus. Menu options vary according to the type of information you browse.

- **Object** menu — Allows you to switch among viewing procedures, global variables, source files, or data types. The type of program element you are currently viewing has a bullet or check mark next to it. Simply select one of the other choices listed at the top of the **Object** menu to change the type of object being displayed. In a single Browse window, you can only view one type of information at a time, but you can have multiple Browse windows open simultaneously.

This menu also allows you to print the contents of the current Browse window, access MULTI's online help information, or close the Browse window. This menu is exactly the same in all Browse windows.

- **Filter** menu — Allows you to specify filters that limit which objects the Browse window displays. You can set a user-defined filter and/or a selection of predefined filters. The availability of predefined filters varies depending on what type of object you are viewing. For more information about filtering, see “Filtering Content in the Browse Window” on page 216.
- **Show** menu — Allows you to specify the columns displayed in the Browse window and the style of the primary column. The availability of columns and styles varies depending on the type of object you are viewing. Menu options are described in the sections that cover each type of Browse window.
- **Tools** menu — Allows you to perform actions on the information in the Browse window or open other tools that allow you to do so. Most of the options in this menu also appear in the shortcut menu. The availability of menu items in this menu and in the shortcut menu vary depending on what type of object you are viewing. Menu options are described in the sections that cover each type of Browse window.

Each Browse window contains a status bar, which lists the number of items displayed in the format:

Shown: *visible/total*

where:

- *visible* is the number of items that are not filtered. See “Filtering Content in the Browse Window” on page 216.
- *total* (also known as the *base set*) is the number of total objects encapsulated by the Browse window. See “Using Filters” on page 219.

If the Browse window contains contracted headings (see “Browse Window Headings” on page 220), the status bar displays the following alternative format:

Shown: *visible/total (expanded/visible Expanded)*

where:

- *visible* and *total* are the same as above.
- *expanded* is the number of entries that are both visible (i.e., not filtered) and not contracted.

Filtering Content in the Browse Window

In all Browse windows, you can select what data to display by using pre-existing and/or user-defined filters. You can enable and disable filters using the **Filter** menu. Some filters are enabled by default when procedures or global variables are first displayed in a Browse window.

The **Filter** menu displays only those filters that apply to the current type of object. User-defined filters are applied to the names of the objects being shown in the Browse window (see “User-Defined Filters” on page 218). The status box at the bottom of the Browse window lists the number of displayed objects as well as the total number of objects, which includes those that have been filtered out.

Predefined Filters

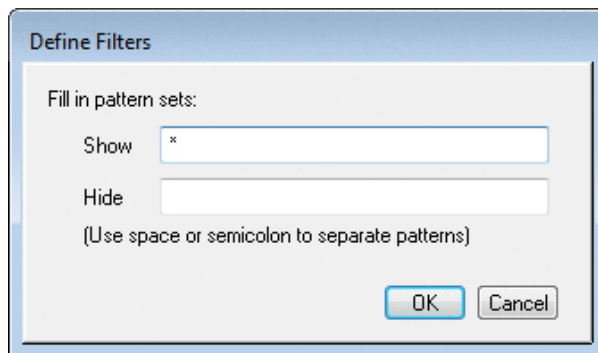
The predefined filters you can enable and disable from the **Filter** menu are described in the following table. Only those filters appropriate to the object type you are viewing are available in the menu. You can apply multiple filters.

Menu option	Effect
Hide C++ VTBLs	Toggles the display of virtual tables in C++ programs. This filter can only be applied to global variables.
Hide C++ Type Identifiers	Toggles the display of type identifiers in C++ programs. This filter can only be applied to global variables.
Hide C++ Type Info	Toggles the display of type information in C++ programs. This filter can only be applied to global variables.
Hide C++ Initialization Names	Toggles the display of initialization names in C++ programs. This filter can only be applied to global variables.
Hide C++ std::*	Toggles the display of objects in the <code>std</code> namespace in C++ programs. This filter can be applied to global variables, procedures, or data types.
Hide .*	Toggles the display of objects whose names begin with a period (.). This filter can be applied to global variables, procedures, or data types.
Hide _*	Toggles the display of objects whose names begin with an underscore (_). This filter can be applied to global variables, procedures, data types, or source files.
Hide Globals from Shared Library	Toggles the display of global variables from shared libraries that were loaded into your program. This filter can only be applied to global variables.
Hide Files without Procedures	Toggles the display of source files that do not contain any procedures. This filter can only be applied to files.
Hide Procedures without Source	Toggles the display of procedures that do not have source code. This filter can only be applied to procedures.
Hide Inlined Procedures	Toggles the display of procedures that are inlined. This filter can only be applied to procedures.
Hide Static Names	Toggles the display of objects that are defined as static. This filter can be applied to either global variables or procedures.
Hide Non-Static Names	Toggles the display of objects that are not defined as static. This filter can be applied to either global variables or procedures.

Menu option	Effect
Hide Writes	Toggles the display of writes. This filter can only be applied to cross references.
Hide Reads	Toggles the display of reads. This filter can only be applied to cross references.
Hide Addresses	Toggles the display of address references. This filter can only be applied to cross references.
Hide Declarations	Toggles the display of declarations. This filter can only be applied to cross references.

User-Defined Filters

To define your own filter to apply to the objects listed in the Browse window, select **Filter** → **User-defined Filter** from the menu bar of any Browse window. The following **Define Filters** dialog box opens:



To specify what objects to display in the Browse window, enter text and wildcard patterns in this dialog box (see “Wildcards” on page 303). To specify multiple patterns in the **Show** and **Hide** text fields, use semicolons or whitespace to separate the patterns.

The next section explains how the Browse window processes user-defined filters with selected predefined filters.

Using Filters

If you specify a user-defined filter and/or one or more predefined filters, the Browse window uses the following algorithm to determine what objects to display.

1. The Browse window determines the base set of objects in the Browse window. When you first open a Browse window, the base set of objects is the set of objects that were initially specified. For example, if you open the Browse window with the **e f*** command (see “Browsing Procedures” on page 220), the base objects are all the procedures whose names begin with the letter **f**. Another example: if you open the Browse window by selecting **Browse** → **Procedures**, the base objects are all the procedures in your program. Each time you change the object type you want to browse by using the **Object** menu, the newly loaded objects become the new base set.
2. The Browse window hides any remaining objects whose names do not match the patterns listed in the **Show** field of the **Define Filters** dialog box.
3. The Browse window hides any remaining objects whose names match the patterns listed in the **Hide** field of the **Define Filters** dialog box.
4. The Browse window hides any remaining objects that match the other filters enabled in the **Filter** menu.

For example, if the user-defined filters are:

- **Show** **f a***
- **Hide** **f a b***

only those objects from the base object set whose names start with **f a** but not **f a b** are displayed.



Note

The base set (described above) does not change based on user-defined filters or the selections you make in the **Filter** menu, which only affect what is displayed in the Browse window. However, the base set does change based on your selections in the **Object** menu.

Browse Window Headings

In procedure, user type, and source file Browse windows, displayed data has additional formatting. In each of these cases, items known as *headings* appear with either a plus sign (+) or a minus sign (-) next to their entry in the primary column.

Clicking a plus sign expands the children of that heading; clicking a minus sign contracts the children. The meaning of each of the headings depends on what type of objects are displayed. When you first open a Browse window, all headings appear fully expanded (a minus sign is displayed next to each heading, and all children are visible).

In filtering and sorting, the Browse window treats headings in the same fashion as regular objects. Those headings colored the same as keywords, however, are of a different type than the displayed base object type. If the selected heading is colored like a keyword, certain **Tools** options appear dimmed and certain right-click menu options do not appear.

For example, in the procedure Browse window all headings appear colored like keywords, indicating that they are types and not procedures. As a result, **Tools** → **Browse References** (among others) is unavailable when such an entry is selected. For more specific information about headings in procedure Browse windows, see “Headings in the Procedures Browse Window” on page 227.

Browsing Procedures

You can open a Browse window that displays procedures by selecting a menu option, using a shortcut, or issuing a command. Depending on how you open the Browse window, it displays either all the procedures in a program or some subset of procedures.

- To open a Browse window displaying all procedures in your program, do one of the following:
 - From the Debugger, select **Browse** → **Procedures**.
 - From an existing Browse window that is not currently displaying procedures, select **Object** → **Procedures**.
 - From the Procedure Locator located on the Debugger navigation bar, select **Browse procedures in program**.

- From the command pane, issue the **browse procedures** or **browse procs** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying only the procedures in a specific file, do one of the following:
 - While viewing the file in the Debugger, select **Browse procedures in current file** from the Procedure Locator on the Debugger navigation bar.
 - Double-click the file in a Browse window displaying source files.
 - From the command pane, issue the **e** command with the arguments **"file" #*** (for example, **e "test.c" #***). For information about the **e** command, see Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying procedures that match a specific pattern, issue the **e** command with a pattern that matches more than one procedure in your program. For example, issuing the **e f*** command opens a Browse window on all the procedures in your program that begin with the letter **f**. For information about the **e** command, see Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying the procedures called from a specific procedure or the procedures that call a specific procedure, right-click a procedure in the source pane and select **Browse Other** → **Browse Callees** or **Browse Other** → **Browse Callers**, respectively, from the shortcut menu.



Note

Sometimes MULTI opens a modal dialog box version of the Browse window when you have not performed any of the preceding actions. This usually happens when you have issued a command such as **b *** and specified a pattern that matches more than one procedure or an overloaded C++ procedure. You can use this version of the Browse window to specify the procedures you want the issued command to operate on so that the command can continue. For an illustration and instructions on how to use this dialog box, see “Procedure Ambiguities and the Browse Dialog Box” on page 227. For information about the **b** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Browsing Procedures General Information

When the Browse window opens, it may contain all the procedures in the program being debugged, or just those meeting all the starting criteria as described above. For example, it may display the procedures that match the wildcard pattern of an **e** command, or the callers of a procedure. But whenever you switch to a different object type with the **Object** menu and then select **Object** → **Procedures** again, the Browse window will display all of your program's procedures.

Procedures are color-coded in the Browse window according to MULTI's color settings. They are displayed in the following way:

- Procedure headings are displayed in the color used to represent keywords (see “Headings in the Procedures Browse Window” on page 227).
- Procedures without source code are displayed in gray.
- Procedures that are inlined are displayed in the color used to represent unused code.
- Procedures that are static (that are not also inlined or lacking source code) are displayed in the color used to represent comments.
- All other procedures are displayed in the normal text color.

Browsing Procedures Show Menu

The following table describes the options available in the **Show** menu when you browse procedures. It also lists which options are enabled by default.



Note

The primary column when you browse procedures is the **Procedure Name** column. The formatting options listed next apply only to this column, which is always displayed when you browse procedures.

Option	Default	Meaning
Name	Off	Formats the Procedure Name column (the primary column) so that the use name is displayed.

Option	Default	Meaning
Mangled Name	Off	Formats the Procedure Name column (the primary column) so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	On	Formats the Procedure Name column (the primary column) so that the unscoped portion of the use name is displayed. For example, in C++, a procedure with the name <code>Foo::bar(int a)</code> would be displayed as <code>bar(int a)</code> . For many entries, this is the same as what would be displayed under the Name option. For more information, see “Headings in the Procedures Browse Window” on page 227.
Breakpoint	On	Shows or hides the BP column. This column displays the first breakdot of the procedure. If a breakpoint is set at the first executable line of the procedure, the icon for the corresponding breakpoint type is shown; otherwise, a green dot is shown. To set a breakpoint at the first executable line of the procedure, click the green dot.
Object File	On	Shows or hides the Object File column. This column displays the object file in which the procedure is located.
Source File	Off	Shows or hides the Source File column. This column displays the source file in which the procedure is located.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the procedure is located, if any.
Library	Off	Shows or hides the Library column. This column displays the name of the library in which the procedure is located, if any.
Address	Off	Shows or hides the Address column. This column displays the address of the procedure.
Size	Off	Shows or hides the Size column. This column displays the size of the procedure in bytes.

Option	Default	Meaning
Type	On	Shows or hides the Type column. This column displays: • GI if the procedure is an inlined, non-static procedure. • SI if the procedure is an inlined, static procedure. • G if the procedure is a not-inlined, non-static procedure. • S if the procedure is a not-inlined, static procedure.

Browsing Procedures Mouse Operations

The following table describes the results of mouse operations on procedures in the Browse window.

Mouse action	Meaning
Click	Displays the selected procedure in the Debugger's source pane. If you click in the BP column, the Debugger will either insert a breakpoint at the selected procedure if no breakpoint is there, or remove the breakpoint there if one already exists. The breakpoint is set at the first executable line of the procedure.
Double-click	Opens a new window displaying the cross references for the selected procedure.
Right-click	Opens a shortcut menu. For more details, see “Browsing Procedures Tools Menu” on page 224.

Browsing Procedures Tools Menu

The following table describes the options available in the **Tools** menu when you browse procedures. Additionally, the table describes shortcut menu options that become available when you right-click content in the Browse window. The table's “Location” column specifies whether the option appears in the **Tools** menu, the shortcut menu, or both menus.

Option	Location	Meaning
Contract All	Both	Contracts every heading.

Option	Location	Meaning
Expand All	Both	Expands every heading (this is the way the Browse window looks when first opened).
Global Scope On/Off	Tools menu	Toggles whether global scope is enabled or not. See “Headings in the Procedures Browse Window” on page 227.
Browse References	Both	Displays the selected procedure's cross references (if any) in another Browse window.
Browse Callers	Both	Displays the selected procedure's callers in another Browse window.
Browse Callees	Both	Displays the selected procedure's callees in another Browse window.
Show in Editor	Both	Opens the selected procedure in MULTI's Editor.
Show in Tree Browser	Both	Opens a Tree Browser window to show the selected procedure's static call graph.
Set Breakpoint	Shortcut menu (BP column only)	Sets a software breakpoint. See also the b command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book
Set and Edit Breakpoint	Shortcut menu (BP column only)	Opens the Software Breakpoint Editor window, which allows you to specify and set a software breakpoint. See the b command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book, and also “Creating and Editing Software Breakpoints” on page 130.
Set Any Task Breakpoint	Shortcut menu (BP column only)	Sets a software breakpoint which can be hit by any task. This option is available only when connected to a target which supports this type of breakpoint. See the sb command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Edit Breakpoint	Shortcut menu (BP column only)	Opens the Software Breakpoint Editor , which allows you to edit the software breakpoint set on the line.
Remove Breakpoint	Shortcut menu (BP column only)	Removes the software breakpoint from the line.
Enable Breakpoint	Shortcut menu (BP column only)	Enables the software breakpoint located on the line.

Option	Location	Meaning
Disable Breakpoint	Shortcut menu (BP column only)	Disables the software breakpoint located on the line.
Set Hardware Breakpoint	Shortcut menu (BP column only)	Sets a hardware breakpoint on the line.
Edit Hardware Breakpoint	Shortcut menu (BP column only)	Opens the Hardware Breakpoint Editor , which allows you to edit the hardware breakpoint set on the line.
Remove Hardware Breakpoint	Shortcut menu (BP column only)	Removes the hardware breakpoint from the line.
Enable Hardware Breakpoint	Shortcut menu (BP column only)	Enables the hardware breakpoint located on the line.
Disable Hardware Breakpoint	Shortcut menu (BP column only)	Disables the hardware breakpoint located on the line.
Set Tracepoint	Shortcut menu (BP column only)	Opens the Tracepoint Editor , which allows you to set a tracepoint.
Edit Tracepoint	Shortcut menu (BP column only)	Opens the Tracepoint Editor , which allows you to edit the tracepoint.
Remove Tracepoint	Shortcut menu (BP column only)	Removes the tracepoint.
Enable Tracepoint	Shortcut menu (BP column only)	Enables the tracepoint.
Disable Tracepoint	Shortcut menu (BP column only)	Disables the tracepoint.
Set BPs on Entries	Both	Sets breakpoints at the first executable line of all displayed procedures.
Delete BPs on Entries	Both	Deletes breakpoints at the first executable line of all displayed procedures.
Breakpoint Window	Shortcut menu (BP column only)	Opens the Breakpoints window for the program being debugged.

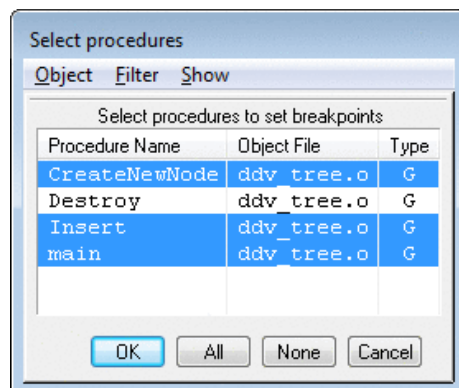
Headings in the Procedures Browse Window

As noted in “Browse Window Headings” on page 220, some entries may be headings when you browse procedures. For procedures, these headings are the “scopes” (or enclosing types) of the procedures listed as children of the heading. For example, in C++ we might have a class `Foo` with a member function `bar(int a)`. The full use name of this function would then be `Foo::bar(int a)`, with `Foo` being the scope and `bar(int a)` the unqualified name of the procedure.

To collect all unscoped types into a global scope, use **Tools → Global Scope On/Off** to enable the global scope option (see “Browsing Procedures Tools Menu” on page 224). This will place all globally accessible functions into a fake scope, to ease viewing by allowing all unscoped names to be hidden by contracting the heading labeled `<global_scope>`.

Procedure Ambiguities and the Browse Dialog Box

Sometimes MULTI will display a modal dialog box version of the Browse window for procedures. This happens if, during a command (such as **b ***), you have specified a pattern that matches more than one procedure or an overloaded C++ procedure. This results in a procedure name ambiguity, and the command cannot continue. You can use this dialog box to specify which procedure you would like the issued command to operate on, and let the command continue.



By selecting the rows in the Browse dialog box, you can specify which procedures should be used in the action or command to resolve the procedure name ambiguity. Depending on the command, you may be allowed to select several procedures, or

just one. After you have made your choice, you can click one of the buttons available at the bottom of the Browse dialog box:

Button	Meaning
OK	Accepts the current selections.
All	Selects all of the procedures displayed in the Browse dialog box, if selecting multiple procedures is applicable.
None	Ensures that none of the procedures displayed in the Browse dialog box is selected, if making no selection is applicable.
Cancel	Closes the Browse dialog box and cancels whatever operation caused the dialog box to appear.

Browsing Global Variables

To open a Browse window for global variables, do one of the following:

- From the Debugger, select **Browse** → **Globals**.
- From the command pane, issue the **browse globals** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- From an existing Browse window, select **Object** → **Globals**.

Browse Globals General Information

Global variables are color-coded in the Browse window according to MULTI's color settings. They are displayed in the following way:

- Static global variables are displayed in the color for comments.
- All other global variables are displayed in the normal text color.

Browsing Globals Show Menu

The following table describes the options available in the **Show** menu when you browse global variables. It also lists which options are enabled by default.



Note

The primary column when you browse global variables is the **Global Name** column. The formatting options listed next apply only to this column, which is always displayed when you browse global variables.

Option	Default	Meaning
Name	On	Formats the Global Name column (the primary column) so that the use name is displayed.
Mangled Name	Off	Formats the Global Name column (the primary column) so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	Off	Equivalent to the Name option.
Object File	Off	Shows or hides the Object File column. This column displays the name of the object file in which the global variable is referred to or defined.
Module	On	Shows or hides the Module column. This column displays the name of the module in which the global variable is located, if any.
Library	Off	Shows or hides the Library column. This column displays the name of the library in which the global variable is located, if any.
Address	Off	Shows or hides the Address column. This column displays the address of the global variable.
Size	Off	Shows or hides the Size column. This column displays the size of the global variable.
Type	On	Shows or hides the Type column. This column displays: • G if the global variable is non-static. • S if the global variable is static.

Browsing Globals Mouse Operations

The following table describes the results of mouse operations on global variables in the Browse window.

Mouse action	Meaning
Click	Displays the selected global variable's definition in the Debugger's source pane, if it can be found.
Double-click	Opens a new window displaying the cross-references for the selected variable.
Right-click	Opens a shortcut menu. For more details, see “Browsing Globals Tools Menu” on page 230.

Browsing Globals Tools Menu

The following table lists the options available in the **Tools** menu when you browse global variables. The same menu is displayed when you right-click content in the Browse window.

Option	Meaning
Print Value	Prints the value of the selected global variable in the command pane. This value is available only if your process is running.
View Value	Opens a Data Explorer to show the value of the selected global variable. This value is available only if your process is running.
Go To Definition	Displays the selected global variable's definition in the Debugger's source pane, if it can be found.
Browse References	Shows the clicked global variable's cross references in a new Browse window.

Browsing Source Files

- To open a Browse window displaying all source files, do one of the following:
 - From the Debugger, select **Browse → Files**.
 - From the File Locator located on the Debugger navigation bar, select **Browse all source files in program**.
 - From an existing Browse window, select **Object → Files**.

- From the command pane, issue the **browse files** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying a more limited selection of source files, issue the **e** command with a suitable pattern (**e** * .c, for example). For information about the **e** command, see Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying all of the files included by the current file, right-click an empty spot in the Debugger's source pane and select **Browse Includers Of This File**.
- To open a Browse window displaying all of the files that this file includes, right-click an empty spot in the Debugger's source pane and select **Browse Files Included By This File**.

Browsing Source Files General Information

The name of each source file is listed in the Browse window. Source files are color-coded according to MULTI's color settings. They are displayed in the following way:

- Source files that do not define any procedures are displayed in gray.
- Source file headings are displayed in the color used for keywords (see “Headings in the Source File Browse Window” on page 233).
- All other source files are displayed in the normal text color.

Browsing Source Files Show Menu

The following table describes the options available in the **Show** menu when you browse source files. It also lists which options are enabled by default.



Note

The primary column when you browse source files is the **Source File Name** column. The formatting options listed next apply only to this column, which is always displayed when you browse source files.

Option	Default	Meaning
Full Name	Off	Formats the Source File Name column (the primary column) so that the full path is displayed.
Base Name	On	Formats the Source File Name column (the primary column) so that only the actual file name is displayed.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the source file is located, if any.

Browsing Source Files Mouse Operations

The following table describes the results of mouse operations on source files in the Browse window.

Mouse action	Meaning
Click	Displays the selected source file in the Debugger's source pane.
Double-click	Opens a Browse window of procedures defined in the selected source file, if any.
Right-click	Opens a shortcut menu. For more details, see “Browsing Source Files Tools Menu” on page 232.

Browsing Source Files Tools Menu

The following table describes the options available in the **Tools** menu when you browse source files. The same menu is displayed when you right-click content in the Browse window.

Option	Meaning
Browse Procedures in File	Opens a Browse window of procedures defined in the selected source file, if any.
Contract All	Contracts every heading.
Expand All	Expands every heading (this is the way the Browse window looks when first opened).
Show in Editor	Opens the selected file in MULTI's Editor.
Show in Tree Browser	Opens a Tree Browser window to show the reference relationships between the selected source file and other source files.

Option	Meaning
Graph Includes	Opens a Graph View window that displays an include file dependency graph centered on the selected source file.

Headings in the Source File Browse Window

As noted in “Browse Window Headings” on page 220, some entries may be headings when you browse source files. For source files, these headings are the directory path of the source files. For example, if there was a Browse window open on a single file `file.c` contained within the directory `directory`, there would be a single heading labeled `directory` with `file.c` as its only child.

Browsing Data Types

To open a Browse window for data types, do one of the following:

- From the Debugger, select **Browse** → **All Types**.
- From an existing Browse window, select **Object** → **Types**.
- From the command pane, issue the **browse types** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Browsing Data Types Show Menu

The following table describes the options available in the **Show** menu when you browse data types. It also lists which options are enabled by default.



Note

Only one column is displayed when you browse data types. You cannot hide this column.

Option	Default	Meaning
Name	Off	Formats the displayed names so that the use name is displayed.

Option	Default	Meaning
Mangled Name	Off	Formats the displayed names so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	On	Formats the displayed names so that the unscoped portion of the use name is displayed. For example, in C++, a data type <code>bar</code> inside the namespace <code>Foo</code> would have the use name <code>Foo::bar</code> . Under this option, this name would be displayed simply as <code>bar</code> , with a parent heading named <code>Foo</code> . For many entries, this is the same as what would be displayed under the Name option. For more information, see “Headings for Data Types Browse Window” on page 235.

Browsing Data Types Mouse Operations

The following table describes the results of mouse operations on data types in the Browse window.

Mouse action	Meaning
Click	Shows the clicked data type's definition in the Debugger source pane, if it can be found.
Double-click	Opens a new window displaying the cross references for the selected type.
Right-click	Opens a shortcut menu. For more details, see “Browsing Data Types Tools Menu” on page 234.

Browsing Data Types Tools Menu

The following table lists the options available in the **Tools** menu when you browse data types. Additionally, the table indicates which options appear in the shortcut menu that is displayed when you right-click content in the Browse window.

Option	Location	Meaning
View Struct	Both	Opens a Data Explorer to show the selected data type's structure.

Option	Location	Meaning
Contract All	Both	Contracts every heading.
Expand All	Both	Expands every heading (this is the way the Browse window looks when first opened).
Global Scope On/Off	Tools menu	Toggles whether global scope is enabled or not. See “Headings for Data Types Browse Window” on page 235.
Browse References	Both	Shows the clicked data type's cross references in a new Browse window.
Browse Superclasses	Both	Shows the clicked data type's superclasses (if any) in a new Browse window.
Browse Subclasses	Both	Shows the clicked data type's subclasses (if any) in a new Browse window.
Show in Editor	Both	Shows the clicked data type's definition in an Editor Window.
Show in Tree Browser	Both	Shows the clicked data type in a Tree Browser window so that you can browse its hierarchy information.

Headings for Data Types Browse Window

As noted in “Browse Window Headings” on page 220, some entries may be headings when you browse data types. For data types, these headings are the “scopes” (or enclosing types) of the data types listed as children of the heading. For example, in C++ we might have a namespace `Foo` with a member class `Bar`. The full use name of this type would then be `Foo::Bar`, with `Foo` being the scope and `Bar` the unqualified name of the type.

You can collect all the C-style structs in your program into a global scope by using **Tools** → **Global Scope On/Off** to enable the global scope option (see “Browsing Data Types Tools Menu” on page 234). This places all C-style structs into a fake scope, which allows you to hide all of these types by contracting the heading labeled `<global_scope>`.



Note

In C++, we also treat template types as scopes over their parameter lists. For example, displaying the class `Foo<int>` in a Browse window would have `Foo` as a heading and `<int>` as a child beneath it. This is so that multiple instantiations of the same base template type do not clutter the

display of data types, and so that they are easy to hide (by contracting the children of a template type you do not want to examine).

Browsing Cross References

When the program being debugged is compiled with cross reference information, you can open a Browse window for cross references by right-clicking a symbol in the Debugger's source pane and selecting **Browse References** from the shortcut menu that appears.

Browsing Cross References General Information

Cross references are color-coded in the Browse window according to MULTI's color settings. They are displayed as follows:

- Definitions are displayed in the normal text color.
- Declarations and common declarations are displayed in the color for dead code.
- Reads are displayed in the color for comments.
- Writes are displayed in the color for strings.
- Reads and writes (that is, cross references that both read and write) are displayed in the color for characters.
- Address references are displayed in the color for keywords.

Browsing Cross References Show Menu

The following table describes the options available in the **Show** menu when you browse cross references. It also lists which options are enabled by default.



Note

The primary column when you browse cross references is the **File Position** column. This column is always displayed when you browse cross references.

Option	Default	Meaning
Breakpoint	On	Shows or hides the BP column. This column displays a breakdot if the reference lies on a line with associated code. If there is a breakpoint set at that line already, this column displays the appropriate breakpoint icon. To set or clear a breakpoint on this line, click the green dot.
Position in Proc	On	Shows or hides the Procedure Position column. This column displays the procedure-relative line position of the reference. If the cross reference is not located in a procedure, it has no procedure-relative line position and this column is empty.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the reference is located, if any.
Column	Off	Shows or hides the Column column. This column displays the column position of the reference.
Type	On	Shows or hides the Type column. This column displays one of the following values: • <code>Def</code> if the cross reference is a definition. • <code>Decl</code> if the cross reference is declaration. • <code>C-Decl</code> if the cross reference is a common declaration. • <code>Write</code> if the cross reference is an assignment or write to the item. • <code>Read</code> if the cross reference is an access or read from the item. • <code>Read, Write</code> if the cross reference both reads from and writes to the item. For example, in C, <code>item++</code> both adds one to <code>item</code> (a write) and returns the original value (a read). • <code>Addr</code> if the cross reference is an address reference (that is, taking the address of the item).

Browsing Cross References Mouse Operations

The following table describes the results of mouse operations on cross references in the Browse window.

Mouse action	Meaning
Click	Displays the selected cross reference in the Debugger's source pane, and highlights the piece of code for the cross reference. If you click in the BP column on a green dot, the Debugger inserts a breakpoint at the location. If you click in the BP column on a breakpoint icon, the Debugger removes the breakpoint.
Double-click	Opens the selected cross reference in the Editor.
Right-click	Opens a shortcut menu. For more details, see “Browsing Cross References Tools Menu” on page 238.

Browsing Cross References Tools Menu

The following table describes the options available in the **Tools** menu when you browse cross references. Additionally, the table describes shortcut menu options that become available when you right-click content in the Browse window. The table's “Location” column specifies whether the option appears in the **Tools** menu, the shortcut menu, or both menus.

Option	Location	Meaning
Show in Editor	Both	Opens the selected cross reference in the Editor.
Breakpoint and tracepoint menu items	Shortcut menu (BP column only)	For descriptions of these menu items, see “Browsing Procedures Tools Menu” on page 224.
Set BP on Entries	Both	Sets breakpoints at all lines for all displayed cross references.
Delete BP on Entries	Both	Deletes breakpoints at all lines for all displayed cross references.
Breakpoint Window	Shortcut menu (BP column only)	Opens a Breakpoints window for the program being debugged.

To set breakpoints at all of the locations where the value of a variable (or a type's member field) is changed:

1. Obtain the references for the variable. (In the Debugger source pane, right-click the variable, and select **Browse References**.)
2. In the Browse window, hide the reads (select **Filter** → **Hide Reads**).
3. Right-click anywhere in the Browse window's source pane, and select **Set BP on Entries** from the shortcut menu.

The Tree Browser Window

You can use the Tree Browser to examine the structure of your program in several ways.

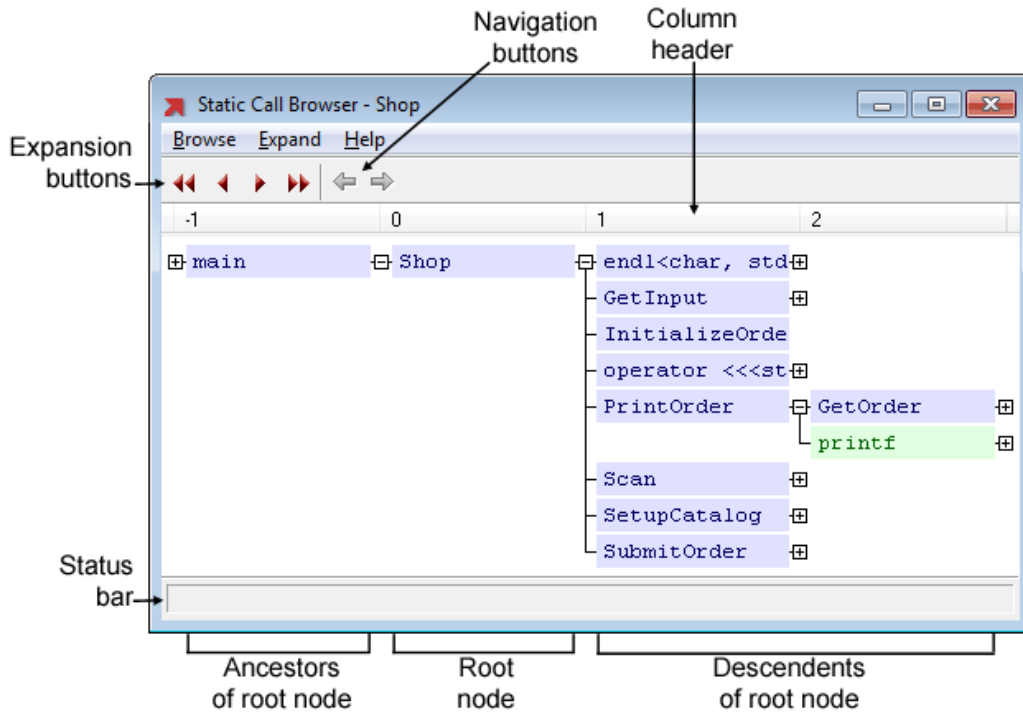
Opening a Tree Browser

To use a Tree Browser, you must be debugging a program. You can open a Tree Browser in several ways, depending on what type of information you want to view. See also:

- “Browsing Classes” on page 243
- “Browsing Static Calls By Function” on page 244
- “Browsing Static Calls By File” on page 245
- “Browsing Dynamic Calls by Function” on page 246

Using a Tree Browser

Regardless of the type of information you are viewing in the Tree Browser, the interface is basically the same.



The main part of the Tree Browser window is a tree graph, which consists of colored nodes. The meanings of the colors vary depending on the type of information you are viewing. (The following sections provide further explanation of coloring.) To customize the colors used to display information in a Tree Browser, see the `tbTypeFg` and `tbTypeBg` configuration options in “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

One node of the tree graph is called the “root node.” It is the particular function (or other object) that you are examining. The name of the root node is displayed in the title bar of the Tree Browser window. You can expand ancestors—callers, if you are looking at a function, or superclasses, if you are looking at a class—of the root node towards the left, and descendants—callees or subclasses—of the root node towards the right. You may expand away from the root node for as many levels as you want. However, if you want to look at an ancestor of a descendent of the root node, for example, you will need to reroot your graph. See “Rerooting” on page 243.

To expand ancestors or descendants of a node, click the plus sign (⊕) next to the node. To contract something you have expanded, click the minus sign (⊖). If there is no plus or minus sign, there is nothing to expand or contract.

There are four ways to expand many nodes at once. They are available from the **Expand** menu or from the expansion buttons on the toolbar.

To expand the descendents of the selected node (or of the root node if no node is selected) until there are no more descendents, until recursion is detected, until 50 levels have been expanded, or until 10,000 new nodes have been revealed by expansion, do one of the following:

- Select **Expand** → **All Descendents**.
- Click **Expand All Descendents** (▶▶).



Note

To cancel this operation, press **Esc**.

To perform a similar expansion on the ancestors of the selected node (or on the ancestors of the root node if no node is selected), do one of the following:

- Select **Expand** → **All Ancestors**.
- Click **Expand All Ancestors** (◀◀).

Sometimes you may want to expand one level instead of all the nodes. To expand one level of the selected node's descendents (or of the root node's descendents if no node is selected), do one of the following:

- Select **Expand** → **One Level of Descendents**.
- Click **Expand Descendents One Level** (▶).

To obtain similar results, you can also click every node's plus sign on the descendent side of the graph. The difference between doing this and performing one of the preceding operations is that the menu items do not expand recursive nodes.

To expand one level of the selected node's ancestors (or of the root node's ancestors if no node is selected), do one of the following:

- Select **Expand** → **One Level of Ancestors**.
- Click **Expand Ancestors One Level** (◀).

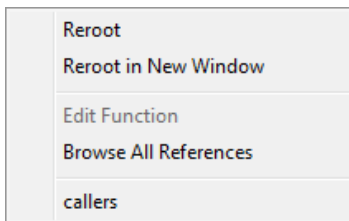
To resize a column of nodes, drag the separator on the column header to the right of the column that you want to resize.

Node Operations

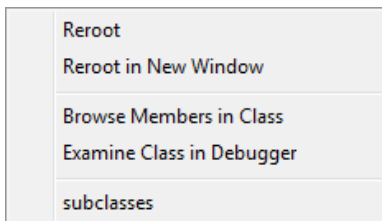
Each node is labeled with a short, descriptive name. In C++, the short name does not include class or namespace qualifiers, which come before the final double colon (: :). To view the full name in a tooltip, hover your cursor over the node.

To view more information about a node, click it. Information about the node, including its full name, is displayed in the status bar of the Tree Browser window. Clicking certain types of nodes, such as function and file nodes, also causes the source code for the node to be displayed in the Debugger source pane.

To open a shortcut menu for a node, right-click the node. The shortcut menu for a function node resembles the following:



A class node shortcut menu resembles the following:



The first two operations, **Reroot** and **Reroot in New Window**, are described in “Rerooting” on page 243. The middle portion of the menu contains various actions you can perform on the node. These actions vary depending on the type of node and are documented in “Opening a Tree Browser” on page 239. The bottom portion of the menu lists the types of ancestors or descendents a node may have. It also allows you to expand or contract them, which is equivalent to clicking the plus/minus sign on the side of the node.

Rerooting

If there is a node on the graph that you want to make the root node so that you can examine both its ancestors and its descendants, you can reroot on that node.

To reroot on a node, do one of the following:

- Click the node and select **Browse → Reroot Selected Node**.
- Right-click the node and select **Reroot**.
- Middle-click the node.

Once you reroot on a node, everything that was previously in the Tree Browser window disappears. However, the previous content of the window is stored in a history mechanism much like that in a Web browser.

To access the history, do one of the following:

- Select **Browse → Back**, or click **Back** (.
- Select **Browse → Forward**, or click **Forward** (.

To reroot a node in a new window, rather than replacing the current window contents, do one of the following:

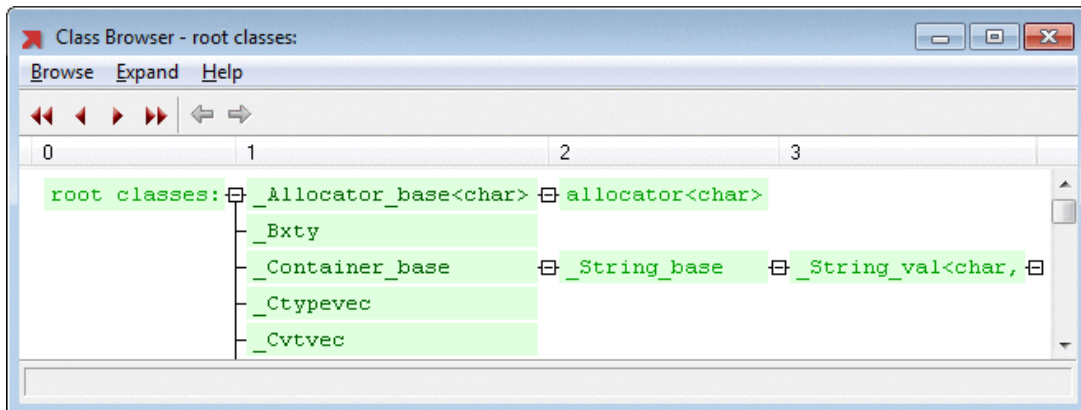
- Click the node and select **Browse → Reroot Selected Node(s) in New Window(s)**.
- Right-click the node and select **Reroot in New Window**.
- Double-middle-click the node.

Browsing Classes

To use a Tree Browser to browse your class hierarchy, do one of the following:

- From the Debugger, select **Browse → Classes**.
- In the Debugger command pane, enter the **browse classes** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A window similar to the following appears.



The children of the `root classes` node are all of the classes (including structs and unions) that do not inherit from another class. A class that is a subclass of another class is shown as a child of its parent class. The color green is used to distinguish classes and structs from unions, which are highlighted in yellow.

To view the members in a class, do one of the following:

- Double-click the class.
- Right-click the class and select **Browse Members in Class**.

To see the definition of the class in the Debugger window, do one of the following:

- Click the class.
- Right-click the class and select **Examine Class in Debugger**.

Browsing Static Calls By Function

The Tree Browser can use information from your program's symbol table to show you which functions your functions call, or those they are called by. These potential, or *static*, paths are solely based on the build-time symbol table of your program, and not on the actual run-time paths taken by the process.

To browse static calls by function, do one of the following:

- From the Debugger, select **Browse → Static Calls**.

- In the Debugger command pane, enter the **browse scalls** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A Tree Browser opens, centered on the function you are currently looking at in the Debugger source window.

To open a Tree Browser on a specific function (for example, `foo()`), do one of the following:

- In the Debugger, right-click the function and select **Browse Other** → **Browse Static Calls**.
- In the Debugger command pane, enter the following:

```
browse scalls foo
```

Color provides information about the function represented by a given node. Purple indicates a normal function for which MULTI has the source code. Green indicates a function for which no information is available—usually because the function is in a library that MULTI does not have symbols for. Pink indicates a recursion in the Tree Browser.

To view a function in the Debugger source pane, click the function node.

To edit a function, double-click the function node. This feature is also available from the shortcut menu that appears when you right-click the function node.

Browsing Static Calls By File

In addition to viewing the static call graph by function, you can also view it by file. The root file is connected to any file that contains a function called from within the root file. This may result in the root file being connected to itself.

To browse static calls by file, do one of the following:

- From the Debugger, select **Browse** → **File Calls**.
- In the Debugger command pane, enter the **browse fcalls** command. For information about this command, see “General View Commands” in Chapter

22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A Tree Browser opens on the file you are currently viewing in the Debugger.

To open a Tree Browser on a specific file (for example, **foo.c**), enter the following in the Debugger command pane:

```
browse fcalls foo.c
```

Color provides information about the file represented by a given node. Pink indicates a file that has debug information. Gray indicates a file that does not have complete debug information.

To view a file in the Debugger, click its node. To edit a file, double-click its node. Right-click a file node to open a shortcut menu that allows you to edit the file, view the file in the Debugger source pane, browse a list of functions in the file, or view file properties.

Browsing Dynamic Calls by Function

Unlike the static call graph, which shows potential calls, the dynamic call graph uses profiling data to display which functions a function actually called during run time. This feature is only available if you have collected call count profiling data with call graph support enabled. See “Overview of Profiling Methods” on page 355.

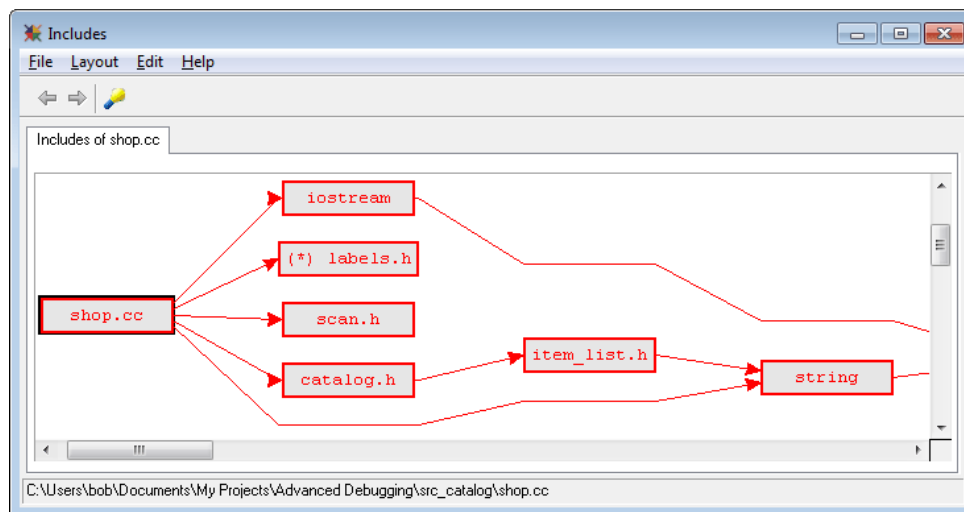
To browse the dynamic call graph, do one of the following:

- From the Debugger, select **Browse → Dynamic Calls**.
- In the Debugger command pane, enter the **browse dcalls** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book..
- To view a particular function specified by *function_name*, enter **browse dcalls *function_name*** in the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Pink nodes indicate functions that have debug information. Gray nodes indicate functions that do not have debug information.

The Graph View Window

The Graph View window displays relationships between items as a two-dimensional graph of objects. An edge between two objects in the graph indicates a relationship between the two items.

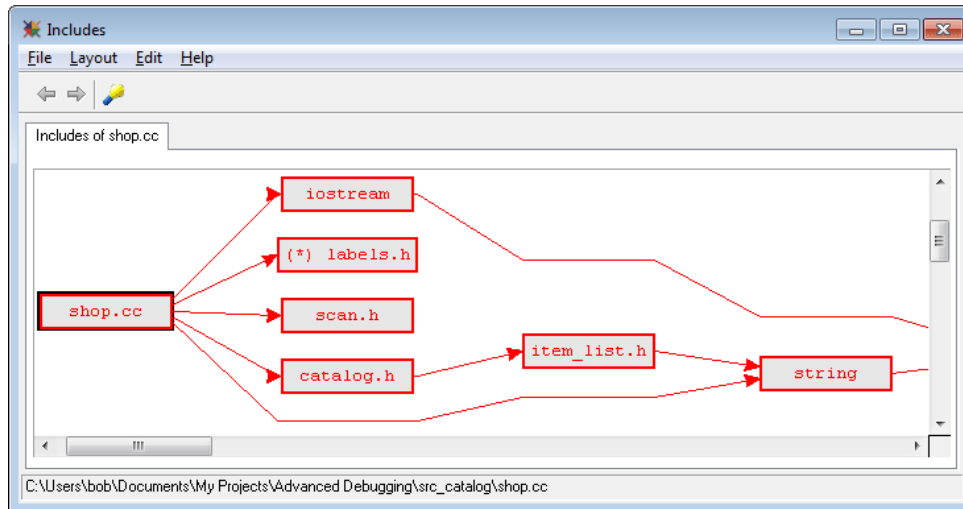


Browsing Includes

To open a Graph View window displaying an include file dependency graph, do one of the following:

- From the Debugger, select **Browse** → **Includes**.
- From the command pane, issue the **browse includes** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The resulting graph is vertically centered on the current file, and shows all files included in this file as well as all files that include this file. An edge pointing from a file to an included file indicates a dependency.



When you display the include dependency graph from a root file (that is, a file that is not itself included by anything), some files may be marked with a leading **(*)**. These are files that appear not to contribute anything to the root file and that can probably be removed from the include graph without harm.

Clicking a file node causes the file to be displayed in the Debugger source pane.

Controlling Layout and Navigating in Graph View

You can control the layout of a graph in the Graph View window via the **Layout** menu or the right-click shortcut menu. You can navigate the graph via the right-click shortcut menu, the **Search for Objects** window, or the history buttons. The following sections describe each in more detail.

The Layout Menu

The Graph View **Layout** menu contains the following options:

Option	Meaning
Redo Layout	Recalculates the layout of the graph. This option may be used to improve the layout of the graph after collapsing nodes.
Vertical Layout	Formats the graph vertically (edges are directed downward). When disabled, the graph is formatted horizontally (edges are directed rightward).

The Graph View Shortcut Menu

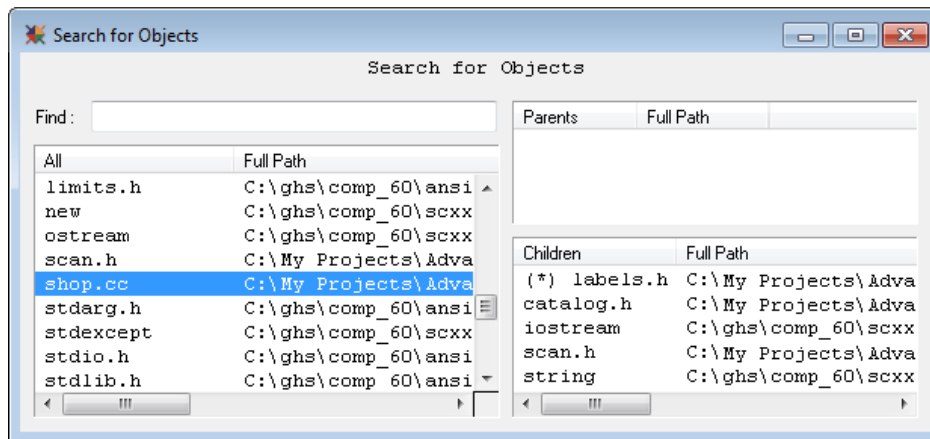
Right-clicking an object in the graph gives a shortcut menu with the following options:

Option	Meaning
Redo Layout	Recalculates the layout of the graph. This option may be used to improve the layout of the graph after collapsing nodes.
Go To Child	Opens a submenu listing children of the selected object. Choosing one of these children scrolls the graph to that object. If the object has more than 10 children, an additional option View Full List appears in the submenu. Selecting this option opens the Search for Objects window.
Go To Parent	Opens a submenu listing parents of the selected object. Choosing one of these parents scrolls the graph to that object. If the object has more than 10 parents, an additional option View Full List appears in the submenu. Selecting this option opens the Search for Objects window.
Collapse Node	Removes the object and all descendants that have only one parent. You can display the object again by selecting Expand Children from the object's parent or by selecting Expand Parent from any of the children that are still visible.
Collapse Children	Removes all children of the object and then traverses the subtrees of each child, removing those objects that no longer show parents. In a simple tree, the net effect is to remove all nodes in the subtree rooted at the object. You can display the children again by choosing Expand Children .
Expand Children	Redisplays the object's children, if they were previously collapsed. Also redisplay the children's children, and so on, since they now have a visible parent.
Collapse Parents	Similar to Collapse Children , except operating on the object's parents, the parents' parents, and so on.
Expand Parents	Similar to Expand Children , except operating on the object's parents, the parents' parents, and so on.
Edit File	Opens the file in an editor.
Re-center Graph	Re-centers the graph on the selected file, producing a graph of files that include the selected file, and files that it includes. This creates a new entry in the history.

The Search for Objects Window

The **Search for Objects** window provides additional navigation tools for Graph View. To open the **Search for Objects** window, do one of the following:

- In the Graph View toolbar, click the **Search** button (🔍).
- Select **Edit** → **Search**.



The **Search for Objects** window consists of the following elements:

- The **Find** text field, which allows you to specify a filter for the entries in the **All** list. Filters may include wildcards and are case-insensitive.
- The **All** list displays a list of all objects that fit the current filter. Selecting an item in the **All** list scrolls to and selects the corresponding object in Graph View. Similarly, selecting an object in Graph View selects the corresponding list item in the **All** list.
- The **Parents** list displays a list of all parents of the selected object. Entries that appear dimmed indicate objects that are currently collapsed. Selecting an item in the **Parents** list causes Graph View to scroll to, but not select, the corresponding object.
- The **Children** list displays a list of all children of the selected object. Entries that appear dimmed indicate objects that are currently collapsed. Selecting an item in the **Children** list causes Graph View to scroll to, but not select, the corresponding object.

The History Buttons

The Graph View history buttons allow you to switch between different versions of the graph. Clicking the **Back** button () moves back in the history, while clicking the **Forward** button () moves forward in the history.

Chapter 13

Using the Register Explorer

Contents

The Register View Window	254
The Register Information Window	270
The Modify Register Definition Dialog	276
The Register Setup Dialog	279
The Register Search Window	280
Using Debugger Commands to Work with Registers	282
Customizing Registers	285

The Register Explorer allows you to configure how registers are defined, accessed, and displayed within the Debugger. The Register Explorer includes the **Register View** window, which allows you to view the values of registers within the Debugger and the **Register Information** window, which shows you detailed information about a register and allows you to add and change register definitions.



Note

Despite its name, the Register Explorer is not limited to working with the physical registers on your target processor. The Register Explorer can also work with objects that derive their values from other registers, from locations in memory, or even from the result of a Debugger expression. All of these objects are treated in the same manner as physical target registers. In this chapter, the term *register* includes everything that can be described in a Register Explorer definitions file.

Your Green Hills Compiler installation includes one or more register definition files for your target architecture. For information about these files' names and locations, see “Register Explorer Startup” on page 286. For information about their format, see “The GRD Register Definition Format” on page 776.

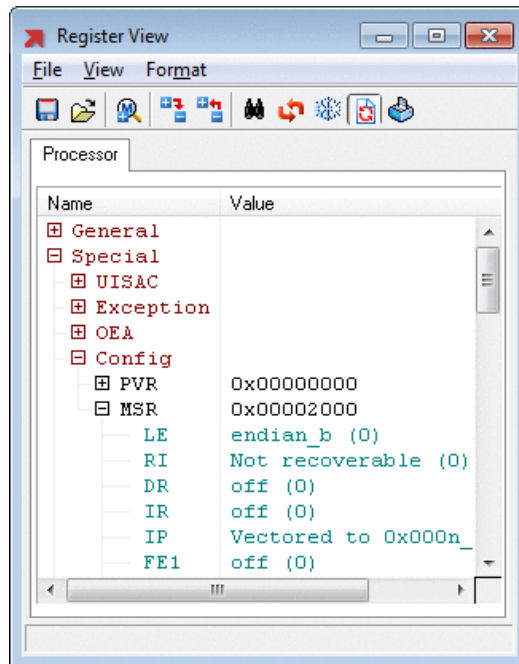
The Register View Window

The Register Explorer provides a **Register View** window specifically for viewing registers. You can use this window to create multiple views of a target's registers and easily switch between them. Since the window can hide registers and groups, you can use it to pare down dozens of registers into the ones that are most relevant for debugging your application.

To open the **Register View** window from a Debugger, you can do one of the following:

- Click the **Registers** button () on the Debugger toolbar.
- Select **View** → **Registers**.
- Run the command **regview** in the Debugger command pane. For information about the **regview** command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Register View** window shows all the registers for your target, your custom registers, and various configuration controls. An example Power Architecture **Register View** window is shown below.



The main components of the **Register View** window include:

- *Menu Bar* — Contains entries for various operations that allow you to configure the tabs and appearance of the **Register View** window. For a description of the individual menus and options, see “The Menu Bar” on page 256.
- *Toolbar* — Provides shortcuts for often-used operations. For a description of the individual buttons, see “Toolbar” on page 259.
- *View tabs* — Show the registers of your target organized in different ways. For information about the default tabs, see “Tabs” on page 260. For instructions on how to create custom tabs, see “Customizing the Register View Window” on page 263.
- *Register tree* — Each tab contains a two-column register tree. You can select registers and groups from this tree and perform actions on them using the right-click menu. For details, see “Register Tree” on page 260.

The Menu Bar

The **Register View** window has three menus, **File**, **View**, and **Format**.

The **File** menu contains the menu items listed in the following table.

Item	Meaning
Save All Register Values into File	Saves values of all registers into a text file.
Save All Register Values in Active Tab into File	Saves the values of all registers in the current tab into a file.
Save Selected Register Values into File	Saves the values of the selected registers in the current tab into a file
Load Register Values from File	Loads register values from a file in the saved register value format. See a saved register value file for an example.
Save Register Definitions Created on the Fly	Saves to a .grd file the register definitions modified during the current debugging session, registers defined with the regadd command, or registers defined from a Data Explorer's View → View “variable” as Register menu item. See “The Register Setup Dialog” on page 279.
Load Register Definitions from File	Loads register definitions from a .grd file and adds them to the definitions already present.
Unload Register Definitions from File	Unloads register definitions from an incrementally loaded .grd file.
Add Memory-mapped Register	Creates a memory-mapped register for a memory address.
Reinitialize Register Information	Reinitialize the register information from the default register definition file(s). This is a slow process for some targets.
Reuse Register Information Window	Toggles the option that specifies whether to reuse an existing Register Information window or open an additional Register Information window when a new register is viewed.

Item	Meaning
Freeze	Freezes the Register View window so that it does not automatically refresh. When the Register View window is not frozen, the values of the registers in the window are updated each time the process stops. If the window is frozen, the register values will not be updated each time the process stops.
Print	Prints all register values.
Close	Closes the Register View window.

The **View** menu contains the following menu items.

Item	Meaning
Search Register	Launches the Register Search window, see “The Register Search Window” on page 280).
Show Hidden Registers	Controls whether registers that are hidden on the active tab are displayed in the register tree. When a check mark appears beside this menu item, registers and groups that have been hidden on the active tab will be displayed in the register tree as light gray entries. When hidden items are displayed, you can edit them to remove the hidden attribute. When no check mark appears next to this menu item, registers and groups that are hidden on the active tab are not displayed in the register tree. To toggle between these settings, select the Show Hidden Registers menu item.
Add New Tab	Adds a new tab to the Register View window. See “Adding a New Register View Window Tab” on page 264.
Remove Active Tab	Deletes the active tab from the Register View window. See “Removing Unwanted Tabs” on page 266.
Promote Active Tab	Moves the tab one to the left in the ordering of tabs at the top of the Register View window. See “Modifying the Ordering of Tabs” on page 265.
Demote Active Tab	Moves the tab one to the right in the ordering of tabs at the top of the Register View window. See “Modifying the Ordering of Tabs” on page 265.
Unroot Active Tab	Makes the active tab display groups starting from the root of the register group tree, if the tree has been re-rooted. See “Changing the Root of the Register Tree” on page 266.

Item	Meaning
Refresh	Refreshes all of the register values displayed in the register tree if the Register View window is not frozen.
Expand All	Expands all nodes on the current tab of the Register View window.
Collapse All	Contracts all nodes on the current tab of the Register View window.

The **Format** menu contains the menu items listed in the table below. You can use this menu to configure the tabs and appearance of the **Register View** window.

Item	Meaning
Natural	When selected, register values will be displayed in their default format. This default format corresponds to how values of the register's type are normally displayed in Data Explorers.
Decimal	When selected, register values will be displayed in signed base 10 integer notation whenever applicable.
Hexadecimal	When selected, register values will be displayed in unsigned base 16 notation whenever applicable.
Binary	When selected, register values will be displayed in unsigned binary notation whenever applicable.
Octal	When selected, register values will be displayed in unsigned base 8 notation whenever applicable.
View Alternate	When selected, register values will be displayed in the natural format as well as in a secondary format. For example, an enumeration value is displayed along with an integer value when this menu item is selected.
Pad Hex Values	When selected, any hexadecimal values that are displayed will include leading zeros to their full bit width. When cleared, hexadecimal values are displayed without leading zeros.
Expand Value	When selected, any registers that are pointers to values stored in memory will be automatically dereferenced and the contents of memory at that location will be displayed. When cleared, only the pointer is shown and not the value to which it points.
Show Changes	When selected, any register values that change while you are stepping through your program are highlighted to make them distinct from registers whose values did not change. When cleared, all registers are displayed in the same way, regardless of whether their values changed or not.

Item	Meaning
Expand Register Bitfields	When selected, the bitfields contained within registers will be displayed as structures (individual values separated by vertical bars). When cleared, the bitfield registers are displayed as integer hexadecimal or decimal constants.

Toolbar

Directly underneath the menu bar is the toolbar, which contains the following buttons:

Button	Meaning
	Saves the values of the selected registers in the current tab into a file.
	Loads register values from a file in the saved register value format. See a saved register value file for an example.
	Creates a memory-mapped register for a memory address.
	Expands all nodes on the current tab of the Register View window.
	Contracts all nodes on the current tab of the Register View window.
	Launches the Register Search window (see “The Register Search Window” on page 280) so that you can search a register you want.
	Refreshes all of the register values displayed in the register tree if the Register View window is not frozen.
	Freezes the Register View window so that it does not automatically refresh. When the Register View window is not frozen, the values of the registers in the window are updated each time the process stops. If the window is frozen, the register values will not be updated each time the process stops.
	Toggles the option that specifies whether to reuse an existing Register Information window or open an additional Register Information window when a new register is viewed.
	Prints all register values.
	Closes the Register View window. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons. To access this option, select Config → Options → General Tab.

Tabs

Directly underneath the toolbar is a list of tabs. Each tab of the **Register View** window shows the registers of your target organized in a different way. In addition to tabs that you create, you may see default tabs, which are automatically created. The possible default tabs are listed below:

- The **Processor** tab usually contains all of the registers present on the target processor.
- The **Board** tab contains all memory-mapped registers and dynamic registers defined in the Debugger, as well as indirect registers. These registers are usually specific to the target board you are using.
- The **Ungrouped** tab displays any user-defined registers that have not been placed into any register group.

These default tabs are present only if they contain registers. For example, the **Ungrouped** tab will probably not appear unless you have defined your own registers and these registers have not been placed in any group.

Each register tab except for **Ungrouped** views the same hierarchical register structure as defined in the register definition files. Different tabs can be configured to display different subsets of the available registers. You can alter the visibility of registers and groups on a single tab without affecting other tabs. See “Selectively Displaying Registers and Groups” on page 264.

To switch between tabs, click the name of the tab you want to view in the tab list. For information about adding a new tab or performing other operations on tabs, see “Customizing the Register View Window” on page 263.

Register Tree

Underneath the tab list is a tree of the registers and groups that are visible for the active tab. The column on the left displays the names of register groups and the names of registers. The column on the right shows the current value for registers. If the list of registers and groups is larger than the visible area of the register tree, scroll to view parts of the tree that are not visible.

Each group in the left column has a plus or minus icon next to it. To expand or contract the group, click this icon. When the group is expanded, the minus sign icon

will be shown and all of the items contained in that register group will be visible in the register tree. When the group is contracted, the plus sign icon will be shown and the items contained in that register group will not be visible in the register tree. Registers containing a pointer also display plus/minus icons.

To select a single row of the register tree, click the row. To select a contiguous range of rows of the register tree, use **Shift**+click. To make a noncontiguous selection of rows, use **Ctrl**+click.

To operate on an entry in the register tree, right-click the entry. A shortcut menu will appear. For more information, see “Performing Actions on Registers Using the Shortcut Menus” on page 262.

To open a Data Explorer (see also “The Data Explorer Window” on page 185) or a **Register Information** window (if the register has bitfields definition) on a register that is in a row of the register tree, double-click the row.

In the **Name** column, a tooltip may be available:

- For registers or groups that have names that extend beyond the width of the **Name** column;
- For registers or groups that have an alternate name specified in the GRD file with the `long_name` or `ln` option;
- For registers that have certain special values, such as a pointer to code.

Colors are used to distinguish different objects:

- Groups are displayed in the color MULTI uses for string constants;
- Registers are displayed in the standard text color;
- Register bitfields are displayed in the color MULTI uses for customized items;
- Dereferences of pointer registers are displayed in the color MULTI uses for keywords.

For more information, see the description of “Syntax Color Settings” in “Colors Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Performing Actions on Registers Using the Shortcut Menus

When you right-click a row or selection in the register tree, a shortcut menu appears. You can use this menu to change properties of the selected items, as well as the active tab of the **Register View** window. Three types of objects can be right-clicked in the **Register View** window: a single register, a single group, or a selection that contains multiple items of the register tree. A shortcut menu appears that contains some of the items from the following table.

Menu Item	Meaning	Shown For
Hide Item or Hide Selected Registers	If the selection was visible in the register tree of the active tab, this choice will make it invisible (unless Show Hidden Items is selected). When you hide a group, all of its children are implicitly hidden as well. See “Selectively Displaying Registers and Groups” on page 264.	Single register Single group Multiple items
Show Item or Show Selected Registers	If the selection was hidden in the register tree of the active tab, this choice will make the selection visible again. See “Selectively Displaying Registers and Groups” on page 264.	Single register Single group Multiple items
Edit Contents of Register	Opens a dialog box that allows the value of the register to be modified. Editing values through this dialog is only possible for registers that do not contain structures or bitfields. Complex structures must be edited from the command line or a Data Explorer. See “Editing Register Contents” on page 267.	Single register
Open a Data Explorer on Register	Opens a Data Explorer displaying the value of the selected register or its field. See “The Data Explorer Window” on page 185.	Single register
Open an Info View on Register	Opens a Register Information window displaying the selected register's detail information. See “The Register Information Window” on page 270.	Single register

Menu Item	Meaning	Shown For
View Memory at Address in Register	Opens a Memory View window displaying the memory at the address represented by the selected register. See Chapter 15, “Using the Memory View Window” on page 323.	Single register
Reroot Tab at Group	Makes the tab display only the children of the selected group. This is useful when configuring new tabs. See “Changing the Root of the Register Tree” on page 266.	Single group
Unroot Active Tab	Reverts the effect of any rerooting commands applied to the tab. This is useful when configuring new tabs. See “Changing the Root of the Register Tree” on page 266.	Single register Single group Multiple items
Copy Values	Copies the contents of the value column for the selected item or items to the clipboard. See “Copying Register Values” on page 267.	Single register Single group Multiple items
Save Selected Register Values into File	Saves the values of the selected registers in the current tab into a file.	Single register Single group Multiple items
Load Register Values from File	Loads register values from a file in the saved register value format. See a saved register value file for an example.	Single register Single group Multiple items

Customizing the Register View Window

In addition to configuring the basic display of the **Register View** window, you can also create and store multiple display configurations that will appear as customized tabs in the **Register View** window.

Each tab in the **Register View** window contains a distinct view of the hierarchical organization of groups and registers. You can choose to display only certain registers and groups on each tab independently of other tabs. This means that you can create new tabs that only display the registers you are interested in at particular points in your process, and flip between these displays by clicking the appropriate tabs.



Note

You cannot modify the way registers are hierarchically grouped or ordered without modifying the register definition files.

The **Register View** window remembers how it was configured between debugging sessions. Settings are saved and restored based upon which register definition file is loaded when the Debugger is opened. This usually means that configurations of the registers for different target processor families are saved and restored independently of each other.

Adding a New Register View Window Tab

The first step in creating a new configuration of the **Register View** window is to add a new tab. To do this, choose **View → Add New Tab** in the **Register View** window. Use the dialog box to enter the name you want to give the new tab. The tab name must contain only alphanumeric characters and underscores. Although not required, this name should be short to keep the tab list narrower than the width of your **Register View** window, making it easier to switch tabs without scrolling through a long tab list. After you click **OK**, a new tab will appear in the list at the top of the **Register View** window and will be made the active tab.

When you create a new tab, it displays the full structure of the register tree by default. You can now show or hide registers and groups to pare this tree down to the information that is most important to you.

Selectively Displaying Registers and Groups

Some target processors may have dozens of registers that are displayed in the default register tree. You may not care about the values of many of these registers, and the ones that are important to you may be scattered throughout the register tree. By changing which items of the register tree are shown or hidden, you can display only the information that is most relevant to you. Registers and groups are hidden on a per-tab basis; hiding or showing a group on one tab does not affect its display on the other tabs.

To hide a register, right-click the register you want to hide and choose **Hide Register** from the shortcut menu.

To hide groups of registers, right-click the group and choose **Hide Group** from the shortcut menu. The group, and anything the group contained, will no longer be displayed on that tab.

To hide multiple objects, use **Shift**+click or **Ctrl**+click in the register tree to select multiple items. When you have selected everything you want to hide, right-click one of the selected items and choose **Hide Selected Items** from the shortcut menu.

To display a hidden register or group, choose **View** → **Show Hidden Registers**. All of the hidden registers and groups are now displayed as light gray items in the register tree. Right-click the item you want to display and choose **Show Item** from the menu. The name of the item now appears in black, indicating that it is no longer hidden. To show multiple items simultaneously, right-click a member of a multiple selection, similarly to how multiple items are hidden at once. To remove the dimmed items from the list, again choose the **View** → **Show Hidden Registers** option.

You can also show and hide items on a tab by using the **regtab** command. For more information about this command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

Modifying the Ordering of Tabs

You can modify the left to right tab order that appears at the top of the **Register View** window. The order of the tabs is maintained across Debugger sessions. Whenever you open a new **Register View** window, the leftmost tab in the ordering is the first one that is displayed.

To move a tab one position to the left, first click its label to make it the active tab. Then choose **View** → **Promote Active Tab** in the **Register View** window.

To move a tab one position to the right, first click its label to make it the active tab. Then choose **View** → **Demote Active Tab** in the **Register View** window.

You can also modify the ordering of tabs in the **Register View** window by using the **regtab** command. For more information about this command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

Removing Unwanted Tabs

Each open **Register View** window must contain at least one tab. If the window contains more than one tab, you can perform one of the following actions to remove an unwanted tab:

- Click the tab's label to make it the active tab, and then select **View → Remove Active Tab**.
- Use the **regtab** command. For information about this command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

Removal is permanent for customized tabs (see “Adding a New Register View Window Tab” on page 264). However, the automatically created **Processor**, **Board**, and **Ungrouped** tabs can be restored. To restore these tabs:

1. Exit MULTI.
2. Remove the relevant **.ini** configuration file from:
 - Windows 7/Vista — *user_dir\AppData\Roaming\GHS\regview*
 - Windows XP — *user_dir\Application Data\GHS\regview*
 - Linux/Solaris — *user_dir/.ghs/regview/*

Note that removing this configuration file also permanently deletes all customized tabs.

Changing the Root of the Register Tree

Newly created tabs always display the full register tree. You may only be interested in the registers in a certain group that is nested within other groups of the register tree. By showing and hiding groups, you can make all the other groups invisible, leaving only the registers you are interested in visible, along with the groups that contain them.

To hide the parents of a visible register or group, change the root of the register tree for a tab. When you change the root of the register tree on a tab, only the children of the group chosen to be the new root are displayed. To specify that a group should be the new root, right-click the group and choose **Reroot Tab at Group** from the

shortcut menu. Afterwards, the register tree on that tab now displays only the children of the group you specified to be the root.

You can display all of the groups in the default register tree and undo a rerooting operation by unrooting the register tree on a tab. First click the desired label to make it the active tab. Choose **View → Unroot Active Tab** in the **Register View** window. The tab will now display all of the default register tree's visible groups.

You can also reroot and unroot tabs by using the **regtab** command. For information about this command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

Copying Register Values

You can place a copy of any information in the register tree onto the clipboard. To make a copy, select the items you want to copy and press **Ctrl+C**. All of the rows that are selected in the register tree will be copied to the clipboard. Since only full rows can be selected, the copy will take names of groups and registers as well as their values.

Sometimes you may want to copy just the value of a particular register or set of registers and omit the register names. To do this, first select the registers whose values you want to copy. Then right-click the selection and choose **Copy Values** from the shortcut menu that appears. If you only want to copy the value of a single register, right-click the register and it will be selected automatically.

Editing Register Contents

Many registers contain values that are not interpreted as bitfields or structures, but as simple unsigned bits, floating-point values, or integers. For registers that lack complicated structures, their contents can be edited directly from the **Register View** window.

To edit the contents of a register, right-click the register you want to modify and choose **Edit Contents of Register** from the shortcut menu that appears. A dialog box will appear that shows the current value of the register. Type in the new value or a Debugger command line expression that evaluates to the new value you want the register to have. When you click **OK**, the register will be assigned either the

value you typed in or the result of evaluating your expression in the current Debugger environment.

To modify the values of registers that have complicated structures, you must do your editing from a **Register Information** window or a Data Explorer. To modify register value with the **Register Information** window:

1. Open a **Register Information** window on the register (right-click the register you want to modify and choose **Open an Info View on Register** from the shortcut menu). A **Register Information** window will appear showing the details of the register's structure.
2. Edit the register's contents as described in “Changing a Register Value” on page 275.

To modify register value with the Data Explorer:

1. Open a Data Explorer on the register (right-click the register you want to modify and choose **Open a Data Explorer on Register** from the shortcut menu). A Data Explorer will appear showing the details of the register's structure.
2. Edit the register's contents as you would edit the values of any other structure in a Data Explorer.

See Chapter 11, “Viewing and Modifying Variables with the Data Explorer” on page 183.

Controlling Refresh of Register Values

By default, the **Register View** window tries to update the values of all the registers it displays whenever your process is halted. There may be cases where you do not want the values to be automatically updated, such as when you want to keep a copy of the current values of a processor's registers to compare with the values at a future time. You may also want to suppress automatic updating if you are using a slow target connection.

To suppress the automatic updating of register values, click the  button in the **Register View** window. The button will appear to be pushed down to indicate that the window is frozen. While the window is frozen, the register values will not be automatically updated from your target. In the frozen state you may still scroll

through the register tree to view all of the register values that were present when the window was frozen. You are not allowed to switch between tabs when the **Register View** window is frozen. To unfreeze the window and allow the register values to be automatically refreshed again, click the  button again.

By default, when the value of a particular register changes, that register's entry will be highlighted in the register tree. To toggle this highlighting on and off, choose **Format → Show Changes**.

If you want to manually refresh register values in the **Register View** window while your process is halted, choose **View → Refresh**. Usually the values will be automatically refreshed each time the process is halted, but you will need to manually refresh to see any changes to volatile registers that occur while your process is halted.

Printing the Window Contents

To print the list of registers and register values in a **Register View** window, click the  button or select **File → Print** and specify the appropriate settings in the **Print Options** dialog that appears.

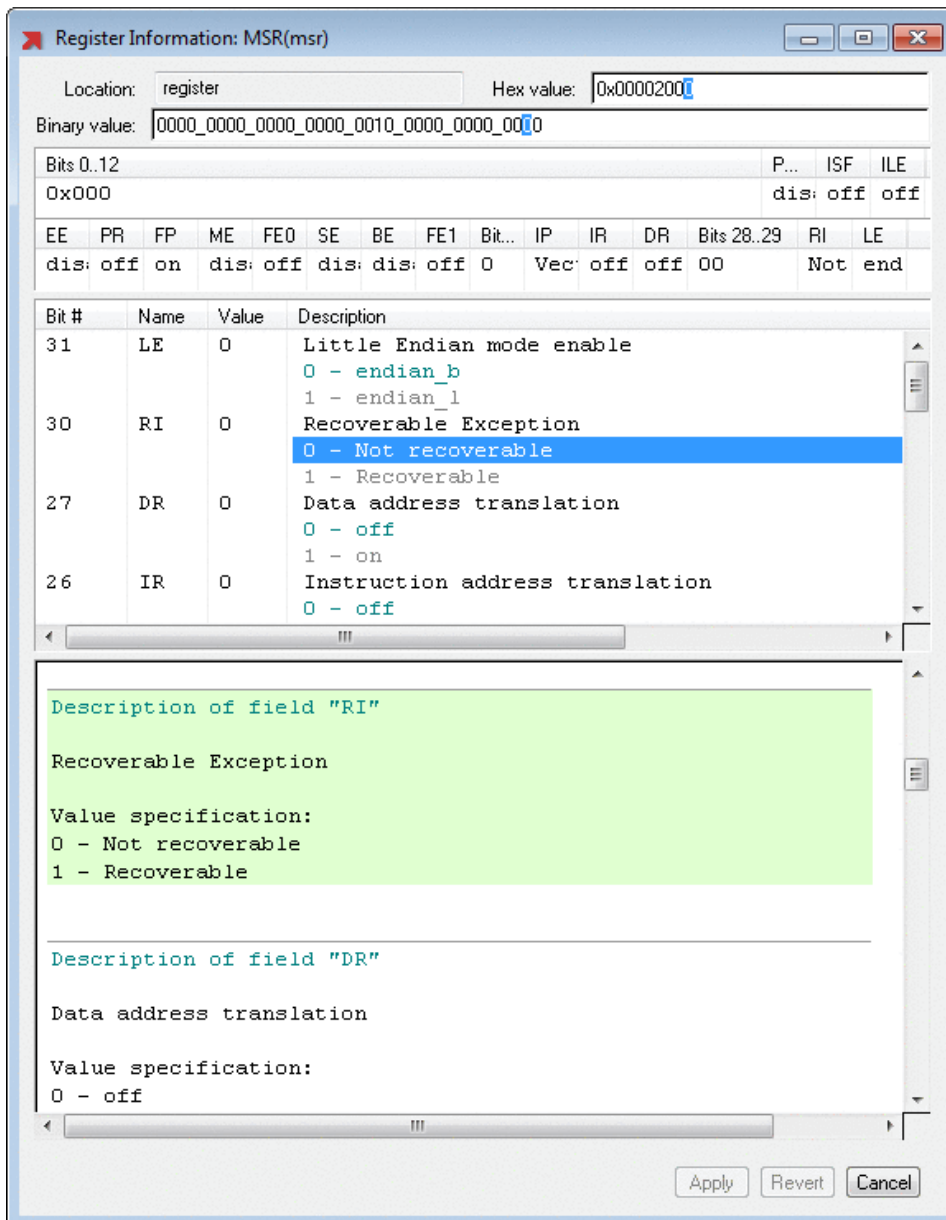
Register View Window Configuration Files

The **Register View** window tab configurations are stored in configuration files to maintain settings across Debugger sessions. These files, whose names end in **.ini**, are stored in the **regview** subdirectory of your personal configuration directory (see “Removing Unwanted Tabs” on page 266). Usually you will not need to modify these configuration files directly. If you are customizing MULTI for internal distributions, however, you may want to add tabs for your users in addition to the **Processor**, **Board**, and **Ungrouped** tabs. To do so, use the `gui_tab` clause in the register definition file. For details, see “The GRD Register Definition Format” on page 776.

The Register Information Window

The **Register Information** window shows detailed information about a register, including bitfields and documentation, and provides convenient approaches to change the register's value.

Here is an example for the Power Architecture `msr` register:



You can open a **Register Information** window in the following ways:

- In the **Register View** window's shortcut menu, choose **Open an Info View on register_name**.
- In the **Register View** window, double-click a register with a bitfield definition.
- Issue the **regview** command followed by a register name. For information about the **regview** command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Register Information** window contains the following parts:

- **Location** — Displays the register's type (register, memory-mapped, etc.) and address information, if any.
- **Hex value** — Displays the register's value, in hexadecimal.

The currently selected hexadecimal digits in the bitfield panes (see below for more information) are displayed in the color used for selections, and any digits changed in this display but not yet written to the target by clicking **Apply** are displayed in red.

- **Binary value** — Displays the register's binary value.

The bits are displayed in groups of 4 or 8 bits. To toggle between 4 and 8 bit groups, select **Show Binary in Nibbles** from the shortcut menu accessible in the bitfield panes. Bit groups are separated by an underline. As in the **Hex value** display, the currently selected binary digits in the bitfield panes (see below for more information) are displayed in the color used for selections, and any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red.

- **Concise display pane** — Displays the register's bitfield names as column headings and values in the corresponding cells. Any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red. For more information, see “Concise Display Pane” on page 272.
- **Detailed display pane** — Displays the register's detailed bitfield information. Any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red. For more information, see “Detailed Display Pane” on page 273.

- Help pane — Displays the register's documentation. For details, see “Help Pane” on page 274.

Whenever you select a bitfield in the bitfield panes, its description (if defined in the register definition file) will be shown in the help pane in a special background color.

- Button set — Allows you to write any changes made in this window to the target, revert any changes, or close the **Register Information** window. For more information, see “Button Set” on page 275.

Concise Display Pane

In the concise display pane, the register's field names are shown in column headers, and the corresponding field values are shown below the field names. The fields are displayed in order from the most significant bit to the least significant bit.

When you right-click a value cell, a shortcut menu appears with the following items:

Item	Meaning
Set Value: <i>value</i>	Sets the value to the bitfield. This menu item is displayed for a writable register's enumeration bitfields or 1 or 2 bit bitfields. The current value is dimmed.
Change Value	Allows you to provide a new value for the bitfields. This menu item is displayed for a writable register's bitfields that are not enumeration types and that are wider than 2 bits.
Modify Register Definition	Launches a Modify Register Definition dialog (see “The Modify Register Definition Dialog” on page 276) to modify the register's bitfield definitions. You can save the modified register bitfield definition into a file by choosing File → Save Register Definitions Created on the Fly from the Register View window.
Show Numeric Values	Toggles whether to show numeric or literal values for bitfields in the concise display pane.
Pad Undefined Bitfields	Toggles whether to display undefined bitfields.
Show Binary in Nibbles	Toggles whether to display binary values grouped by nibbles (4 bits) or bytes (8 bits).

Item	Meaning
Reverse Bit Numbering	Reverses the numbering for bits. This option only changes the displayed bit indices; it does not change the register's value.
Reverse Byte Order	Swaps the bytes in the register value displayed in the Register Information window. This option changes the register value in the Register Information window. Click the Apply button to write the modification to the target.

Detailed Display Pane

In the detailed display pane, displayed just below the concise display pane, more comprehensive information about each bitfield is present. The bitfields are listed in the order they are defined in the corresponding register definition file.

The register's bitfields and their possible values (for enumeration bitfields) are displayed in rows in the detailed display pane. The following columns are present in the display:

Item	Meaning
Bit #	Bit number or bit number range for a bitfield. The column's value can be changed by toggling the Reverse Bit Numbering option in the shortcut menu.
Name	Bitfield name.
Value	Bitfield's numeric value.
Description	Bitfield's description. If a bitfield has a single-line description, it will be fully displayed in the column; if a bitfield has a multiple-line description, the first line will be displayed in the column followed by . . . ; if a bitfield has no description but has a long name, the long name will be displayed in the column.

When you right-click a cell, a shortcut menu appears with the following items:

Item	Meaning
Show Possible Field Values	Toggles whether to display the possible values for each bitfield of the register. When the flag is on, an enumeration bitfield's possible values are shown below the bitfield. In the row showing a possible bitfield value, the other columns are blank. Double-click one of these bitfield value choices to change the bitfield to that value.
Other items	Other items in the shortcut menu have the same functions as the corresponding items in the shortcut menu of the concise display pane.

Help Pane

The help pane is below the detailed display pane.

At the top, it displays the register's general description in the following order:

- The register's name, aliases etc
- The register's long name (if any)
- The register's description (if any)

Then, for each bitfield, the help pane displays information in the following order:

- The bitfield's name
- The bitfield's long name (if any)
- The bitfield's description (if any)
- The bitfield's possible values (for enumeration types)

The bitfields are shown in the order they are defined in the corresponding register definition file.

Whenever you click a cell in the concise display pane or the detailed display pane, the help pane will automatically scroll to and highlight the corresponding bitfield's documentation.

Button Set

The button set is at the bottom of the **Register Information** window. It contains the following buttons:

Item	Meaning
Apply	Writes the register value displayed in the Register Information window to the target.
Revert	Discards any changes made to the register value and reloads the current register value.
Close	Closes the Register Information window. If any unwritten changes have been made, those changes will be discarded.

Changing a Register Value

The **Register Information** window provides various ways to change a register's value, provided the register is not read-only:

- In the **Hex value**, **Binary value**, the concise display pane's numeric cells and the detailed display pane's **Value** column, you can directly modify the register value:

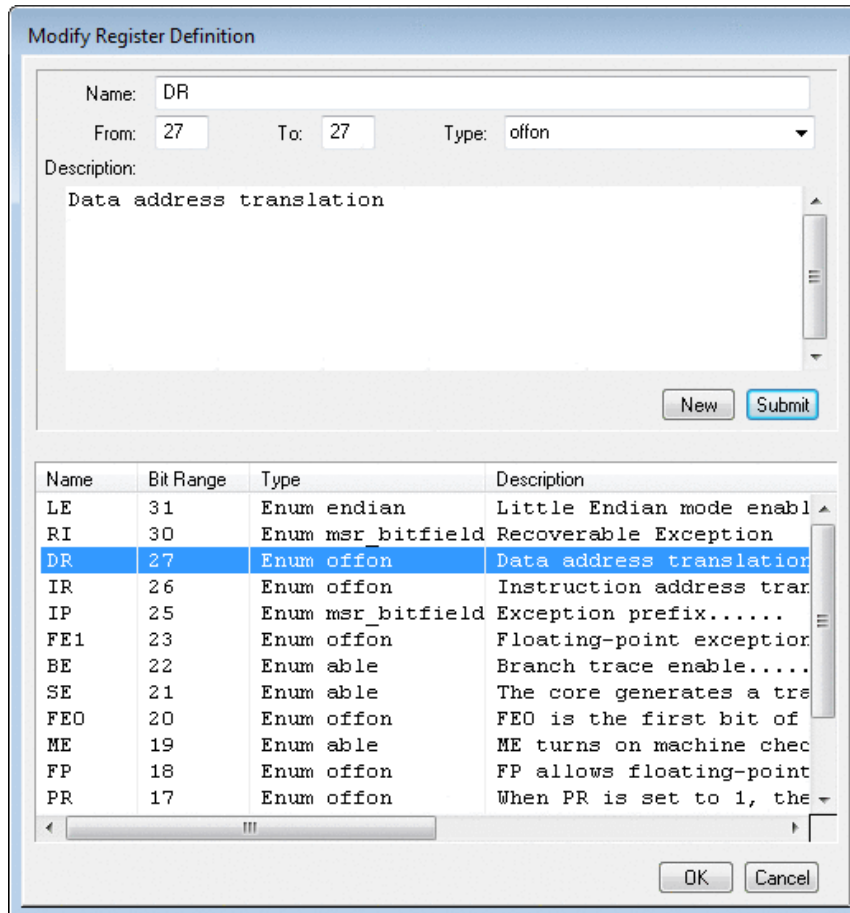
1. Select the text you want to change (the hex prefix in **Hex value** cannot be changed);
2. Type in the new value.

If an invalid character is typed, you may hear a beep.

- In the shortcut menus of the concise display pane and the detailed display pane, you can select the value entries to set the corresponding bitfield's value for enumeration bitfields, or select **Change Value** to type in values for non-enumeration bitfields wider than 2 bits.
- In the detailed display pane, you can double-click a listed enumeration value to set the corresponding bitfield to that value.
- In the concise display pane and the detailed display pane, you can double-click a bitfield with only one bit to toggle its value.

The Modify Register Definition Dialog

You can use the **Modify Register Definition** dialog (an example is shown below) to change a register's bitfield definition for the current debugging session.



The **Modify Register Definition** dialog can be launched by choosing **Modify Register Definition** from the shortcut menu of the **Register Information** window's concise display pane or detailed display pane.

If you modify an existing register's definition or create a new register, you will be asked to save these definitions into a GRD file when you quit MULTI. To save any new or modified register definitions to a GRD file, choose **File → Save Register Definitions Created on the Fly** from the **Register View** window. You can merge the changes into your default register definition system later or dynamically load the GRD file in future debugging session by choosing **File → Load Register Definitions from File** from the **Register View** window.

The **Modify Register Definition** dialog contains three parts:

- The bitfield editor pane, which contains fields for a bitfield's attributes and a set of buttons to manipulate the changes. For more information, see “Bitfield Editor Pane” on page 277.
- The bitfield list pane, which lists the information for all bitfields. For more information, see “Bitfield List Pane” on page 278.
- The button set, which allows you to commit (**OK**) or discard (**Cancel**) the changes made in the dialog.

Bitfield Editor Pane

The bitfield editor pane has the following attributes. Whenever you select an entry in the bitfield list pane, the settings for the selected bitfield will be displayed in the editor pane. If you click the **New** button, a template for a new bitfield will be displayed in the editor pane.

Item	Meaning
Name	The name of the bitfield. Bitfields within a register must have distinct names. For a new bitfield, MULTI will generate a unique name; you can change it to a more meaningful name.
Description	The description of the bitfield, which will be shown in the Register Information window.
From	The starting bit position (inclusive) of the bitfield. The bits are numbered starting with 0.
To	The ending bit position (inclusive) of the bitfield.
Type	The type of the bitfield: <code>hex</code> , <code>binary</code> , <code>signed</code> , <code>unsigned</code> , or an enumeration type defined in one of the GRD files that have been loaded.

The functions of the buttons in the bitfield editor pane are listed below:

Button	Meaning
New	Creates a new bitfield in the bitfield editor pane.
Submit	Propagates the changes shown in the bitfield editor pane to the bitfield list pane. If the bitfield shown in the bitfield editor pane is an existing bitfield in the bitfield list, the corresponding entry in the bitfield list will be changed; otherwise, the new bitfield will be added to the end of the bitfield list. In order to accept any changes made, you still have to click the OK button in the lower button set.

Bitfield List Pane

The bitfield list pane appears below the bitfield editor pane. Its columns correspond to the attributes of a bitfield, as described in “Bitfield Editor Pane” on page 277.

To display or change a bitfield's attributes, click a list entry to select it. The bitfield's attributes will be displayed in the bitfield editor pane, where they can be examined or changed.

The following choices appear in the right-click menu of the bitfield list:

Item	Meaning
Add	Creates a new bitfield in the bitfield editor pane.
Delete	Deletes the selected bitfields.
Modify	Displays the most recently selected bitfield in the bitfield editor pane where it can be examined or changed.
Move Up	Moves the selected bitfield or bitfields up one row.
Move Down	Moves the selected bitfield or bitfields down one row.
Sort by Bit Position in Ascending Order	Sorts the bitfields by their bit position in ascending order.
Sort by Bit Position in Descending Order	Sorts the bitfields by their bit position in descending order.

The following keyboard shortcuts can be used to manipulate the bitfield list:

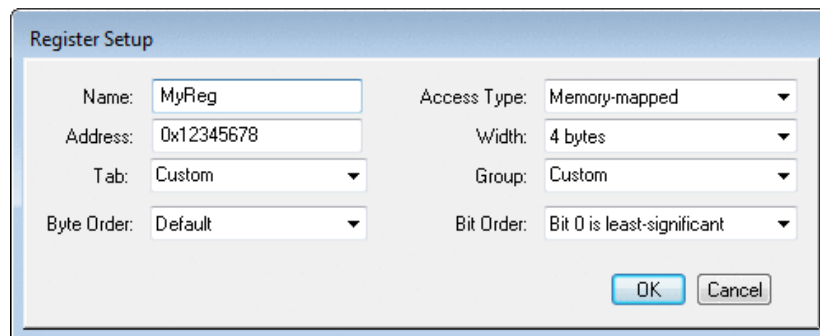
Key Action	Meaning
Ctrl+UpArrow	Moves the selected bitfields up one row.
Ctrl+DownArrow	Moves the selected bitfields down one row.
Ctrl+D	Deletes the selected bitfields.
Delete	Deletes the selected bitfields.

The Register Setup Dialog

You can create a new register in one of these ways:

- Choose **View** → **View “variable” as Register** from a Data Explorer.
- Run the **regadd** command in the Debugger command pane. For information about the **regadd** command, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Register Setup** dialog opens so that you can specify the basic information for the newly created register.



The **Register Setup** dialog contains the fields listed in the table below.

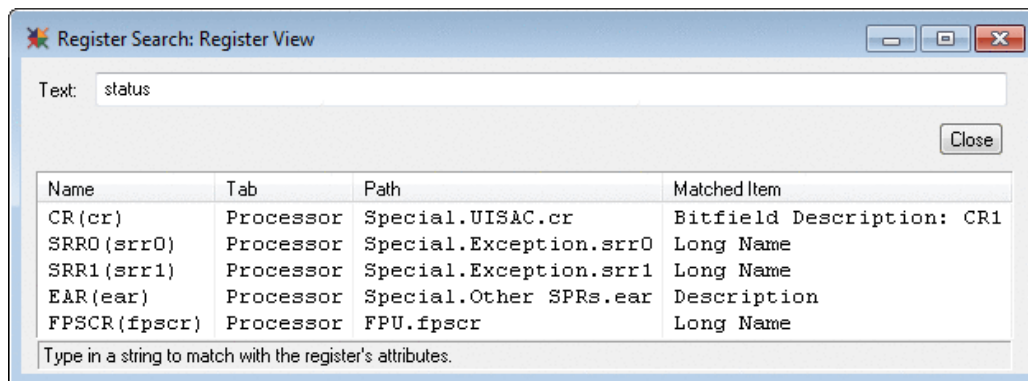
Field	Meaning
Name	Name of the register. This name must not already be used for an existing register.
Access Type	Access type of the register. The only available choice is <code>Memory-mapped</code> .
Address	The address at which the register value can be read.

Field	Meaning
Width	Access size used to read the register value.
Tab	Register View window tab on which the register will be displayed. Choose one of the tab names from the list.
Group	Group to which the register is to be associated. Choose one of the group names from the list.
Byte Order	Specifies the byte order of the register: • Default— Always the same as the target • Big Endian — Always big endian • Little Endian — Always little endian
Bit Order	Chooses whether bits will be numbered with bit 0 referring to the least significant bit, as on most CPU architectures, or with bit 0 referring to the most significant bit, as on Power Architecture.

When a new register is created, a new **Register View** window will be launched to show the register in the proper tab, and a **Register Information** window will be launched to show the register's detailed information.

The Register Search Window

To locate a register on a target with many registers, use the **Register Search** window (see an example below).



You can launch the **Register Search** window by either:

- Clicking the  button in the toolbar of the **Register View** window

- Choosing **View** → **Search Register** from the **Register View** window

The **Register Search** window contains three parts:

- **Text** field — Enter a string to be matched against the register's various attributes. After each typed character, MULTI searches the register database loaded from GRD files, and displays matching registers in the Register List. The register attributes searched include:
 - Register names, including name, short name, long name and alias
 - Register description
 - Register bitfield descriptions
- **Close** button — Closes the **Register Search** window.
- Register list — Displays registers matching the text in the **Text** field.

The register list's columns display the information in the following table.

Field	Meaning
Name	Register name.
Tab	Tab in which the register is displayed in the Register View window.
Path	The path of the register in the hierarchy shown in the Register View window. The tab name is at the beginning of the path.
Matched Item	The register attribute that matches the text entered in the Text field.

To locate a register in the **Register View** window, single-click the entry in the register list; to display a register in the **Register Information** window (if the register has bitfields defined) or in the Data Explorer (otherwise), double-click the corresponding entry in the Register List.

The following items appear in the right-click menu of the register list:

Item	Meaning
Show “register” in Register Explorer	Displays the register in the Register View window. The tab to which the register belongs is displayed and all group nodes in the register's path are expanded, if necessary.

Item	Meaning
Show Register Information for “register”	Shows the register in the Register Information window.
Show “register” in Data Explorer	Shows the register in the Data Explorer.

Using Debugger Commands to Work with Registers

Registers that are defined by Register Explorer can be accessed directly from the command line or used in command line expressions. To use the value in the register named *reg* on the command line, enter a dollar sign followed by its name (*\$reg*).

Assume that a 32-bit memory-mapped register *b* is defined and currently contains the value 5. You can display the register value as follows:

```
> $b
$b = 0x00000005
```

You can assign a new value to the register:

```
> $b=3+3
$b = 0x00000006
```

Bitfield registers and structured registers are accessed from the command line in the same way as C-style structs. Assuming that a bitfield register *y* with the following bitfield description in the GRD format is defined as follows:

```
# Bitfields of register y

bitfield {
    y_bitfield {
        "cache_state" {loc="0..1"}
        "reserved" {loc="2..31"}
    }
}
```

This register definition indicates that `y` has two bitfields, `cache_state` in the low two bits and `reserved` in the high 30 bits. Assume the initial value of `y` is `0x13`. If you print out the register in the command pane, it will be displayed as a C-style struct:

```
> $y
y = struct y {
    reserved = 0x00000001;
    cache_state = 0x00000003;
} y
```

You can display an individual field of a bitfield register. For example:

```
> $y.cache_state
cache_state = 0x00000003
```

You can assign to a field of a bitfield register. For example:

```
> $y.cache_state=2
cache_state = 0x00000002
```

In the above example, the `cache_state` bitfield will be modified without affecting the `reserved` bitfield:

```
> $y
struct y {
    reserved = 0x00000001;
    cache_state = 0x00000002;
} y
```

Registers that are pointers to structures are accessed like C-style pointers on the command line. Assume that a structure `frame` and a register `fp` are described in the register definition files, as follows:

```
# Structure frame definition

struct {
    frame {
        hi_word {type="short"}
        lo_word {type="short"}
    }
}
```

```
# some characteristics of register fp

register {
    fp {address=0; type="frame *"}
}
```

Assume that `fp` is currently pointing to a the memory address `0xbbbb` that contains `0x000a0007`. The following illustrates what you would see in the command pane when printing out the contents of `fp`.

```
> $fp
$fp : *fp *0xbbbb: struct frame {
    hi_word = 10;
    lo_word = 7;
}
```

The pointer was automatically dereferenced when the register contents were printed. To print out a single field, dereference the field:

```
> $fp->lo_word
lo_word = 7
```

To assign to one of the fields, dereference the field:

```
> $fp->hi_word=1 << 3
hi_word = 8
```

Registers that are pointers to raw types are accessed the same way as variables that are pointers to the corresponding type. When using a register's value as an argument to a `MULTI` command or a command line procedure call, use the same syntax as you would for printing out register values or assigning to registers from the command line.

Customizing Registers

Customizing Registers from the Command Line

You can modify existing register definitions and define new registers from the command line while you are debugging a program. For descriptions of the syntax and function of these commands, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

Any modifications to the register definitions made from the command line are active only until you reload the program or connect to a different target. For persistent modifications, you must use **.rc** files or customize the default register definition files, as described below.

Customizing Registers in Default .rc Files

Any register modifications that are present in a program's default **.rc** file are persistent across reloads and establishing target connections. Default **.rc** files are the best place to insert definitions of registers you want to consistently use while debugging a specific program. For more information about default **.rc** files, see Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

The **regappend** and **regload** commands can appear in default **.rc** files and will be applied whenever the program is reloaded or a connection to a new target is established. For information about the **regappend** and **regload** commands, see “Register Commands” in Chapter 14, “Register Command Reference” in the *MULTI: Debugging Command Reference* book.

Customizing Default Register Definition Files

When you install the Green Hills Compiler, some default register definition files are installed. These files define a set of registers and register groups for popular target processors and boards. You should not need to modify these files. You can customize the default set of files, however, to add new registers or register groups

that are relevant to your target environment. This section describes how the default files are located and when they are loaded.



Tip

If you are customizing registers for a single program only, you should probably use the default **.rc** file approach. See also “Customizing Registers in Default .rc Files” on page 285. The default register definition files should only be modified if you want definitions to be applied to all of the programs you debug.

Register Explorer Startup

When you open a Debugger, MULTI searches for the root GRD file, which is used to display the registers of the target hardware. If MULTI cannot find or successfully load the root GRD file, most debugging functionality will not be available.

The root GRD file is the first of the following four register files that MULTI locates. MULTI searches for the register files in the order listed. For information about the directories MULTI searches when trying to locate the register files, see “File Locations” on page 287.

1. **prog.grd** — *prog* is the name of the program you are debugging. **prog.grd**, if it exists, is a user-defined file.
2. **dbserv-dbtarget.grd** — *dbserv* is the debug server you are connected to, and *dbtarget* is your debug server target. **dbserv-dbtarget.grd**, if it exists, is a user-defined file.
3. **cpu.grd** — *cpu* is the acronym for your processor family. (For a list of possible *cpu* values, see the following table.) **cpu.grd** is included in your Green Hills Compiler installation. If you have not defined the register files listed above, **cpu.grd** is used as the root GRD file.
4. **dbserv.grd** — *dbserv* is the debug server you are connected to. **dbserv.grd**, if it exists, is a user-defined file.

Appropriate values for *cpu* (in ***cpu.grd***) are listed in the following table.

Processor Family	Value of <i>cpu</i> (in <i>cpu.grd</i>)
680x0/683xx/ColdFire	68
ARC	arc
ARM/Thumb	arm
Blackfin	bf
FirePath	fp
FR	fr20
Lexra	lexra
M-CORE	mc
MIPS/MIPS 16	mips
nCPU	ncp
NDR	ndr
Power Architecture	ppc
StarCore	sc
SH	sh
Sparc	sparc
TriCore	tri
V81x/V83x	810
V85x	850
x86/Pentium	386

File Locations

MULTI searches for the register files listed in “Register Explorer Startup” on page 286, one at a time, in the following five standard directory locations unless otherwise noted. When MULTI finds one of the register files, it discontinues the search and uses the located file as the root GRD file (see “Register Explorer Startup” on page 286). MULTI searches the directories in the order listed.

1. The directory containing the program being debugged.
2. The directory containing the MULTI executable (`multi.exe`).

3. The **registers** directory located in your personal configuration directory.
4. The **registers** directory located in the site-wide configuration directory.
5. The **registers** directory located in the Green Hills Compiler **defaults** directory.

For information about configuration directories, see Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.



Note

MULTI does not search for *prog.grd* in the **registers** directories.



Tip

If the root GRD file contains non-absolute include directives, MULTI attempts to resolve the locations of the directives by searching the directory where the root GRD file is located. If you defined the root GRD file and you want to include the Compiler installation's *cpu.grd* file in it, you can use the variable `${__TOOLS_DEFAULTS_DIR__}` when specifying the path to *cpu.grd*. For example, for a Power Architecture target, you could specify:

```
%include "${__TOOLS_DEFAULTS_DIR__}/registers/ppc.grd"
```

Overloading Default Register Descriptions

You should usually keep your customized register definitions in the default **.rc** file for your program. If you overload one of the default files, be careful. If you do not include the appropriate default register definition file for your target, you may not be able to access any standard registers.

Setting Defaults for an Individual Program

You can add custom register definitions for an individual program by using a register definition file named after your program. Suppose you have a program *prog* that is compiled for a Power Architecture target. You should add custom register definitions into a *prog.grd* register definition file in your configuration directory. This file's contents should be of the following form in GRD format.

```
# foo.grd
#
# ... description of this file ...

# Include standard Power Architecture register definitions

#include "${__TOOLS_DEFAULTS_DIR__}/registers/ppc.grd"

#
# add custom register descriptions here
#
```

Whenever you debug a program *prog*, MULTI will find ***prog.grd*** first and load it, applying the standard Power Architecture register definitions from **ppc.grd** and then the custom register definitions.

**Note**

Be sure you insert an appropriate `%include` for the target on which your process is running. Otherwise you may be unable to view the standard registers on your target.

Setting Defaults for Multiple Programs

You can provide custom register definitions that are applied whenever you debug any program compiled for a specific target. These universal modifications are useful for defining things like custom register groups that you always use. Changes that you put in one of the default register definition files for a particular target are applied when you debug any program compiled for that target.

To apply custom register definitions to all programs compiled for a specific target, follow these steps:

1. In your local configuration directory, create a directory named **registers** if it does not already exist. Your local configuration directory is:
 - Windows 7/Vista — ***user_dir*\AppData\Roaming\GHS**
 - Windows XP — ***user_dir*\Application Data\GHS**
 - Linux/Solaris — ***user_dir*/.ghs/**

2. In the **registers** directory, create a file with the name ***cpu.grd***, where *cpu* is the acronym for your processor family (for example, ***ppc.grd***). For a list of possible *cpu* values, see the table in “Register Explorer Startup” on page 286.
3. Use a text editor (such as the MULTI Editor) to edit the new ***cpu.grd*** file:

- a. To include MULTI's default register definition file, add:

```
%include "${__TOOLS_DEFAULTS_DIR__}/registers/cpu.grd"
```

to the top of the file. *cpu* is the acronym for your processor family. If you do not insert this `%include` directive, you may be unable to view the standard registers on your target.

- b. Add your custom register definitions to the end of the file. For example:

```
# include my custom defintions
#include "my_regs.grd"
```

Chapter 14

Using Expressions, Variables, and Procedure Calls

Contents

Evaluating Expressions	292
Viewing Variables	297
Variable Lifetimes	299
Examining Data	300
Wildcards	303
Procedure Calls	304
Macros	307
MULTI Special Variables and Operators	309
Syntax Checking	320

This chapter describes how you can type expressions into the Debugger command pane to examine and perform calculations with the values of variables in your program. It also describes how you can use MULTI variables to examine and modify the behavior of the Debugger.

Evaluating Expressions

To evaluate an expression, enter it into the Debugger's command pane. Expressions may contain:

- Symbols (for example, you can refer to variables in your program)
- Numerical constants
- String constants, which can contain the standard C language character escapes
- Math and assignment (for example, you can add values together and assign values to variables in your program)
- Procedure calls
- Macros

The Debugger calculates the value of the expression and displays it. If you want to watch the value of an expression and have it reevaluated as you step through your program, you should use a Data Explorer. See “The Data Explorer Window” on page 185.

Expression Scope

Expressions are evaluated in a specific context, which affects what variables can be meaningfully used in an expression. MULTI determines the scope of an expression based on the position of the blue current line pointer (➡) and the scope rules of the language in use.

For example, suppose you are stopped inside the function `main()`:

```
int main() {  
    int foo = 3;  
    printf("Hello World! foo = %d", foo);  
    return foo;  
}
```

If the current line pointer is located on a line above the function `main()` (that is, out of its scope), and you enter the following expression in the command pane:

```
> print foo
```

MULTI prints the output:

```
Unknown name "foo" in expression.
```

However, if the current line pointer is located inside the function `main()`, the same input:

```
> print foo
```

outputs:

```
foo = 3
```

For information about variable lifetimes, see “Variable Lifetimes” on page 299.

Expressing Source Addresses

In any expression, you may want to include the address associated with a particular source location. MULTI accepts any of the following methods of expressing source addresses:

<code>#line</code>	Address of the code at line number <i>line</i> in the current file.
<code>("foo.c" # proc # line)</code>	Address of line <i>line</i> for procedure <i>proc</i> in file foo.c .
<code>proc # line</code>	Address of line <i>line</i> for procedure <i>proc</i> .
<code>("foo.c" # proc ## label)</code>	Address of label <i>label</i> for procedure <i>proc</i> in file foo.c .
<code>proc ## label</code>	Address of label <i>label</i> for procedure <i>proc</i> .

Language-Independent Expressions

The Debugger always follows the operator definitions for the current language. For example, in C, the assignment operator is one equal sign (=) and the equality comparison is two equal signs (==). On the other hand, in some languages, (Ada, for example) colon equal (:=) is the assignment operator, and one equal sign (=) is the equality comparison. This can make definition of language-independent expressions difficult. To overcome this problem, the Debugger always recognizes colon equal (:=) as the assignment operator (in addition to the correct operator in the current language) and two equal signs (==) as the comparison operator. To ensure that expressions are language-independent, these operators should be used to implement features or capabilities (such as button definitions) that may remain in operation over several source languages.

If you are debugging multiple-language applications, use syntax appropriate to the language of the source file currently displayed in the Debugger.

Language Keywords

When the Debugger evaluates expressions, it understands the following keywords for the current source language.

Language	Keywords
C	char, const, double, enum, float, int, long, long long, q15, q31, short, signed, sizeof, struct, union, unsigned, void, volatile, __accum, __bigendian, __bytereverse, __fixed, __littleendian
C++	(All C Keywords), class, namespace, bool

Caveats for Expressions

Consider the following when constructing expressions:

- If the expression begins the same way as a Debugger command or begins with a number, put the expression in parentheses () or explicitly use the **print** or **call** command to distinguish it from the Debugger command. For example, to look at the value of the variable `c`, enter `(c)` or `print c`. To call a procedure named `c()`, enter `call c()`. If you just enter `c`, the Debugger will execute the **c** (continue) command. For information about the **print** command, see

Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **call** command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

- If a filename such as **class.cc** or **namespace.cc** is the same as a language keyword and is used in an expression or command, you must enclose the filename in quotation marks. For example, the **e foo.cc#2** command executes successfully but the **e class.cc#2** command does not. The second command must be **e "class.cc"#2**.
- Use **** followed by **Enter** to span multiple lines.
- Comments begin with **/ *** (forward slash + asterisk) and end with either a new line or ***/** (asterisk + forward slash).
- C++ style **(//)** end-of-line comments are also supported.
- If the process has not been started, the expression may only contain constants (global addresses may be considered constants, depending on your system). After the process has started, the expression may also contain in-scope variables and procedure calls.
- If the process is running, the expression may only contain constants and, if the system allows, global and static variables and their addresses.
- If the process has not been started and was linked against shared objects, expressions referring to procedures and variables located within these shared objects may not be allowed.
- There are restrictions on procedure calls and overloaded operator calls. For more information, see “Procedure Calls” on page 304.
- The MULTI Debugger allows you to run, halt, step and set breakpoints in C99 programs just as in regular C programs. However, the MULTI expression evaluator generally follows C89 rules, not C99 rules. Specifically, expressions whose meaning depends on any of the following are not guaranteed to be evaluated correctly by MULTI:
 - C99 complex types
 - C99 imaginary types
 - C99 **NAN** and **INFINITY** constants
 - Differences in the types of integer literals in C99

- MULTI may not be able to parse C++ expressions (such as casts) involving types that contain non-type template parameters.
- In C++, template types with multiple adjacent right angle brackets (>) may appear without intervening spaces, contrary to the C++ standard.
- C++ templated functions are not supported.
- In C++, the `.*` and `->*` pointer-to-member operators and values of pointer-to-member types are not supported.
- In C++, casts to reference types are not supported.
- In C++, the Debugger never calls destructors while evaluating an expression.
- In C++, you must include the `enum`, `struct`, or `union` keyword when referring to an enumeration, structure, or union.
- In C++, expressions containing fully qualified function names are treated as command line procedure calls (with one exception*). To work around this behavior, leave off the types when specifying function names. If more than one function with the given name exists—for example, `foo(int)` and `foo(char)`—a Browse window prompts you to pick one.

*Expressions containing fully qualified names are treated as expressions when used in conjunction with MULTI's **e** or **examine** command. For information about the **e** command, see Chapter 11, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **examine** command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

- In C++, referring directly to the name of an overloaded operator function is legal only within address expressions. See “Using Address Expressions in Debugger Commands” in Chapter 1, “Using Debugger Commands” in the *MULTI: Debugging Command Reference* book.
- Support for AltiVec vector data types is limited:
 - You cannot perform math operations on them.
 - You cannot cast them to or from non-vector types. You can, however, convert pointers to them. For example, you can convert `void *` to `vector float *`.

- Vector assignment is supported, but direct numerical assignment is not. For example:

```
vector unsigned int vector1;  
vector unsigned int vector2;  
  
vector1 = vector2; /* This will work in an expression. */  
vector1 = {1,2,3,4}; /* This will not work. */
```

- Expressions involving values of function pointer type are not supported on targets such as 64-bit Power Architecture where the ABI requires the use of function descriptors. The Debugger will attempt to detect and reject attempts to write a value of function pointer type to a target variable on these targets.
- Compiler intrinsics are not supported.
- Expressions involving the `long double` floating-point type may lose precision or otherwise yield incorrect values.
- Classes defined inside functions cannot be referenced in expressions.
- GNU built-in functions are not supported.
- Input of integer constants wider than 64 bits is not supported. Most mathematical and logical operators are not supported on values wider than 64 bits.

Viewing Variables

There are several different methods for viewing the value of a variable, including the following:

- Click the variable in the source pane to see information about the variable in the command pane. For more information, see “Viewing Information in the Command Pane” on page 172.
- Double-click the variable in the source pane.
- Use the **print** command or the **examine** command. For information about these commands, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.
- Enter the variable name in the command pane.
- Right-click the variable and select **View Value**.

You can use any of the formats listed in the following table to unambiguously refer to a specific variable. Note that an arbitrary variable called `fly` is used in these examples. Spaces before and after the `#` symbol are optional, but the pair of straight double quotes (`" "`) around a filename is required. Local variables need to be either static or within a procedure on the stack.

<code>fly</code>	Performs a scope search for the variable <code>fly</code> , starting at the blue arrow and proceeding outward. Local variables, local static variables, and parameters are checked first, then file static variables, global variables, and special variables.
<code>\$fly</code>	Searches the list of special variables for <code>fly</code> . See “MULTI Special Variables and Operators” on page 309.
<code>:fly</code> <code>::fly</code>	Searches for a global variable named <code>fly</code> .
<code>(stack_depth # fly)</code> <code>(stack_depth ## fly)</code>	Uses the <code>stack_depth</code> procedure on the call stack for the scope search. This is useful if you are debugging a recursive procedure and multiple instances are on the stack. You can then pick the instance and display the value of the variable for that instance. The procedure currently containing the program counter is at stack depth 0 (zero). See the <code>e</code> command in Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Parentheses are required for these cases.
<code>(stack_depth ## label # fly)</code>	Local variable <code>fly</code> in the lexical block at label <code>label</code> in procedure at stack depth <code>stack_depth</code> . Parentheses are required for this case.
<code>("foo.c" # proc ## label # fly)</code>	Local variable <code>fly</code> in the lexical block at label <code>label</code> in procedure <code>proc</code> in file <code>foo.c</code> . Parentheses are required for this case.
<code>proc ## label # fly</code>	Local variable <code>fly</code> for block at label <code>label</code> in procedure <code>proc</code> .
<code>("foo.c" # proc # fly)</code>	Local variable <code>fly</code> for procedure <code>proc</code> in file <code>foo.c</code> . Parentheses are required for this case.
<code>proc # fly</code>	Local variable <code>fly</code> for procedure <code>proc</code> .

<pre>("foo.c" # fly)</pre>
Static variable <code>fly</code> in file <code>foo.c</code> . Parentheses are required for this case.
<pre>.</pre> (This designator is a period.)
Represents the result of the latest expression.

Variable Lifetimes

The Green Hills compilers augment the location description (such as register number, stack offset, memory location) for user variables with lifetime information, which indicates when the value at the given location is valid.

When you use Debugger commands (for example, **print** or **view**) or Data Explorers to evaluate expressions, the messages listed in the following table may appear next to the value of the expression.

Dead
The value displayed may be invalid because the location used to store the value of this variable may have been reused by the compiler to store the value of a temporary (or another) variable.
Not Initialized
The value displayed may represent an uninitialized value.
Optimized Away
The variable was optimized away by the compiler and does not have any storage. No value will be displayed in conjunction with this message.
Part of Expression Dead
One or more of the values used in the displayed expression may be invalid. See above for more on what this means.
Part of Expression Not Initialized
One or more of the values used in the displayed expression may be uninitialized. See above for more on what this means.

For information about expression scope, see “Expression Scope” on page 292.

Examining Data

The following sections describe commands and shortcuts for examining data. In the Debugger source pane, you can examine most items by double-clicking them. See “The Data Explorer Window” on page 185.

Variables

Variable names are represented exactly the same way they are named in the program. The case sensitivity of the current source language is used.

To display the value of a variable in the Debugger command pane, do one of the following:

- Click the variable name in the source pane.
- Enter the variable name in the Debugger command pane. If the variable has the same name as a MULTI command, preface the variable with the **print** command (for example, `print e`) or enclose the variable in a pair of parentheses (for example, `(e)`). For information about the **print** command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Expression Formats

You can use an expression format to specify the output format of some commands. An example expression format follows:

print [/format] *expr*

An expression format is of the form:

[*count*] [*style*[*size*]]

where *count* is the number of times to apply the format style *style*, and *size* is the number of bytes to format. For example, `print /4d2 fly` prints, starting at `fly`, four 2-byte numbers in decimal. If not specified, *count* defaults to one, and *size* defaults to the size of the type printed.

In addition to a number, *size* can be specified as one of the following values:

- `b` — One byte (usually a character)
- `s` — Two bytes (usually a short)
- `l` (lowercase `L`) — Four bytes (usually an int)

These are appended to *style*. For example, `print /xb fly` prints a single character-sized hex value. Whenever a format contains a preset size (that is, character-sized in the case of `/b`, or int-sized in the case of `/o`), the size is dictated by the target (for example, a target-sized character).



Note

Size specifiers override any preset size from a format. For example, `print /Ob fly` prints a byte-sized octal, not an int-sized octal.

The following table lists the options for *style*.

Value	Effect
a	Prints the string <i>expr</i> as a string. By default, this prints to the first null character, but you can use the <i>size</i> value to force the printing of a given number of bytes, regardless of the occurrence of null characters. Note that <i>expr</i> should actually be a string; if it is a pointer to a string, you should use the <code>s</code> format style (see below). Same as <code>print /s &expr</code> .
A	Prints <i>expr</i> as an address, using the size of an address as its preset size.
b	Prints <i>expr</i> in decimal as the target's default character size.
B	Prints <i>expr</i> in binary.
c	Prints <i>expr</i> as an ASCII character.
C	
d	Prints <i>expr</i> in decimal. If capitalized, <i>expr</i> is printed as an int-sized value.
D	
e	Converts <i>expr</i> to the style <code>[-]d.ddde+dd</code> , where there is one digit before the radix character and the number of digits after it is equal to the precision specification given for <i>size</i> . If <i>size</i> is not present, then the size of a double on the current target is used.
E	

Value	Effect
f F	Converts <i>expr</i> to the decimal notation in the style <code>[-]ddd.ddd</code> , where the number of digits after the radix character is equal to the precision specification given for <i>size</i> . If <i>size</i> is not present, then the size of a double on the current target is used. If <i>size</i> is explicitly zero, then no digits or radix characters are printed.
g G	Prints <i>expr</i> in style <code>f</code> or <code>e</code> . The style chosen depends on the converted value. Style <code>e</code> is used only if the exponent resulting from the conversion is less than -4 or greater than the precision given by <i>size</i> . Trailing zeros are removed from the result. A radix character appears only if followed by a digit. This is the default for floats and doubles. If <i>size</i> is not present, then the size of a double on the current target is used.
i	Using the <i>expr</i> as an address, disassembles a machine instruction.
I	(Uppercase <code>i</code>) Using the <i>expr</i> as an address, disassembles a machine instruction. If the address maps evenly to a line number in the source, it prints the source line first. This allows you to see what the compiler generated for a line of source. Using the mixed source/assembly mode in the source pane is an easier way to view the same information.
n N	Prints <i>expr</i> using the “normal” format based on its type. If no format is specified, this is the default. If capitalized, <i>expr</i> is not dereferenced; otherwise, structures and pointers are dereferenced.
o O	Prints <i>expr</i> in octal. If capitalized, <i>expr</i> is printed as an int-sized value.
p	(Lowercase <code>p</code>) Prints the name of the procedure containing address <i>expr</i> , along with the filename and the source line or instruction that addresses maps. If <i>size</i> is <code>1</code> or <code>b</code> , only the procedure name will be printed. If <i>size</i> is <code>2</code> or <code>s</code> , only the filename and procedure name will be printed.
P	(Uppercase <code>p</code>) Prints the name and signature of the procedure containing address <i>expr</i> .
q	Prints the floating point number that has the same bit representation as the binary number given by <i>expr</i> . For example: <pre>> print /q 0x3ff199999999999a 1.100000</pre> If <i>expr</i> is larger than 4 bytes in size, a double is printed instead. See also <code>x</code> below.
r	Prints the bounds of a ranged type or variable of a ranged type, such as a C bitfield.
s	Prints a string using <i>expr</i> as a pointer to the beginning of the string. Same as <code>print /a *expr</code> .

Value	Effect
s	Prints a formatted dump of a structure. This is the default for items of type struct.
t	Prints the type of variable or procedure.
u U	Prints <i>expr</i> in unsigned decimal. If capitalized, <i>expr</i> is printed as an int-sized value.
v	Prints <i>expr</i> using the “normal” format based on its type. This is identical to n.
x X	Prints <i>expr</i> in hexadecimal. If capitalized, <i>expr</i> is printed as an int-sized value. If <i>expr</i> is a floating point number, prints the binary number that has the same bit representation as the floating point number given by <i>expr</i> . For example: <pre>> print /x 1.1 0x3ff19999_9999999a</pre> See also q above.
z	Prints <i>expr</i> as a set.



Note

For more information about expression formats and the various commands that use them to display data or expressions, see the commands **print**, **examine**, and **eval** in Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Language Dependencies

In C++, when the Debugger displays a class, it also displays the fields of all the parents of that class, including virtual parents, if that information is available. Static fields associated with a class are also displayed. Additionally, fields of the class or its parents that are stored by reference will be displayed as an address preceded by an at sign (@).

Wildcards

In some contexts, the Debugger supports the use of wildcards in procedure and file names (for example, with the **e** command and in the File Locator and Procedure Locator). A question mark (?) matches any single letter, while an asterisk (*) or an at sign (@) matches any number of letters. For example, the sequence ??* matches all names that are at least two characters long.

The following table lists several different formats for referring to procedures in C++. Text you substitute for the replaceable words (*italicized*) may contain wildcards.

<i>class::func</i>	Matches all members whose names match <i>func</i> of all classes whose names match <i>class</i> , regardless of their arguments.
<i>::func</i>	Matches all global or static functions whose names match <i>func</i> . Argument types may be supplied to restrict the match.
<i>func</i>	Matches all functions, whether class members or not, whose names match <i>func</i> .

Procedure Calls

Making procedure calls from the Debugger command pane requires your program to be linked with **libmulti.a**, a library supplied by Green Hills. For more information, see the documentation about enabling command line procedure calls in the *MULTI: Building Applications* book.

Expressions may contain procedure calls. For example:

```
fly = AddArgs(1, 2) * 3;
```

In C++, overloaded operators may be called provided that they are not inlined and that the left side of the expression is not contained completely within a register. Thus, the following expression:

```
complex(1,2) + complex(2,3)
```

is converted into the appropriate procedure calls. Constructors are called when appropriate, again provided that they are not inlined.

However, if `fly` were a small struct contained entirely within a register, the following expression:

```
fly += bee;
```

would fail.

You can make a command line procedure call to any non-inlined procedure in the program being debugged. For example, assume that the procedure `printf()` is referenced in the program, and thus the code for it is on the target. In this case you may enter:

```
printf("Hello, %s!\n", "world")
```

If the name of the procedure you are trying to call is the same as that of a MULTI command, the MULTI command is executed instead of the procedure. To specify that MULTI commands are ignored when you call a procedure from the command pane, enter the **call** command before the procedure. For example, suppose you want to make a command line procedure call to a procedure named `e` that takes an integer. Enter:

```
call e(1)
```

in the command pane. If you enter:

```
e (1)
```

MULTI executes the `e` command instead. For information about the **call** command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

To find out what procedures are available to be called, use one of the following commands to list the procedures in the program being debugged:

- **browse procs** (see the **browse** command in “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book)
- **lp *** (lowercase `l`, lowercase `p`, asterisk; see the **l** command in Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book)



Tip

To gain access to library routines that are not referenced anywhere in the program code, and thus are not linked into the program image, add a dummy reference to the program and recompile.

Caveats for Command Line Procedure Calls

- Any breakpoints encountered during command line procedure calls are handled as usual.
- In some cases, interrupts may need to be disabled before command line procedure calls will work.
- If MULTI detects an executing task, it prevents you from making command line procedure calls via a freeze-mode connection to an operating system kernel. You can bypass this restriction if you know it is safe to do so by issuing the command **configure AllowProcCallInOsaTask on** (see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book).
- If function prototype information is available, the Debugger checks the function prototype and converts each argument expression to the type of the corresponding argument. If it is not available, automatic promotion of arguments and detection of invalid arguments are not supported. In this case, you should ensure that function arguments specified are compatible with the function being called.
- For calls to procedures written in assembly language, the Debugger cannot perform argument conversions, check that the types of arguments are correct, or check that the number of arguments is correct.
- When evaluating an expression, the Debugger may call any compiled function, with or without arguments, including both application and operating system functions. However, an OS function on the target system is only called if it is already linked into your program. You are responsible for linking any system calls that are called from the command line into the program.
- C++ templated functions are not supported.
- In C++ or any language with inlined procedures, a procedure that is always inlined (so there is no stand-alone version of the procedure) may not be called.
- In C++, when the expression evaluator is unable to disambiguate overloaded procedure names, a dialog box prompts you to identify what function to use.
- In C++, default arguments are not inserted.
- In C++, the class member `operator()`, the function call operator, and the `new()` and `delete()` operators are not supported.

- In C++, any procedure or method whose return type has a copy constructor cannot be called from the command line.
- In C++, any procedure or method that takes a parameter having a type with a copy constructor cannot be called from the command line.
- If you are using a run-mode debug server such as **rtserv** or **rtserv2**, you may only make command line procedure calls on halted tasks that are actually runnable (as opposed to halted tasks that were pended before being halted). MULTI attempts to detect and prevent procedure calls to tasks that appear to have been halted while performing a system call (tasks are not runnable in this context). However, MULTI is not always able to detect situations in which it is unsafe to make command line procedure calls on a given run-mode task that is halted. If you are debugging an INTEGRITY run-mode target, see “Run-Mode Debugging” in the *INTEGRITY Development Guide* for more information.
- Command line procedure calls to functions that use the host I/O facilities of the Debugger to make blocking calls (for example, reading input from the user via the I/O pane) can cause the Debugger to appear unresponsive. In these cases, you can hit the **Esc** key to halt the blocked thread.
- You cannot step or run back into TimeMachine mode while a command line procedure call is in progress.
- Arguments of function pointer type are not supported on targets such as 64-bit Power Architecture where the ABI requires the use of function descriptors. The Debugger will attempt to detect and reject attempts to pass a value of function pointer type as an argument to a command line procedure call on these targets.

Macros

From the command pane, you can evaluate C/C++ macros defined in the program you are debugging if the macro is used somewhere in the program and if the program has been compiled with generation of debugging information enabled. See the *MULTI: Building Applications* book for information about generating debugging information.

The Debugger treats object-like macros differently than function-like macros. Examples of both types of macros follow.

```
#define OBJ_MACRO 0
```

is an object-like macro, whereas

```
#define FUNC_MACRO(x) (x*2)
```

is a function-like macro.

If an object-like macro and another symbol in your program share the same name, the symbol takes precedence over the macro if the symbol is local to the current function or file. If, however, a function-like macro and another symbol in your program share the same name, the macro always takes precedence over the symbol.

When you evaluate a macro, MULTI expands it regardless of your location in the program (even if you are currently viewing a file that is out of the scope of the macro definition). If multiple definitions of a macro exist across the program, the Debugger uses the definition local to the current file. If the current file does not contain a definition, MULTI uses a specific definition designated by the debug information.

The following table lists the most common methods of interaction with macros.

Action	Meaning
Click	- Object-like macros: displays the evaluated value of the macro, whatever that may be. In some instances, this may result in an error message similar to <code>Unknown name "foo" in expression</code>. This is because the Debugger is evaluating the macro in the current context, and a variable necessary to completely evaluate it is not defined within the current scope. - Function-like macros: displays the definition of the macro.
Click and Select	- Object-like macros: equivalent to a click. - Function-like macros: displays the evaluated value of the macro, if the arguments are selected as well. If they are not, this is equivalent to a click. The caveats that apply to clicking an object-like macro (see above) apply to clicking and selecting a function-like macro and its arguments.
Right-click	Opens a shortcut menu. See “The C/C++ Macro Shortcut Menu” on page 711.

Action	Meaning
In a standard expression (that is, <i>not</i> an address expression)	Expands the macro with its evaluation in the current scope, as click above (or click and select for function-like macros) does. Expression evaluation is aborted entirely if any of the variables used by the macro are not defined in the current scope.

MULTI Special Variables and Operators

The Debugger maintains a list of special variables and operators that are not a part of your program, but can be used in the Debugger as if they were. For example, you can use a special variable or operator in an expression that you evaluate in the Debugger.

The MULTI special variables and operators include user-defined variables (such as `$foo`), pre-defined system variables (such as `_DATA`), processor registers (such as `$r1`), and special operators (such as `$bp_adr(%bp_label)`).

When the Debugger is evaluating an expression and it finds a variable name such as `foo`, it first performs a scope search in the program to see if the variable exists. If the variable does not exist, then the list of special variables and operators is searched. Variable names beginning with a dollar sign (\$), such as `$foo`, are assumed to be special variables or operators.

User-Defined Variables

You can define a new variable by entering an appropriate statement into the Debugger command pane. The statement must adhere to the following format:

\$variable_name [=*expression*]

where:

- *variable_name* should not be a reserved keyword. Reserved keywords include commands and system variables.
- The value of the optional argument *expression* initializes the variable. If you do not specify *expression*, the initial value of the variable is zero (0).

User-defined variables are of the same type as the last expression assigned. For example, entering:

- `$foo=1`

creates the special variable `$foo`, assigns it the value 1, and makes its type `int`.

- `$bar=3*4`

creates the special variable `$bar`, assigns it the value 12, and makes its type `int`.

- `$baz=myInstance.a`

creates the special variable `$baz`, assigns it the value of `myInstance.a`, and makes it the same type as `myInstance.a`.

User-defined variables are just like any other, except that it is meaningless to evaluate their addresses.

System Variables

The following table lists the currently defined system variables. Modifying the values of these variables changes the way the Debugger operates. To display the value of a system variable, enter it in the Debugger command pane with a dollar sign prepended to it (for example, `$ANSICMODE`).

ANSICMODE
When set to 0 (zero), expressions are evaluated as they are in K+R C. When set to 1, which is the default value, expressions are evaluated as in ANSI C. Generally, this affects how <code>unsigned shorts</code> , <code>unsigned chars</code> , and <code>unsigned bitfields</code> are coerced. By default in K+R, they are coerced to <code>unsigned int</code> , whereas in ANSI they are coerced to <code>int</code> . Thus <code>((unsigned short) 3) / -3</code> yields different results in ANSI and K+R. The type of <code>sizeof</code> is different, as is the interpretation of the <code>op=</code> operators in certain obscure cases.
CONTINUECOUNT
If this is 0 (zero) or 1, the Debugger will stop at the next breakpoint. If it is 2, the Debugger will stop at the second breakpoint reached by the process, and so on. Use the <code>c</code> command to set its value. For example, to set it to 3, enter: <code>c @3</code> . For information about the <code>c</code> command, see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

DEBUGSHARED
Enables/disables debugging of shared objects. This system variable is only relevant with targets that support shared libraries, such as certain native Linux/Solaris platforms or advanced embedded real-time operating systems.
DEREFPOINTER
Controls whether or not pointers are automatically dereferenced when displayed by the print , examine , and view commands. For information about the print and examine commands, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. For information about the view command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
DISNAMELEN
Controls the length of the label printed when labels appear in certain cases in disassembly. For example, the disassembly of a call or branch may include the target label; DISNAMELEN controls how many characters of that constant to print.
FASTSTEP
When set to non-zero (the default), the Debugger attempts to speed up source code stepping. On each source step, the Debugger analyzes the code, sets temporary breakpoints on all possible step destinations, and allows the process to run until one of the breakpoints is hit. This allows the process to run at nearly full speed during the source step. When set to 0 (zero), the Debugger usually steps one machine instruction at a time until it reaches the next source line. However, it still sets temporary breakpoints and runs to step over function calls. This method is generally much slower. This system variable corresponds to Debug → Debug Settings → Fast Source Step .
SERVERTIMEOUT
How long (in seconds) MULTI will wait for a debug server to respond before concluding that the server has failed. MULTI will prompt the user to close the connection or keep waiting.
TASKWIND
If this is 0 (zero), the Task Manager (for multitasking targets) will be disabled.
VERIFYHALT
Verifies halting a process before setting a breakpoint by bringing up a confirmation dialog.

System variables beginning with an underscore (`_`), such as those shown in the following table, represent the internal state of the Debugger and are not normally listed. To see them, use the command `!s _` (lowercase `l`, lowercase `s`, underscore). For information about the `!s` command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

<code>_ASMCACHE</code> When set to 1, which is the default value, the disassembly of program code in the Debugger window is done by reading data from the executable file, not from the program being debugged. This allows a faster disassembly display to the screen. Setting <code>_ASMCACHE</code> to 0 (zero) forces the Debugger to read the text to be disassembled from the program being debugged, instead of from the buffer or executable file. If instruction memory is modified or destroyed and <code>_ASMCACHE</code> is 1, then displays of disassembled instructions will continue to show the original, unmodified instructions in the executable file. This is confusing because the instructions actually executed are not those shown by the disassembly display. Sometimes, when a peculiar behavior occurs on the target system, such as when the process stops on an apparently valid instruction or when it refuses to single-step or continue past a valid instruction, then the instruction memory on the target system has been corrupted. Try setting <code>_ASMCACHE</code> to 0 (zero) and redisplaying the assembly code. You may find invalid instructions at the point of failure. (You may need to turn off <code>_INTERLACE</code> (assembly) mode and examine another part of the program, turn <code>_INTERLACE</code> mode back on, and then return to the point of the entry to clear out the Debugger's internal disassembly cache.)
<code>_AUTO_CHECK_COHERENCY</code> If nonzero, automatic coherency checking is enabled. If zero, automatic coherency checking is disabled. For more information, see “Detecting Coherency Errors” on page 536. See also the <code>_AUTO_CHECK_NUM_ADDRS</code> variable below.
<code>_AUTO_CHECK_NUM_ADDRS</code> Sets the number of addresses to check for coherency. Because extra memory is read on every stop, use care when setting the number of addresses to be read. Setting a large number of addresses may slow normal run control. By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer). The value of this variable is only meaningful if automatic coherency checking is enabled. See the <code>_AUTO_CHECK_COHERENCY</code> variable above. For more information, see “Detecting Coherency Errors” on page 536.

_CACHE

If non-zero, the Debugger uses a cache for reading memory from the target. The cache is invalidated every time the process state changes. This speeds up embedded debugging. Because certain devices such as hardware registers and control ports must be accessed using a specific memory access size, you must disable the **_CACHE** variable before viewing memory that corresponds to these devices.

For related commands, see the **memread** and **memwrite** commands in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

_DATA

Used only for position-independent data (PID) systems where the executable is linked as if it were at one address, while it runs at another. This variable is set to the offset between the location at which the data segment resides and at which it is linked. It should not be set to -1. This is set on the command line with the **-data** option.

_DISPMODE

Determines whether assembly code is interlaced with source code in the source pane. See “The Command Pane Shortcut Menu” on page 712. This variable has no effect in **Assem** source pane display mode.

_INIT_SP

Tells the Debugger the value of the stack pointer at program start up, in certain target environments where this information is not available.

_LANGUAGE

Shows which language-specific rules for the expression evaluator are in use. 0 (zero) means C, 3 means C++, 7 means Assembly, and 31 [default] means auto-select based on the type of current file. You should not usually need to change this system variable from its default value.

_LINES

This controls the number of lines displayed by the **printwindow** command, and in non-GUI mode, also controls the number of lines shown between **More?** prompts. For information about the **printwindow** command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

_OPCODE

If non-zero, then disassembly mode displays the hexadecimal value of the instruction.

_TEXT

Used only for position-independent code (PIC) systems where the executable is linked as if it were at one address, while it runs at another. This variable is set to the offset between the location at which the text segment resides and links. It should not be set to -1. This variable is set on the command line with the **-text** option.

The following system variables are read-only.

**Note**

Supported values for the system variables whose names begin with `_TARGET` are defined in the `os_constants_internal.grd` file located at `compiler_install_dir/defaults/registers`.

<code>_BREAK</code>	The current breakpoint number.
<code>_CURRENT_TASKID</code>	The task ID of the currently executing OSA task.
<code>_ENTRYPOINT</code>	The address of the entry point (for example, <code>_start</code>) for the current thread, if any. Note that this may differ from the entry point of user code (for example, <code>main</code>). <code>_ENTRYPOINT</code> is not adjusted for the PIC base address of PIC programs.
<code>_EXEC_NAME</code>	The name and path of the program currently loaded in the Debugger.
<code>_FILE</code>	The name of the current file.
<code>_INTERLACE</code>	Indicates whether assembly code is displayed in the source window. If there is assembly code currently displayed in the source window, then this is 1; otherwise it is 0 (zero).
<code>_LAST_COMMAND_STATUS</code>	The status of the last MULTI command execution (0 means failure, 1 means success).
<code>_LAST_CONNECT_CMD_LINE</code>	The argument(s) last run with the MULTI connect command. For information about the connect command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
<code>_LINE</code>	The current line number.
<code>_MULTI_DIR</code>	The name of the directory that contains the MULTI executable.
<code>_MULTI_MAJOR_VERSION</code>	MULTI's major version (for example, 1 in MULTI 1.2.3).

<code>_MULTI_MICRO_VERSION</code>	MULTI's micro version (for example, 3 in MULTI 1.2.3).
<code>_MULTI_MINOR_VERSION</code>	MULTI's minor version (for example, 2 in MULTI 1.2.3).
<code>_NONGUIMODE</code>	This indicates whether or not MULTI was started in non-GUI (-nodisplay) mode.
<code>_OS_DIR</code>	The full path to the directory containing the Green Hills RTOS installation used to build the program currently loaded in the Debugger. <code>_OS_DIR</code> contains an empty string if the RTOS installation directory cannot be determined.
<code>_PID</code>	The identification number (ID) of the process, as reported by the debug server.
<code>_PROCEDURE</code>	The name of the current procedure.
<code>_PROCESS</code>	MULTI's internal slot number for the current process.
<code>_REMOTE_CONNECTED</code>	This is 1 if a remote target connection has been established, 0 (zero) otherwise.
<code>_RTSERV_VER</code>	The version number of rtserv (2 indicates rtserv2 , 1 indicates rtserv , and 0 indicates a non- rtserv connection).
<code>_SELECTION</code>	A string variable representing the current selection from the source pane.
<code>_SETUP_SCRIPT</code>	The full path to the file containing the setup script for the current connection. If the current process is not connected or the current connection does not have a setup script, <code>_SETUP_SCRIPT</code> contains an empty string.
<code>_SETUP_SCRIPT_DIR</code>	The full path to the directory containing the setup script for the current connection. If the current process is not connected or the current connection does not have a setup script, <code>_SETUP_SCRIPT_DIR</code> contains an empty string.

<code>_STATE</code> The process state, where: • 1 = No child • 2 = Stopped • 3 = Running • 4 = Dying • 5 = Just forked • 6 = Just exec'ed • 7 = About to resume • 8 = Zombied
<code>_TARGET</code> Contains a unique identifier indicating the target processor's family and variant.
<code>_TARGET_COPROCESSOR</code> Contains a unique identifier indicating the target coprocessor's variant.
<code>_TARGET_FAMILY</code> The family of the target on which the program being debugged is running.
<code>_TARGET_IS_BIGENDIAN</code> Indicates whether the target is big endian (1) or little endian (0).
<code>_TARGET_OS</code> Displays an identifier indicating which OS is running on the target, if such information is available.
<code>_TARGET_MINOR_OS</code> Displays an identifier indicating which minor type of OS is running on the target, if such information is available.
<code>_TARGET_SERIES</code> This may contain information about whether the target processor is a member of a certain series of processors (for example, the PowerPC 400 series). This does not work for all processor families.
<code>_TASK_EXIT_CODE</code> The exit code of the current task.

`_TOP_PROJECT`

The full path to the Top Project used to build the program currently loaded in the Debugger. The setting of this variable reflects the path that was correct at the time the program was built; if you move the project, you must rebuild it and reload the program to update this variable. This variable contains an empty string if the name of the project file cannot be determined.

`_TOP_PROJECT_DIR`

The full path to the directory containing the Top Project used to build the program currently loaded in the Debugger. The setting of this variable reflects the path that was correct at the time the program was built; if you move the project, you must rebuild it and reload the program to update this variable. This variable contains an empty string if the name of the project file cannot be determined.

Processor-Specific Variables

Processor registers are included as predefined variables. To find out what register names are available on your system, use the command `l r` (lowercase `L`, lowercase `R`) to list the registers. For information about the `l` command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

All registers can be treated as integers of the correct size for the register. Be careful when you modify the contents of registers when you are debugging high-level code; the results of these modifications can often produce unpredictable effects.

The following predefined Debugger internal variable is also included.

`$result`

Holds the return value of the most recently completed procedure. This variable is an alias for the register on the processor architecture used for returning integers. On most systems, this is also the register to return pointers. It may be written to as well as read from.

This value is not guaranteed to be correct; it depends on the return type of the function and the processor architecture. The actual return variable may be located elsewhere. As with any register, you must be careful when you change its value.

Special Operators

The Debugger maintains a list of special operators that are not a part of your program, but can be used in the Debugger as if they were. For example, you can use special operators in expressions that you evaluate in the Debugger.

<code>\$bp_adr(%bp_label)</code>
Returns the address of the breakpoint with label <i>bp_label</i> .
<code>\$entadr(procedure)</code>
Returns the address of the first instruction after the given procedure <i>procedure</i> 's prologue code. (The procedure prologue code is also known as the stack frame setup code or the entry code.)
If <i>procedure</i> is not given, the current procedure is used.
<code>\$exists("symbol")</code> <code>\$M_sym_exists("symbol")</code>
Returns a Boolean indicating whether the symbol <i>symbol</i> exists within the program currently being debugged.
<code>\$in(procedure)</code>
Returns true if the current process is stopped and the current program counter (PC) is in the given procedure <i>procedure</i> .
<code>\$M_called_from(level, "string")</code>
Returns a Boolean indicating true if the <i>level</i> is 0, and " <i>string</i> " is the name of a function on the current call stack. Otherwise, it returns true if " <i>string</i> " is the name of the function at the call stack depth indicated by <i>level</i> .
<code>\$M_can_read_address(num)</code>
Returns true if the address indicated by <i>num</i> is readable; returns false otherwise.
<code>\$M_file_exists("file")</code>
Returns a Boolean indicating whether the file <i>file</i> exists within the program currently being debugged.
<code>\$M_get_temp_memory(size)</code>
Returns an address that points at a chunk of memory of <i>size</i> address units. If <i>size</i> is too large, an error message is printed and -1 is returned as the address. This chunk of memory is guaranteed to be valid only until MULTI needs more temporary memory on the target. In practice, it should remain valid until the next time the user stores a string or other data structure to the target (by calling a constructor at the command pane, for example).
Using this special operator requires that your program be linked with libmulti.a , a library supplied by Green Hills. For more information, see the documentation about enabling command line procedure calls in the <i>MULTI: Building Applications</i> book.

<code>\$M_offsetof(<i>type</i>, <i>field</i>)</code> <code>offsetof(<i>type</i>, <i>field</i>)</code>
Returns the offset in bytes of the member field <i>field</i> within the aggregate type <i>type</i> . The first parameter must be either an aggregate (struct, union, or class) type name or an instance of an aggregate type. The second parameter must be the name of a field within that type. Note that entering <code>offsetof(<i>type</i>, <i>field</i>)</code> in the command pane may execute a macro or procedure call if the current program has a macro or procedure call with the name <code>offsetof</code> . To guarantee the use of the Debugger's built-in operator, use <code>\$M_offsetof()</code> .
<code>\$M_sec_begin("section")</code>
Returns the address of the beginning of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_sec_end("section")</code>
Returns the address of the end of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_sec_exists("section")</code>
Returns a Boolean indicating if the section <i>section</i> exists within the program currently being debugged.
<code>\$M_sec_size("section")</code>
Returns the size of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_strcmp(<i>expr</i>, "string")</code>
Returns an integer greater than, equal to, or less than zero if the string pointed to by the first parameter is lexicographically greater than, equal to, or less than the string pointed to by the second parameter. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.
<code>\$M_strcpy(<i>expr</i>, "string")</code>
Returns the number of bytes written. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.
<code>\$M_strprefix(<i>expr</i>, "string")</code> <code>\$M_strcaseprefix(<i>expr</i>, "string")</code>
Returns a Boolean indicating whether the first parameter is prefixed by the second (<code>\$M_strcaseprefix</code> is case-insensitive). The first parameter must be an expression that evaluates to a string and the second parameter must be a string constant.
<code>\$M_strstr(<i>expr</i>, "string")</code>
Returns a pointer to the first occurrence of the second parameter string within the first parameter string, or a null pointer if it is not found. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.

```
$M_typeof(var)
```

Returns the type of *var*, suitable for use in casting. For example:

```
$foo = (($M_typeof(my_var)) 0x10000)
```

would give the value of 0x10000 to the MULTI local variable `$foo` and give it the same type as `my_var`.

```
$M_var_is_valid(var)
```

Returns true if and only if the indicated variable exists in the current scope, is initialized, and is not yet out of scope (that is, dead). Returns false otherwise.

```
$retadr(procedure)
```

Returns the address of the first instruction of the given procedure *procedure*'s epilogue code. (The procedure epilogue code is also known as the stack frame release code or the return code.)

If *procedure* is not given, the current procedure is used.

```
sizeof(variable)
```

```
sizeof(type)
```

```
sizeof("string")
```

```
sizeof((expression))
```

Behaves similarly to the C `sizeof` operator and returns the size in bytes of the type of *variable*, of *type*, or of the type resulting from *expression*. If "string" is specified, this operator returns `strlen("string") + 1`. Unlike the C `sizeof` operator, this operator actually evaluates any argument provided, such that if an expression with side effects is specified, the side effects occur.

Syntax Checking

The syntax checking mechanism checks the validity of a command without actually executing it, and thus without requiring target interactions and without changing the system settings.

The Debugger command `sc` performs syntax checking. It can be used in two different ways.

To check the syntax of a single command, enter:

```
sc "command"
```

To check the syntax of an entire script file and all nested files, enter:

```
sc < script_file_name
```

For more information, see the **sc** command in “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

Syntax checking is also automatically invoked whenever a breakpoint with an associated command or condition is created. The validity of the commands associated with the breakpoint is checked in the context that would exist if the breakpoint were hit. If a syntax error is found in the breakpoint command, an error message is issued and the breakpoint is not created.



Note

You can use the **bpSyntaxChecking** configuration option to disable this automatic syntax checking. For more information, see the **bpSyntaxChecking** option in “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

For example, entering the command:

```
sc "print abcdef"
```

will echo the error message:

```
Syntax Checking: Unknown name "abcdef" in expression.
```

and entering the command:

```
b main { print abcdef; }
```

will echo the error messages:

```
Syntax Checking: Unknown name "abcdef" in expression.
```

```
Failed to set software breakpoint owing to syntax error.
```


Chapter 15

Using the Memory View Window

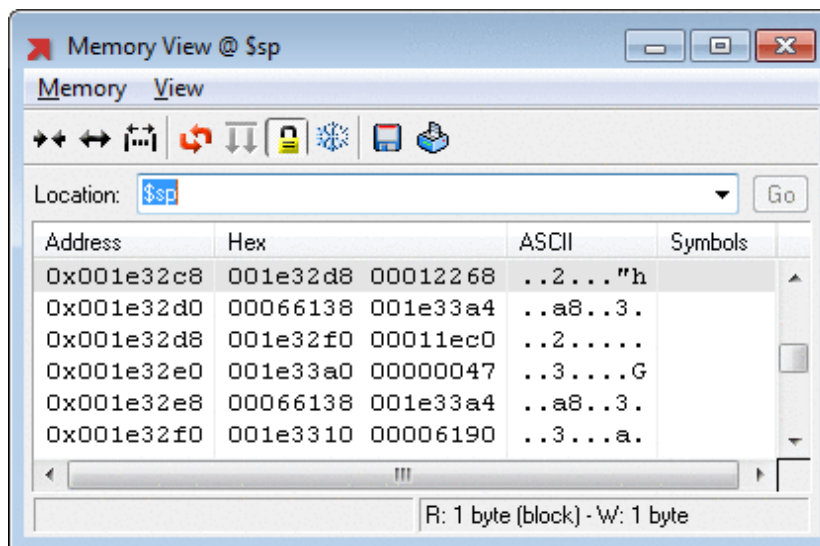
Contents

Setting the Active Location	325
The Memory Pane	326
Controlling Memory Access	328
Editing Memory	329
The Memory View Toolbar	330
Memory View Menus	332

The **Memory View** window is useful for examining large buffers, strings, and other contiguous blocks of memory. The window can be configured to display memory in a variety of formats. Memory may also be modified from this window.

To open the **Memory View** window, perform one of the following actions:

- Click the **Memory** button (M) located on the toolbar.
- Select **View** → **Memory**.
- In a Data Explorer, select **Tools** → **Memory View on “variable”** to open a **Memory View** window examining the same memory location as the Data Explorer.
- In the command pane, enter the **memview** command. For information about this command, see Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Setting the Active Location

The active location for the **Memory View** window is displayed in the title bar and in the **Location** bar at the top of the window, and the row containing this location is highlighted in the memory pane.

When you specify a new active location, target memory is read and the displayed memory begins with the address of the active location. To change the active location, do one of the following:

- Enter an address expression, a section name, or a section name and offset (`.text+0x400`, for example) in the location bar, and press **Enter**.
- Select a previously entered location from the location bar's drop-down list.



Note

Scrolling the window does not change the highlighted location or the contents of the **Location** bar. To return to the active location, press **Enter**.

Locking the Current Symbol's Location

By default, if you enter a symbol as the active location, MULTI sets the memory pane location to the address for the symbol at the time you entered it. If the address for the symbol changes, the text in the location bar turns red, but the address highlighted in the memory pane does not change. To display and change the active location to the new address, press **Enter**.

If you would like the **Memory View** window to track the location of the symbol each time it changes, click the **Lock Address** button () so that it no longer appears pushed down. In this mode, the active location is updated every time the address for the symbol changes.

The Memory Pane

The memory pane is composed of rows of memory displayed in an address column and multiple view columns. To change the number of bytes displayed in each row, use the following buttons:

- **Contract** () and **Expand** () — Change the byte-width of the rows by a factor of two each time they are pressed.
- **Set Row Width** () — Opens a window in which you can specify the width of the rows.

The **Address** column displays the location of the first byte of each row and can never be removed from the pane. You can view the rows of memory in ascending or descending order. To change the order, click the header of the **Address** column. Each view column displays the formatted contents of memory at that location.

Formatting View Columns

You can independently format each view column in the memory pane. To format a view column, do one of the following:

- Right-click the header of a column and select a formatting option from the shortcut menu.
- Select **View** → **Column *number name*** and select a formatting option from the submenu.

The primary formatting options are **Hex**, **Decimal**, **Binary**, **Floating Point**, **Fixed Point**, **Ascii**, **Symbols**, **Disassembly**, and **Offset**. You may see additional view column formats if your CPU supports them.

In addition to primary formatting options, secondary formatting options may appear if the selected primary format supports them. For the comprehensive list of options, see the “The Column Submenu” on page 334.

The following information is specific to select formats.

- Numerical formats — Any memory that cannot be read is displayed as a series of dashes.

- **Ascii** format — Any memory that cannot be represented by an ASCII character is displayed as a dot (.).
- **Symbols** format — Symbols that continue from previous addresses are represented with the string “ . . . ”.
- **Disassembly** format — In regions of memory without symbol information, inaccurate instructions may be displayed if the beginning of the window is not aligned with the start of an instruction.
- **Big Endian** and **Little Endian** format — If the selected byte order is not the default order for your target, it is displayed in the column header.

Managing View Columns

The following table summarizes the actions you can perform on view columns:

Action	Procedure
Add Column	One of: • Select View → Add New Column → <i>format</i>. • Right-click the header beyond the right edge of the last column and select a format from the shortcut menu. • Right-click the header of a column and select Add New Column from the shortcut menu.
Remove Column	One of: • Right-click the header of a column and select Remove from the shortcut menu. • Select View → Column <i>number name</i> → Remove.
Hide Column	One of: • Right-click the header of a column and select Hide from the shortcut menu. • Select View → Column <i>number name</i> → Hide.
Show Column	Select View → Column <i>number name</i> → Show .

Action	Procedure
Resize Column Automatically	One of: • Right-click the header of a column and enable Size to Fit from the shortcut menu. • Enable View → Column <i>number name</i> → Size to Fit.
Reorder Columns	Drag the header of a column to its new location.

Controlling Memory Access

You can control how the **Memory View** window accesses memory by freezing the window, setting access sizes, or disabling block reads. The following sections describe each of these actions.

Freezing the Memory View Window

The contents of the memory pane are automatically updated each time the target halts or steps. To prevent this update, click the **Freeze** button (). Location changes and automatic updates are disabled until the window is unfrozen. To unfreeze the window, click the **Freeze** button again, and the contents of the window are updated. To force the window to update its contents, even if frozen, by immediately re-reading target memory, do one of the following:

- Click the **Reload** button (.
- Right-click the memory pane and select **Reload from Target**.
- Select **Memory** → **Reload from Target**.

Setting Access Sizes

The preferred read and write access sizes are shown in the *access size display area*, located in the right side of the status bar. The **Memory View** window defaults to these sizes when accessing target memory. To change the access sizes, do one of the following:

- Double-click the access size display area. In the **Access Sizes** dialog box that appears, select the new access sizes from the drop-down lists.

- Right-click the access size display area, select **Read Access Size** or **Write Access Size**, and select the new access size from the submenu.
- Select **Memory** → **Set Access Sizes**. In the **Access Sizes** dialog box that appears, select the new access sizes from the drop-down lists.

The contents of the memory pane are automatically refreshed to reflect your changes.



Note

Access sizes do not affect how memory is displayed in the view columns.

Disabling Block Reads

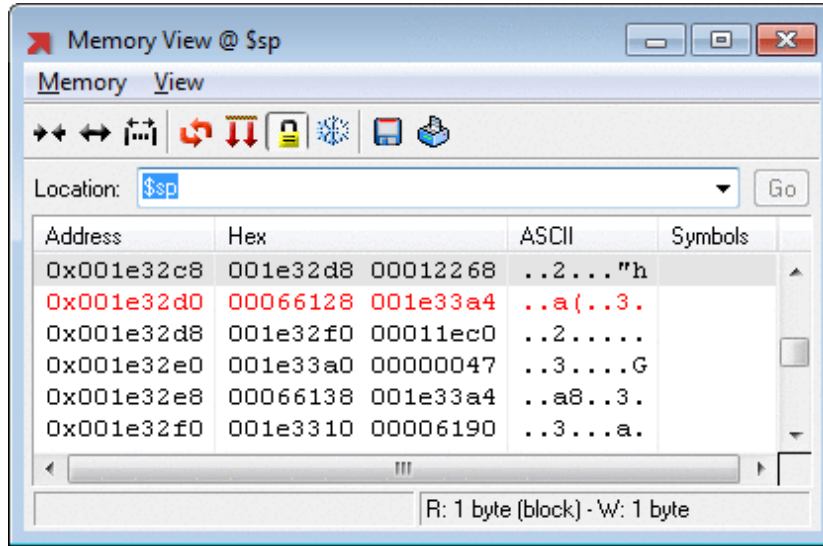
To increase performance, the **Memory View** window performs all target memory reads as block memory accesses. If you would like to set an access size because of target or memory requirements, but your debug server prevents you from doing so, disable the **Use Block Reads** option. The **Memory View** window then performs all reads as individual memory accesses of the preferred read size. To disable the **Use Block Reads** option, do one of the following:

- Clear **Memory** → **Use Block Reads**.
- Right-click the access size display area, and clear **Use Block Reads**.

Editing Memory

You can edit the contents of memory in the **Hex**, **Decimal**, **Binary**, and **ASCII** columns. To edit the contents of memory, perform the following steps:

1. Click the character you want to edit.
2. Type a new value. All columns update to reflect your change, which is displayed in red lettering to indicate that it has not yet been written to the target.



3. Repeat steps 1 and 2 for all of the characters you would like to modify.
4. Do one of the following:
 - Click .
 - Right-click the memory pane and select **Write Changes to Target**.
 - Select **Memory** → **Write Changes to Target**.

Until you perform this step, your changes will not be written to target memory.

The Memory View Toolbar

The **Memory View** toolbar contains buttons that allow you to control the **Memory View** window and perform memory-specific operations. The following table describes the buttons in the **Memory View** window.

Button	Effect
 Contract	Contracts the byte-width of the rows by a factor of two each time it is pressed.
 Expand	Expands the byte-width of the rows by a factor of two each time it is pressed.
 Set Row Width	Opens a window in which you can specify the width of the rows.

Button	Effect
 Reload	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Clicking this button is equivalent to selecting Reload from Target from the Memory menu or the shortcut menu.
 Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Clicking this button is equivalent to selecting Write Changes to Target from the Memory menu or the shortcut menu.
 Lock Address	Locks the address of the symbol in the location bar. This is the default behavior. To unlock the address and track the location of the symbol each time it changes, click this button so that it no longer appears pushed down. For more information, see “Locking the Current Symbol's Location” on page 325.
 Freeze	Prevents location changes and automatic updates. To re-enable these window updates, click the Freeze button again.
 Open Cache View	Opens the Cache View window. See “The Cache View Window” on page 392. This button is not available for all targets.
 Find in Caches	Opens the Cache Find window on the active location. See “The Cache Find Window” on page 396. This button is not available for all targets.
 Save Current View to File	Saves the contents of the Memory View window to a text file. The information contained in the text file is formatted exactly as it appears in the columns of the Memory View window.
 Print	Prints the memory pane contents. All view column data is printed for each address currently in view.
 Close	Closes the Memory View window. Clicking this button is equivalent to selecting Memory → Close. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons. To access this option from the Debugger, select Config → Options → General Tab.

Memory View Menus

The following sections describe the menu items available from the **Memory View** window.

The Memory Menu

The following table lists the options available in the **Memory** menu.

Item	Meaning
Copy	Opens a window that allows you to copy one region of memory to another. This is equivalent to the copy -gui command.
Fill	Opens a window that allows you to fill a specified region of memory with the given value. This is equivalent to the fill -gui command.
Find	Opens a window that allows you to search memory for a specified value. This is equivalent to the find -gui command.
Compare	Opens a window that allows you to compare two regions of memory. This is equivalent to the compare -gui command.
Load	Opens a window that allows you to load the contents of a file on the host machine into a portion of memory on the target. This is equivalent to the memload -gui command.
Dump	Opens a window that allows you to save a region of memory on the target to a file on the host. This is equivalent to the memdump -gui command.
Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Selecting this menu item is equivalent to clicking the Write Changes to Target button () or selecting Write Changes to Target from the shortcut menu.

Item	Meaning
Reload from Target	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Selecting this menu item is equivalent to clicking the Reload button () or selecting Reload from Target from the shortcut menu.
Set Access Sizes	Opens a window that allows you to specify preferred read and write access sizes. Selecting this menu item is equivalent to double-clicking the access size display area.
Use Block Reads	Toggles whether or not the Memory View window performs all target memory reads as block memory accesses. By default, this option is enabled. For more information, see “Disabling Block Reads” on page 329. Selecting this menu item is equivalent to right-clicking the access size display area and selecting Use Block Reads.
Open Cache View	Opens the Cache View window. See “The Cache View Window” on page 392. Selecting this menu item is equivalent to clicking the Open Cache View button () . This menu item is not available for all targets.
Find in Caches	Opens the Cache Find window on the active location. See “The Cache Find Window” on page 396. Selecting this menu item is equivalent to clicking the Find in Caches button () . This menu item is not available for all targets.
Save View to File	Saves the contents of the Memory View window to a text file. The information contained in the text file is formatted exactly as it appears in the columns of the Memory View window. Selecting this menu item is equivalent to clicking the Save Current View to File button () .
Print	Prints the memory pane contents. All view column data is printed for each address currently in view. Selecting this menu item is equivalent to clicking the Print button () .
Save Settings as Default	Saves the memory pane settings as the default Memory View window configuration.

Item	Meaning
Close	Closes the Memory View window. Selecting this menu item is equivalent to clicking the Close button (X).

For information about the commands referenced in the preceding table, see Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

The View Menu

The following table lists the options available in the **View** menu.

Item	Meaning
Add New Column	Opens a submenu providing a number of view column formats. Selecting a format adds a view column of the specified type to the memory pane. New columns are added to the right of the last column in the window. For more information about view formats, see the “The Column Submenu” on page 334.
Column <i>number name</i>	Opens a submenu with a number of options affecting the specified column. See the “The Column Submenu” on page 334. This menu item is repeated for each column. Selecting this menu item is equivalent to right-clicking a column's header.

The Column Submenu

The following table lists the options available from the **View** → **Column *number name*** submenu. You can also access these same menu items by right-clicking a column's header. You may see additional view column formats if your CPU supports them.

Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

Item	Meaning
Hide/Show	Hides or shows the column. This option toggles between Hide and Show .
Remove	Deletes the column.
Size to Fit	Sets the column to automatically adjust its width to contain displayed data.
Hex	Displays the column's memory contents as hexadecimal numbers.*
Decimal	Displays the column's memory contents as decimal numbers.*
Binary	Displays the column's memory contents as binary numbers.*
Floating Point	Displays the column's memory contents as floating-point numbers.*
Fixed Point	Displays the column's memory contents as fixed point numbers.*
Ascii	Displays the column's memory contents as ASCII characters. Any memory that cannot be represented by an ASCII character is displayed as a dot (.).
Symbols	Displays the column's memory contents as a comma-separated list of global symbols. Symbols that continue from previous addresses are represented with the string “. . .”.
Disassembly	Displays the column's memory contents as a semicolon-separated list of assembly instructions. In regions of memory without symbol information, inaccurate instructions may be displayed if the beginning of the window is not aligned with the start of an instruction.
Offset	Displays the offset between the active location and the start of the row.
<i>number</i> byte(s)	Sets the column's unit size. Available <i>numbers</i> are 1, 2, 4, and 8. These menu items are only available for the Hex, Decimal, and Binary columns.
Signed	Interprets memory data as signed decimal integers. This menu item is only available for the Decimal column.
Unsigned	Interprets memory data as unsigned decimal integers. This menu item is only available for the Decimal column.

Item	Meaning
Single Precision	Interprets memory data as single precision floating point numbers. This menu item is only available for the Floating Point column.
Double Precision	Interprets memory data as double precision floating point numbers. This menu item is only available for the Floating Point column.
q15	Interprets memory data as q15 fixed point numbers. This menu item is only available for the Fixed Point column.
q31	Interprets memory data as q31 fixed point numbers. This menu item is only available for the Fixed Point column.
Big Endian	Sets the column byte order to big endian. If this is not the default byte order for your target, the setting is displayed in the column header. This menu item is only available for the Hex , Decimal , Binary , Floating Point , and Fixed Point columns.
Little Endian	Sets the column byte order to little endian. If this is not the default byte order for your target, the setting is displayed in the column header. This menu item is only available for the Hex , Decimal , Binary , Floating Point , and Fixed Point columns.

*Numerical formats display unreadable memory as a series of dashes.

The Shortcut Menu

The following table lists the options available in the shortcut menu that appears when you right-click the memory pane.

Item	Meaning
Set Hardware Breakpoint	Opens a submenu that allows you to set a hardware breakpoint on the memory address you right-clicked. For a description of the items available in this submenu, see “The Set Hardware Breakpoint Submenu” on page 337. This menu item is only available for the Address column.

Item	Meaning
Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Selecting this menu item is equivalent to clicking the Write Changes to Target button () or selecting Memory → Write Changes to Target .
Reload from Target	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Selecting this menu item is equivalent to clicking the Reload button () or selecting Memory → Reload from Target .

The Set Hardware Breakpoint Submenu

The following table lists the options available from the shortcut menu's **Set Hardware Breakpoint** submenu. This submenu is only available for the **Address** column.

Item	Meaning
Set Write Hardware Breakpoint	Sets a new hardware breakpoint that is hit when your program writes to the memory address you right-clicked.
Set and Edit	Opens the Hardware Breakpoint Editor with the memory address that you right-clicked already filled in. This allows you to configure the properties of the hardware breakpoint before it is set.

Chapter 16

Viewing Memory Allocation Information

Contents

Using the Memory Allocations Window 340

Using the Memory Allocations Window

If you built or linked your program using the MULTI Builder, you can use the **Memory Allocations** window to examine your program's heap allocations. The **Memory Allocations** window is most useful for tracking down memory leaks, displaying overall heap usage, visualizing heap fragmentation, and detecting run-time memory errors.

To get the most out of the **Memory Allocations** window when you are developing for an embedded target, enable run-time memory checking by setting Builder or driver options prior to compiling your program. For Solaris and x86-based Linux targets, you may be able to access run-time memory checking information without setting build options. Unless you have enabled call count profiling, simply open the **Memory Allocations** window before starting program execution. If you have enabled call count profiling, you must explicitly enable run-time memory checking via Builder or driver options. See the *MULTI: Building Applications* book for information about enabling run-time memory checks.

You can open the **Memory Allocations** window at any time by doing one of the following:

- Select **View → Memory Allocations**.
- Enter the **heapview** command in the command pane. For more information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

The **Memory Allocations** window is designed for use during run-mode debugging. However, the window may also display helpful information if launched on the master process during a freeze-mode debugging session (that is, if the master process is selected in the target list when you open the window). The **Memory Allocations** window is not accessible if you attempt to open it when an OSA task is selected in the target list. For information about freeze-mode debugging, the master process, and OSA tasks, see Chapter 26, “Freeze-Mode Debugging and OS-Awareness” on page 605.

Before you can perform any of the operations available through the **Memory Allocations** window, your process must be halted, and, if you are debugging in run

mode, you must be able to run the program; it cannot be blocked or pending. A convenient way to halt the process is to set a breakpoint on the last line of your program's source code and then open the window when the process hits the breakpoint. For information about run-mode debugging, see Chapter 25, “Run-Mode Debugging” on page 577.



Note

When you use the **Memory Allocations** window to view allocations, leaks, and run-time errors, code that may allow interrupts to be serviced is executed on the target. If any interrupt servicing results in calls to memory allocation routines, the memory checking code may crash or give inaccurate results. You may need to disable interrupts before using these memory analysis operations. Use of the visualization should be safe, however, as it only reads the target memory and does not execute any code on the target.

The **Memory Allocations** window contains the following three tabs:

- **Visualization** — Displays an interactive visualization of the program's heap and information about the application's overall memory usage. For more information, see “Viewing the Memory Allocation Visualization” on page 343.
- **Leaks** — Displays unreachable blocks of memory. For more information, see “Viewing Memory Leaks and Allocation Information” on page 347.
- **Allocations** — Displays all allocated blocks of memory, including the unreachable blocks displayed on the **Leaks** tab. For more information, see “Viewing Memory Leaks and Allocation Information” on page 347.

By default, the **Visualization** tab is displayed when the **Memory Allocations** window is first opened (unless you have used the **showleaks** or **showallocations** argument with the **heapview** command). For information about the **heapview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

The deprecated **findleaks** command provided in previous versions of MULTI is equivalent to **heapview showleaks**.

The **Memory Allocations** window contains four menus: **File**, **Checking**, **View**, and **Help**. Menu items are described in the table below. Unless otherwise stated, all descriptions of toggle menu items explain the behavior of the menu item when enabled.

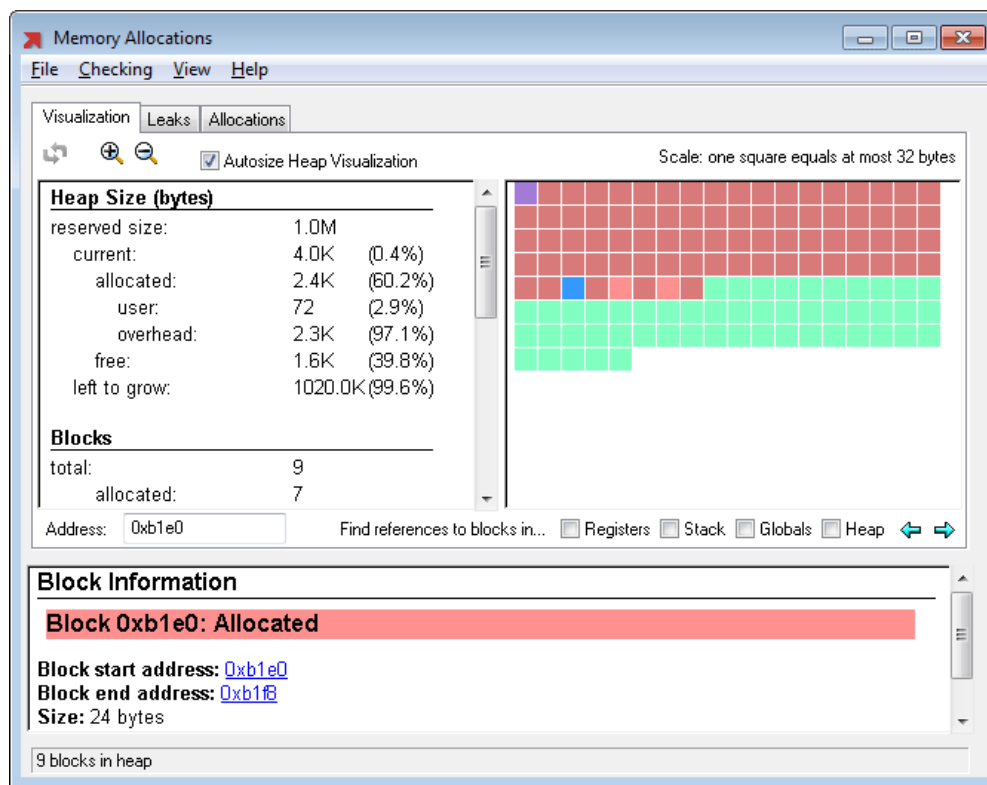
File → Save <i>information</i>	Opens a Save Report dialog that allows you to specify a file to save the contents of the active tab (<i>information</i> will appear on the menu as Visualization , Leaks , or Allocations , depending on which tab is active).
File → Print <i>information</i>	Prints the contents of the active tab (<i>information</i> will appear on the menu as Visualization , Leaks , or Allocations , depending on which tab is active).
File → Close	Closes the Memory Allocations window.
Checking → Disable Checking Checking → Minimal Checking Checking → Occasionally Maximal Checking Checking → Frequently Maximal Checking Checking → Maximal Checking	Specifies the amount and frequency of run-time error checking. For more information, see the documentation about performing selective run-time memory checking in the <i>MULTI: Building Applications</i> book. Note: The current state of the Checking menu items is stored on the target. Selecting an item from the Checking menu causes a procedure call to be executed on the target, so be sure the process is halted in a safe location first (see “Before Performing Operations that Execute Procedure Calls” on page 349). If the target cannot be accessed, the menu items in the Checking menu appear dimmed.
View → Human Readable Numbers	Displays rounded numbers in the Memory Allocations window.
View → Show Hexadecimal Values	Displays the hexadecimal equivalent for memory sizes in the Memory Allocations window.
View → Display Heap Discontinuities as Blocks	Displays unused sections of the heap as memory blocks. Otherwise, discontinuities are collapsed into a single bar.
View → Clear Runtime Errors	Clears the run-time errors displayed in the lower pane of the Memory Allocations window.

Help → Memory Allocations Help

Opens the MULTI Help Viewer on documentation for the **Memory Allocations** window.

Viewing the Memory Allocation Visualization

To view a visualization of an application's memory usage, click the **Visualization** tab of the **Memory Allocations** window (if it is not already the active tab) and click the **Refresh Visualization** button (↻). An example **Visualization** tab view is shown next.



The **Visualization** tab of the **Memory Allocations** window is divided into several panes.

The upper-left pane displays general statistics about the memory usage of your application. The information that is available varies depending upon whether your application was built with run-time memory checking enabled. The following list describes the information that may be available in this pane.

- **Heap Size (bytes):**

- **reserved size** — Amount of memory initially reserved for the heap in bytes.
- **current** — Current size of the heap given to the application by the operating system.



Note

The current heap size may exceed the reserved size if the `malloc()` library extends the heap beyond the region initially reserved for it. For more information, see your target operating system's documentation for `malloc()`. If you are using INTEGRITY, see the documentation about heap configuration in the *INTEGRITY Development Guide*.

- **allocated** — Aggregate amount of heap memory currently used by your application in allocated blocks. This is further broken down into:
 - **user** — Amount of heap memory allocated by the user. This information is only present when run-time memory checking is enabled.
 - **overhead** — Amount of memory used internally by the `malloc()` library for memory error checking. This information is only present when run-time memory checking is enabled.
- **free** — Aggregate amount of heap memory available to your application in free heap blocks.
- **left to grow** — Amount of reserved memory that is left for the heap to grow. This information is not present if the heap size exceeds the reserved size and extends into unreserved memory.

- **Blocks:**

- **total** — Total number of blocks in the heap visualization.
- **allocated** — Number of blocks in the heap allocated by the application.
- **free** — Number of free blocks in the heap.
- **discontinuities** — Number of discontinuities in the heap section. Discontinuities can appear when the heap extends into unreserved memory.

- **Calls** — Number of calls to the memory management functions **sbrk()**, **malloc()**, **calloc()**, **realloc()**, and **free()**. The **sbrk()** function is typically called internally by the **malloc()** library to request memory from the operating system. These functions are only present when run-time memory checking is enabled.
- **Other Statistics:**
 - **block alignment** — All heap blocks are a multiple of the alignment for the architecture currently being debugged.
 - **cumulative allocated** — Total number of bytes the application has allocated up to the current point in its execution. Because this statistic counts only allocations, it never decreases during the course of execution. This information is only present when run-time memory checking is enabled.
 - **peak user allocated** — Maximum aggregate amount of memory the application has allocated at any one time. This information is only present when run-time memory checking is enabled.
 - **largest allocated block size** — Size of the largest block of memory currently allocated. This information is only present when run-time memory checking is enabled.
 - **largest free block size** — Size of the current largest free block of memory. This information is only present when run-time memory checking is enabled.

The upper-right pane of the **Visualization** tab displays a visualization of the heap. In this visualization, sequences of red squares indicate allocated blocks of memory, while sequences of green squares indicate free blocks of memory. Adjacent allocated or free blocks alternate shades for clarity. Blocks that are recognized as `malloc()` overhead are marked in purple. Discontinuities in the heap are shown as long gray bars unless you have enabled **View → Display Heap Discontinuities as Blocks**, in which case, discontinuities are displayed as gray blocks.

When **Autosize Heap Visualization** is selected (the default), the visualization is sized to fit best in the window. The zoom level is chosen automatically based on the visible size of the visualization pane and the amount of allocated and free space in the heap. You can use the zoom buttons (🔍 and 🔍) to look at specific sections in more or less detail. Clicking one of these buttons automatically deselects **Autosize Heap Visualization**. Note that each heap block (no matter how small) is always

depicted using at least one square, so the amount of heap represented by each square may vary widely when you have zoomed out all the way.

Click any of the squares in the heap visualization to get more details about a specific heap block. These details appear in the bottom pane, described next. You can navigate among heap blocks by clicking them in the visualization or by using the arrows located in the middle-right side of window.

The bottom pane of the **Visualization** tab displays specific details about the heap block you have chosen in the heap visualization pane. In addition, the check boxes located above the bottom pane allow you to search several locations in memory for references to a particular heap block. The following list describes the memory locations you can search.

- **Registers** — Search the general purpose registers for references to this heap block.
- **Stack** — Search the program stack for references to this heap block.
- **Globals** — Search the program globals (such as those defined in `.data` or `.bss` sections) for references to this heap block.
- **Heap** — Search the other allocated blocks of the heap for references to this heap block.

If run-time errors are detected while the **Memory Allocations** window is open, they are displayed below the block references. To clear run-time errors, select **View** → **Clear Runtime Errors**.

Updating the Visualization

Because gathering heap visualization information can take a long time, the **Visualization** tab is not automatically refreshed when your program's state changes. A warning message appears, however, and you can click the **Refresh Visualization** button (🔄) to refresh the visualization and the statistics. Switching among the tabs does not refresh the visualization.

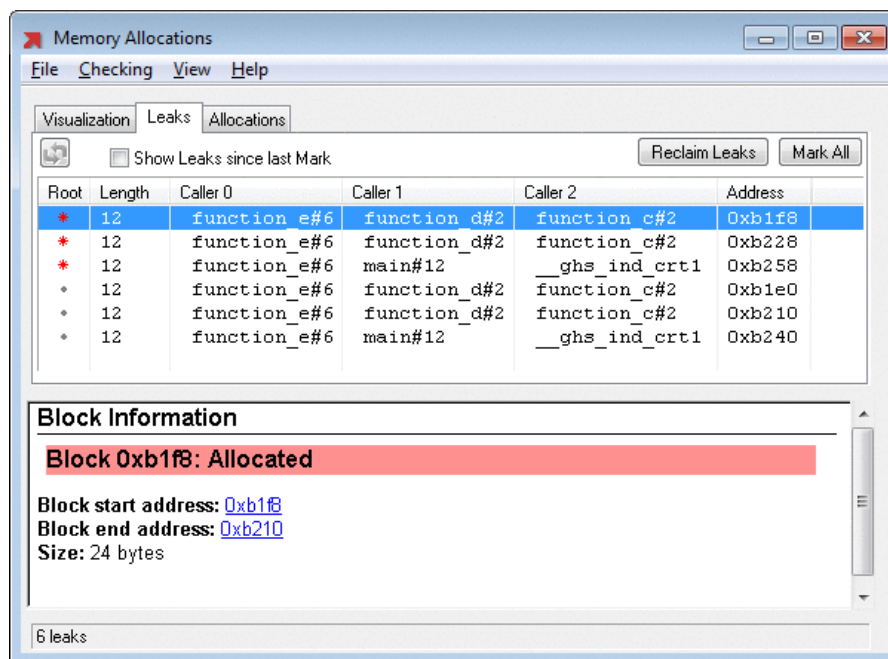


Note

Updating the **Visualization** tab while the program is halted inside `malloc()` library code may cause incorrect heap data to be displayed.

Viewing Memory Leaks and Allocation Information

The **Leaks** tab of the **Memory Allocations** window displays unreachable blocks of memory, while the **Allocations** tab displays all allocated blocks of memory, including unreachable blocks. An example **Leaks** tab view is shown next.



The columns of the **Leaks** and **Allocations** tabs display the following information for each block of memory:

- **Root** — Any memory block that is referenced by another block in the heap is marked with a dot in this column. If a memory block is not referenced by anything else on the heap, it is considered a root and is marked with a red asterisk. In cases where a memory block contains pointers to memory that becomes unreachable when the block itself becomes unreachable, this marking makes it easier to find the origin of memory leaks. For example, suppose a linked list is leaked. The memory block that contains the head of the linked list is marked with a red asterisk, and everything following the head of the list is marked with a gray dot.

This information is only present when run-time memory checking is enabled.

- **Length** — The size of the allocated block in bytes.

- **Caller 0, Caller 1**, etc. — The procedure and line number (or address) of the first five levels of the call stack of the allocation. (On some native targets (Solaris and x86-based Linux), there may be more than five levels.)
- **Address** — The address of the block of memory.
- **PID or TID** — The process or task ID of the process that called `malloc()`, if applicable. This column is only shown when run-time memory checking is enabled and the target supports reporting of process or task IDs on a per-allocation basis.

If the **Leaks** or **Allocations** tab is up to date with respect to the **Visualization** tab, block information for the selected leak or allocation (including the references selected in the **Visualization** tab) is shown in the bottom pane.



Note

Viewing memory leaks and allocation information requires that a small amount of target memory be available. If the heap has been exhausted, checking for leaks and allocations is not supported.

Manipulating Procedures and Memory Blocks

You can perform the following actions on the information that is displayed on the **Leaks** and **Allocations** tabs of the **Memory Allocations** window:

To	Do this
View a memory block in a Memory View window	Double-click the address of the memory block.
Display a procedure in the Debugger's source pane	Click the procedure.
Edit a procedure in the Editor	Double-click the procedure.
Reclaim some leaks by calling <code>free()</code> on them	Select the desired leaks and click the Reclaim Leaks button on the Leaks tab. Note: This causes a procedure call to be executed on the target, so be sure the process is halted in a safe location. For more information, see “Before Performing Operations that Execute Procedure Calls” on page 349.

Before Performing Operations that Execute Procedure Calls

Some of the operations available from the **Leaks** and **Allocations** tabs of the **Memory Allocations** window cause a procedure call to be executed on the target. As a result, you should be sure that the process is halted in a safe location before performing one of these operations, which are identified in the following list:

- Displaying the **Checking** menu or selecting any of its items.
- Refreshing leaks or allocations (see “Refreshing Leaks and Allocations Information” on page 349)
- Reclaiming leaks (see “Manipulating Procedures and Memory Blocks” on page 348)
- Tracking leaks or allocations (see “Tracking Leaks and Allocations” on page 350)

A convenient way to halt your process in a safe location is to set a breakpoint in the program and wait for the breakpoint to be hit.

The following information describes locations where it is unsafe to be halted when performing one of the preceding operations.

- On INTEGRITY, the program should not be halted inside a Task that you have not created, such as the kernel's Idle Task. For more information, see the documentation about run-mode debugging limitations in the *INTEGRITY Development Guide*.
- The program should not be halted inside a task that holds the lock from `__ghsLock()`. Additionally, no task in the address space should be halted in a location where it holds the lock from `__ghsLock()`. The preceding operations, which rely on being able to obtain this lock, become pended if you try to complete them while the lock from `__ghsLock()` is held.
- When run-time memory checking is enabled, the program should not be halted inside `malloc()` library code.

Refreshing Leaks and Allocations Information

Because the operation of finding leaks and allocations can be slow and invasive, the **Leaks** and **Allocations** tabs do not automatically refresh every time the process

is stopped. To refresh the leaks and allocations information, click the **Refresh Leaks** or **Refresh Allocations** button (🔄). Note that refreshing leaks or allocations causes a procedure call to be executed on the target, so be sure that the process is halted in a safe location. For more information, see “Before Performing Operations that Execute Procedure Calls” on page 349.



Note

Refreshing leaks or allocations requires a small amount of dynamic memory. To guarantee heap integrity, ensure that the target will not exhaust its heap.



Note

Because tasks running in the same address space share memory, refreshing leaks or allocations information for one task while other tasks in the same address space are still running can lead to inconsistent results. If MULTI detects that other tasks in an INTEGRITY AddressSpace are still running, a dialog box appears to inform you of this and to give you the opportunity to cancel the refresh operation. In this situation, we recommend that you do the following:

1. Click **No** to cancel the refresh operation.
2. Set breakpoints in your code to stop each of the other tasks. Note that it is important to stop each of the tasks in your code, and not in kernel code or in other shared library code. Failing to stop the tasks in your code can cause the refresh operation to become pended, waiting for system resources to be released.
3. Wait until all your tasks have halted at the breakpoints you set.
4. Refresh leaks or allocations information for the original task.

Tracking Leaks and Allocations

You can track memory leaks and allocations while your process executes by using the **Mark All** button, which marks all blocks of memory currently allocated in the heap, and the **Show Leaks since last Mark** or **Show Allocations since last Mark** check box, which toggles whether only unmarked (or new) leaks or allocations are

shown. Marked leaks are shaded gray in the **Leaks** and **Allocations** tabs. To monitor memory leaks and allocations:

1. Halt the application at a safe and useful checkpoint. For information about safe locations, see “Before Performing Operations that Execute Procedure Calls” on page 349.
2. Click the **Refresh Leaks** or **Refresh Allocations** button () to update the **Leaks** or **Allocations** information.
3. While still halted in a safe location, click the **Mark All** button on the **Leaks** or **Allocations** tab of the **Memory Allocations** window. This marks the current set of allocations.
4. Resume execution of the application.
5. Halt the application in a safe location again.
6. Click the **Refresh Leaks** or **Refresh Allocations** button () to update the **Leaks** or **Allocations** information.
7. Select the **Show Leaks since last Mark** or **Show Allocations since last Mark** check box so that the tabs display only those leaks and allocations that occurred since the last time you marked the allocations.
8. Repeat this process as necessary as you run through the entire application.



Note

The **Mark All** button and the **Show Leaks since last Mark** and **Show Allocations since last Mark** check boxes are only available when run-time memory checking is enabled.



Tip

You can achieve the same results by using Debugger commands in the command pane. The command **heapview setmark** marks the current set of allocations, and the command **heapview showleaks -new** or **heapview showallocations -new** displays the new leaks or allocations. For more information about the **heapview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Caveats for Leak Information

- The **Leaks** tab of the **Memory Allocations** window may not display all of the leaks in the program being debugged at any given moment. If dead variables are still on the stack or in registers, MULTI assumes they are alive and does not report anything they reference as a leak. Once the process has run further (for example, returned out of the current routine, which should clear out the stack), some or all of these false negatives vanish and additional leaks may show up.
- In some target implementations, a register may be a base register, a general purpose register, or both. It is possible that the value of the base register may point into the heap (for example, on some targets, when the register is the SDA base pointer, and there is no data). This can prevent a memory leak from being reported for a dead object at the same location as the base register.
- Blocks referenced only via pointer arithmetic may be falsely marked as leaks. Additionally, false positives may occur in INTEGRITY applications. For more information, see the documentation about memory leaks in the *INTEGRITY Development Guide*.

Chapter 17

Collecting and Viewing Profiling Data

Contents

Types of Profiling Data	354
Overview of Profiling Methods	355
Collecting Profiling Data	356
Caveats for Profiling	360
Disabling the Collection of Profiling Data	360
Viewing Profiling Information	361
The Profile Window	361
Viewing Profiling Information in the Debugger	377
Performing Range Analyses	379
Managing Profiling Data	380

MULTI's powerful profiling capabilities allow you to gather and view information about the performance and coverage of your programs. You can use this information to analyze and optimize the efficiency of your code.

Types of Profiling Data

MULTI supports three basic types of profiling data: *program counter (PC) samples*, *call count data* (with and without call graph support), and *coverage analysis data*. Each are described below:

- PC samples — Contain data about where the process spends its time. You can view how much time is spent in each function, each basic block, each source line, each machine instruction, and the entire program. You may be able to make execution faster by using this information to improve code that accounts for an unnecessarily large percentage of the total execution time.
- Call count data
 - With call graph support — Contains a record of how many times each function is called, as well as which child functions are called by each parent function and how many times each child is called.
 - Without call graph support — Contains a record of how many times each function is called.
- Coverage analysis data — Contains a record of how many times each basic block executed. If a given basic block is never executed, the group of instructions making up the block is considered dead code for the given sample input. Since the application never reaches the dead code, you can either remove the code from the application or try to discover if any functionality is missing from the application as a result of it.

Coverage data can be useful for performance analysis, especially if PC samples or trace data is not available. The Debugger can use coverage data to determine the number of times each instruction was executed and then to show how many instructions were executed in each function and source line, giving you a rough idea of how much time was spent where.

For a general description of the methods used to collect profiling data, see the next section.

Overview of Profiling Methods

Both the method that MULTI uses to collect profiling data and the types of data that MULTI collects is determined by what you choose to profile. You can profile all tasks in your system (if you are using INTEGRITY version 10 or later), a trace-enabled target, a run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program. Profiling a target running an operating system that does not support multiple address spaces (such as u-velOSity) is equivalent to profiling a stand-alone program. The following list states what types of data MULTI outputs for each of these (see “Types of Profiling Data” on page 354).

- If you profile all tasks in your system (requires INTEGRITY version 10 or later) — INTEGRITY periodically samples the PC of running tasks, and the run-mode debug server delivers the following profiling data to MULTI:
 - PC samples

MULTI uses the samples to display information for all tasks in your system. For more information, see “Profiling All Tasks in Your System” on page 603.

- If you profile a trace-enabled target — MULTI converts trace data into profiling data that is much more complete, in terms of which instructions are represented, than profiling data generated by sampling methods. (Sampling methods are usually timer-driven and can miss important hot spots in a process.)

MULTI is able to convert collected trace data into all the following types of profiling data:

- Program counter (PC) samples
- Call count with call graph data
- Coverage analysis data
- If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program — MULTI collects one or more of the following types of data:
 - PC samples
 - Call count data with or without call graph support
 - Coverage analysis data



Note

Only PC samples provide task-specific profiling data. You cannot collect call count or coverage analysis profiling data for a specific task—only for the encompassing AddressSpace.

Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program

Green Hills simulators, run-mode debug servers, and some native debug servers can sample the PC without instrumenting your code. If you are using a debug server that does not support PC sampling, or if you want to be able to analyze additional types of profiling data, you must compile your program with the appropriate Builder or driver options set. See the *MULTI: Building Applications* book for information about obtaining profiling information.

Collecting Profiling Data

This section contains detailed instructions for collecting profiling data for all tasks in your system, for a trace-enabled target, a single run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program. For a general overview of the methods used to collect profiling data, see “Overview of Profiling Methods” on page 355.

The scope of profiling is limited to what you select in the target list *prior* to enabling the collection of profiling data. If you select a run-mode connection and then enable profiling, profiling data is collected for all tasks in your system. If you select a trace-enabled target, profiling data is collected for everything on the target. If you select a run-mode task or AddressSpace on an INTEGRITY target, or if you select a stand-alone program, profiling data is collected for the task (PC samples only), AddressSpace, or stand-alone program. If you collect profiling data for one of these items and then want to profile another, you must disable profiling collection and clear existing profiling data before selecting and collecting data for the second item.



Note

If you enable profiling from the Trace List or PathAnalyzer, or with the **tracepro** command or the **TimeMachine** → **Profile** menu selection, profiling data is generated for all traced tasks or AddressSpaces, regardless

of the selection in the target list. (For information about the **tracepro** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.)

Collecting Profiling Data for All Tasks in Your System

To collect profiling data for all tasks in your system (requires INTEGRITY version 10 or later), perform the following steps:

1. In the Debugger's target list, select the run-mode connection.
2. Enable the collection of profiling data and open the **Profile** window by doing one of the following:
 - Select **View** → **Profile**.
 - In the Debugger command pane, enter the **profile** command. For information about the **profile** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

By default, profiling is enabled when you open the **Profile** window. Do not close the **Profile** window until you are finished viewing profiling information. (Closing this window halts the collection of profiling data and clears existing profiling data.) For information about the **Profile** window, see “The Profile Window” on page 361.

3. In the target list, select a task to view profiling data for that task in the **Profile** window.

For information about the type of data that appears in the **Profile** window, see “Overview of Profiling Methods” on page 355.

In addition to the profiling information that appears in the **Profile** window, the Debugger's target list displays processor usage for all tasks in your system. For more information, see “Profiling All Tasks in Your System” on page 603. For information about other ways to view collected profiling data, see “Viewing Profiling Information” on page 361.

Collecting Profiling Data for a Trace-Enabled Target

To collect profiling data for a trace-enabled target, perform the following steps:

1. Trace the code that you want to profile, and retrieve the trace data as described in “Managing Trace Data” on page 405.
2. Generate profiling data from the collected trace data and open the **Profile** window on that data by doing one of the following:
 - In the Debugger, select **TimeMachine** → **Profile**.
 - In the Trace List or the PathAnalyzer, click the **Profile** button (.
 - In the Debugger command pane, enter the **tracepro** command. For information about the **tracepro** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
3. If you are profiling an INTEGRITY, velOSity, or u-velOSity system — Select a traced task or AddressSpace in the target list to view profiling information (if available) for the selected item in the **Profile** window.



Note

If INTEGRITY trace data includes task switch information, no profiling data is displayed when you select an AddressSpace in the target list. If the trace data does not include task switch information, profiling data for all tasks in the same AddressSpace is displayed regardless of which task is selected. If you select the OSA master process, the **Profile** window only shows profiling data for code that is not associated with a specific task, such as exception handlers and the scheduler.



Note

Depending on what kind of trace hardware you are using, the trace data may contain cycle counts, instructions, or timestamps. The unit used to represent time will be displayed in the status bar of the **Profile** window (for example, you may see **Time is in cycles.**).

For information about the types of data that appear in the **Profile** window, see “Overview of Profiling Methods” on page 355.

You can view collected profiling data in a variety of ways. For more information, see “Viewing Profiling Information” on page 361.

Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program

To collect profiling data for a run-mode task or AddressSpace on an INTEGRITY target, or to collect profiling data for a stand-alone program, perform the following steps:

1. Before compiling your program, determine whether it is necessary to set Builder options (or corresponding compiler driver options) to enable profiling. For information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 356.
2. If necessary, compile your program with the appropriate profiling options set.
3. Connect to your target.
4. In the Debugger's target list, select the run-mode task, run-mode AddressSpace, or stand-alone program that you want to profile.
5. Enable the collection of profiling data and open the **Profile** window by doing one of the following:
 - Select **View** → **Profile**.
 - In the Debugger command pane, enter the **profile** command. For information about the **profile** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

By default, profiling is enabled when you open the **Profile** window. Do not close the **Profile** window until you are finished viewing profiling information. (Closing this window halts the collection of profiling data and clears existing profiling data.) For information about the **Profile** window, see “The Profile Window” on page 361.

6. Step or run the program.
7. Usually, you should let your program run at least once before expecting to see any information in the open **Profile** window. (By default, MULTI processes collected profiling data automatically each time your program exits. After the data has been processed, profiling information appears in the **Profile** window.) In some situations, however, you may prefer to halt the process early and

manually dump and process collected profiling data. For more information, see “Manually Dumping Profiling Data” on page 381 and “Manually Processing Profiling Data” on page 383.

For information about the types of data that appear in the **Profile** window, see “Overview of Profiling Methods” on page 355.

For information about the different ways you can view collected profiling data, see “Viewing Profiling Information” on page 361.

Caveats for Profiling

- If you are profiling a run-mode task on an INTEGRITY target, only PC samples provide task-specific profiling data. You cannot collect call count or coverage analysis profiling data for a specific task—only for the encompassing AddressSpace.
- If you profile a stand-alone program, you can only collect PC samples if you open the **Profile** window prior to downloading, stepping, or running the program.

Disabling the Collection of Profiling Data



Note

This information is only relevant if you are profiling all tasks in your system, a run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program.

To disable the collection of profiling data:

- If you are profiling all tasks in your system — Halt the target. Alternatively, click the **Close** button (✖) to close the **Profile** window, disable the collection of profiling data, and clear existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)
- If you are profiling a run-mode task or AddressSpace, or if you are profiling a stand-alone program — In the **Profile** window, click the **Stop collecting profile information** button (🛑). Alternatively, click the **Close** button (✖) to close the **Profile** window, disable the collection of profiling data, and clear

existing profiling data. (Whether this last button appears on your toolbar depends on your MULTI configuration.)

Viewing Profiling Information

After you have collected profiling data (see “Collecting Profiling Data” on page 356), you can view the information in a variety of ways:

- *In profiling reports* — The tabs in the **Profile** window display profiling information in different report formats. For more information, see “Profiling Reports” on page 363.
- *In the Debugger window* — The Debugger can indicate the percentage of time spent in each source line (or each instruction, if in assembly-only mode or interlaced assembly mode), which lines of code were never executed, the number of times each source line was executed, or the number of times each function was called. For more information, see “Viewing Profiling Information in the Debugger” on page 377.
- *In a Range Analysis window* — This window allows you to specify a range of hexadecimal addresses to examine. For more information, see “Performing Range Analyses” on page 379.
- *In a dynamic call graph Browse window* — This window displays a call graph generated at run time. For more information, see “Browsing Dynamic Calls by Function” on page 246.



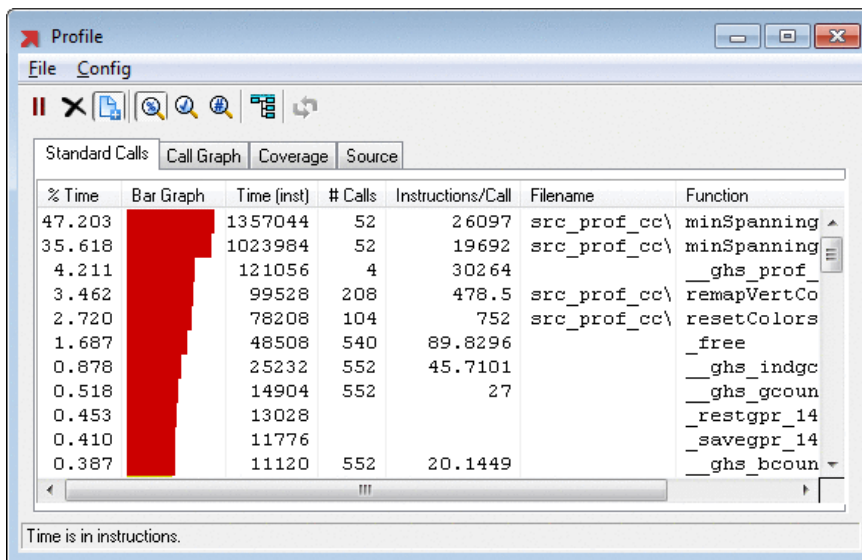
Note

Support for these display capabilities depends on what you profile and/or on the type of profiling data available. For more information, see the referenced sections.

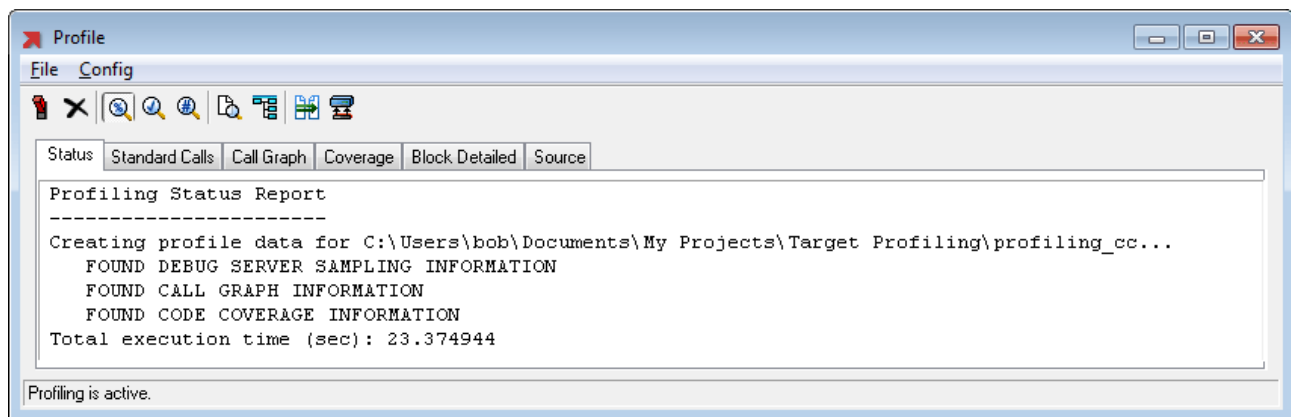
The Profile Window

The **Profile** window allows you to view collected profiling data in various report formats. For information about how to enable profiling and open the **Profile** window, see “Collecting Profiling Data for All Tasks in Your System” on page 357, “Collecting Profiling Data for a Trace-Enabled Target” on page 358, or “Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 359.

If you profile all tasks in your system or if you profile a trace-enabled target, the **Profile** window looks similar to the following graphic.



If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program, the buttons on the toolbar and the availability of reports and of menu items differ. In this situation, the **Profile** window looks similar to the following graphic.



Note

The information available in the **Profile** window may vary slightly according to your target type. For example, profiling with a cycle-accurate simulator may yield additional columns in the reports containing cycle information. For additional notes about profiling for your specific target,

see the *MULTI: Configuring Connections* and the *MULTI: Building Applications* books for your target type.

The following sections explain the various features of the **Profile** window. For a detailed description of all the menu items and buttons, see “The Profile Window Reference” on page 372.

Continual Updates in the Profile Window



Note

This information is only relevant if you profile all tasks in your system, or if you profile a trace-enabled target.

When the following conditions are met, the **Profile** window continually updates select information:

- If you are profiling all tasks in your system — The window continually updates processor usage per function (that is, the standard calls report).
- If you are profiling a trace-enabled target and **Retrieve trace when buffer fills** is enabled — The window continually updates processor usage per function, and it updates call count and call graph information (the standard calls and call graph reports). For information about the **Retrieve trace when buffer fills** option, see “The Trace Options Window” on page 480.

To pause the continual updating of information, click the **Pause updating of profile display** button () in the **Profile** window.

Profiling Reports

Each of the tabs in the **Profile** window represents a profiling report. Depending on what profiling data is available, there can be up to six profiling report tabs in the **Profile** window: **Status**, **Standard Calls**, **Call Graph**, **Coverage**, **Block Detailed**, and **Source**. To view any one of these reports in the **Profile** window, simply click the corresponding tab. (If the data required for a particular report was not collected, no tab appears for that report.) You can also view each profiling report by entering the following command in the Debugger command pane:

```
profilereport report
```

where *report* specifies one of the report types and can be:

- *status* — See “Status Report” on page 365.
- *calls* — See “Standard Calls Report” on page 365.
- *graph* — See “Call Graph Report” on page 367.
- *coveragesummary* — See “Coverage Report” on page 369.
- *coveragedetailed* — See “Block Detailed Report” on page 370.
- *sourcelines* — See “Source Report” on page 371.

Each profiling report generally consists of several columns of information and a status bar at the bottom of the report. You can sort most of the columns by clicking the column header. To sort in the opposite direction, click the same header again. To resize a column, drag the separator to the left or right.

By default, reports omit C++ qualifiers when displaying function names. You can view fully qualified function names by hovering your mouse pointer over a function, or you can display fully qualified function names in the reports by selecting **Function Names** → **Long** from the **Config** menu. To change back to the default behavior, select **Function Names** → **Short**.

Within each of the reports, single-clicking a function or other component causes the Debugger to jump to that function or component in the source pane. Double-clicking opens a MULTI Editor window on the function or component (if appropriate). Right-clicking opens a shortcut menu. For a description of the shortcut menu items, see “The Procedure Shortcut Menu” on page 708.

You can save, append, or print any profiling report by selecting a menu item or by using the **profilereport** Debugger command:

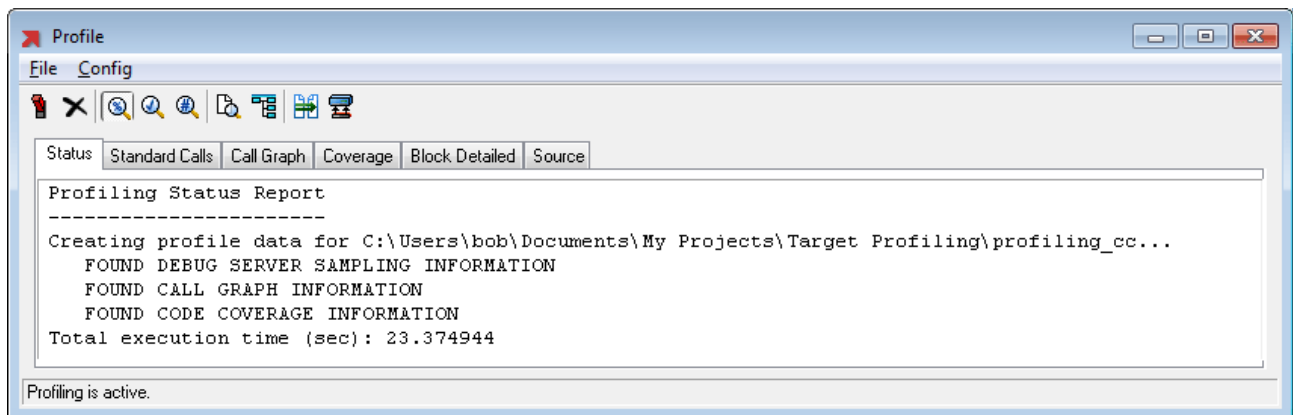
- To save the text of the report that is currently displayed in the **Profile** window, select **File** → **Save Report** or use the **profilereport save** command. The default file extension given to the saved text file is **.rep**.
- To append the text of the currently displayed report to a previously created file, select **File** → **Append Report** or use the **profilereport append** command.
- To print the text of the report that is currently displayed, select **File** → **Print Report** or use the **profilereport print** command.

For more information about the **profilereport** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Each of the profiling reports is described in greater detail in the following sections.

Status Report

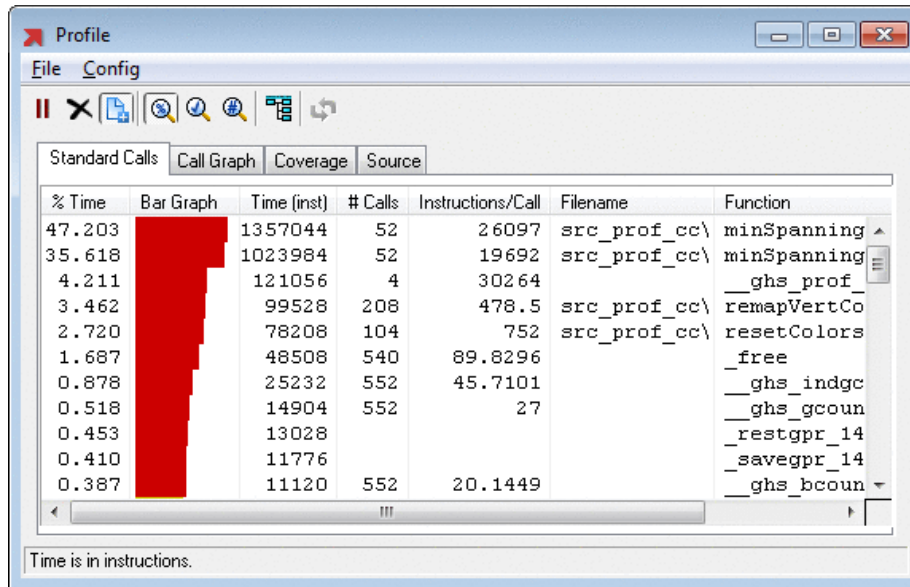
The status report is the report that appears by default when you open the **Profile** window.



This report is available if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program. It gives general information about profiling, such as what type of data has been collected. A message in the status bar indicates whether or not profiling data is being collected.

Standard Calls Report

The standard calls report gives a summary of processing time per function. It is available if PC samples, call count data, or coverage analysis data is available. For information about the availability of these data types, see “Overview of Profiling Methods” on page 355.



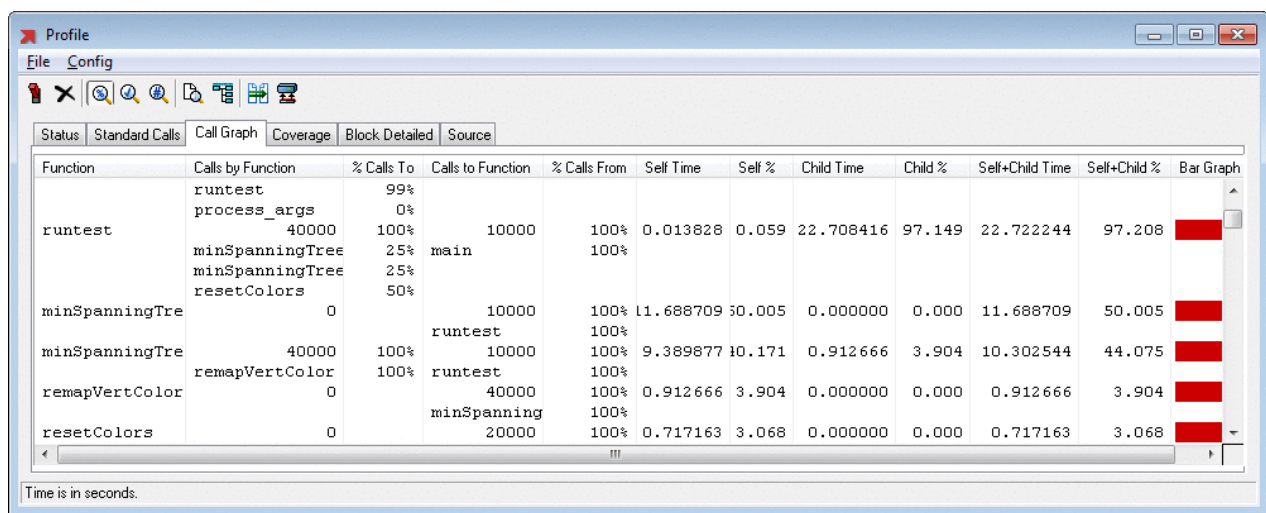
The columns of the standard calls report are described in the following table. Some columns do not contain information unless you used trace data or compiled your program with Builder or driver options that generate call count data. For more information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 356.

Header	Meaning
% Time	The percentage of the total process time spent in each function. (The total of the percentages in this column may not be exactly 100 percent, due to roundoff error.)
Bar Graph	Graphs the value of the % Time column.
Time (unit)	The amount of time in milliseconds (ms), seconds (s), instructions (inst), or cycles (cyc) spent in each function.
# Calls	The number of times that the function was called. This column only contains information if call count data is available. For information about the availability of call count data, see “Overview of Profiling Methods” on page 355. If you are profiling a trace-enabled target, call counts are not available for some internal helper functions used by the compiler. Usually these functions have names that begin with one or two underscore characters, and they are called at the beginning and end of many functions to save and restore registers.

Header	Meaning
Unit/call	The amount of time (in milliseconds, seconds, instructions, or cycles) spent on each call to the function. This column only contains information if call count data is available. For information about the availability of call count data, see “Overview of Profiling Methods” on page 355.
Filename	The location of the function.
Function	The name of the function.

Call Graph Report

The call graph report gives a summary of processing time spent per function, including time spent in each function's descendants. (Descendants of a function are all routines directly or indirectly called by that function.) This report is available if call count with call graph profiling data is available. For information about the availability of call count with call graph profiling data, see “Overview of Profiling Methods” on page 355.



The columns of the call graph report are described in the following table.

Header	Meaning
Function	The name of the function.
Calls by Function	The number of function calls made by the function, followed by the actual function(s) called.

Header	Meaning
% Calls To	The percent of the total calls made by the function to each of the listed functions.
Calls to Function	The number of times that the function was called, followed by the function(s) that called it.
% Calls From	The percent of the total calls made to the function by each of the listed functions.
Self Time*	The actual time spent in each function.
Self %	The percentage of total time spent in each function.
Child Time*‡	The actual time spent in all of the children of each function. Child times are calculated by multiplying the amount of time attributed to each child function by the percentage of times the parent called that child.
Child %‡	The percentage of total time spent in the children of each function.
Self+Child Time*‡	The total processing time spent in each function and its children.
Self+Child %‡	The percentage of total time spent in each function and its children.
Bar Graph‡	Graphs the value of the Self+Child % column.

*: The message in the status bar indicates whether the times are listed in milliseconds, seconds, instructions, or cycles.

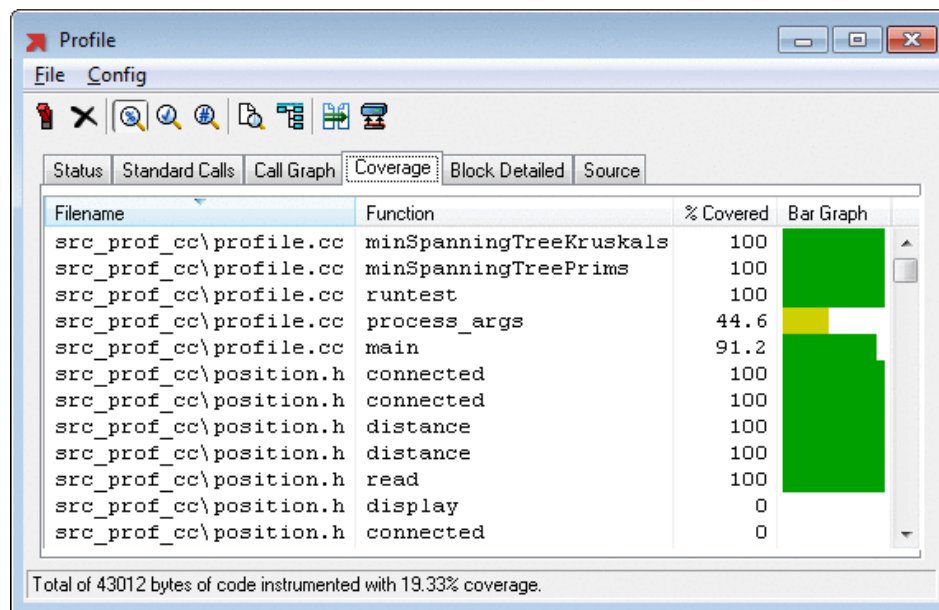
‡: The method used to calculate the **Child Time** (and everything derived from the **Child Time**) is a rough approximation. The average amount of time spent in each call to the child function is multiplied by the number of calls made by each parent to determine the **Child Time** assigned to that parent. This is a good approximation of the **Child Time** if the amount of time spent in each call to the child function does not vary significantly, but it can be a poor approximation if it does. For example, consider the case where functions A and B are the only callers of function C. Function A calls function C 1000 times, and each call takes 1 millisecond. Function B only calls function C 1 time, but the call takes 1 second. Since the profile data does not capture execution times for each individual call to function C, the Debugger would assume that each call to function C took approximately 2 milliseconds. That assumption would result in **Child Time** values of 2 seconds and 2 milliseconds when in reality the **Child Times** should both be 1 second.

Coverage Report

The coverage report provides the percentage of bytes covered. Examining this report is an easy way to find code that is not being executed. This can be useful for discovering dead code and for verifying that a test suite is testing everything it is intended to test.

Each line in the report provides the percentage of bytes of code that were executed in a function. If profiling data has been generated from trace data, the coverage report includes every function in the AddressSpace or stand-alone program. If coverage analysis data has been generated via compiler instrumentation, the coverage report includes every instrumented function in the AddressSpace or stand-alone program. Typically, standard library functions are not instrumented. The status bar message gives a summary of coverage for the entire AddressSpace or stand-alone program.

This report is available if coverage analysis data is available. For information about the availability of coverage analysis data, see “Overview of Profiling Methods” on page 355.

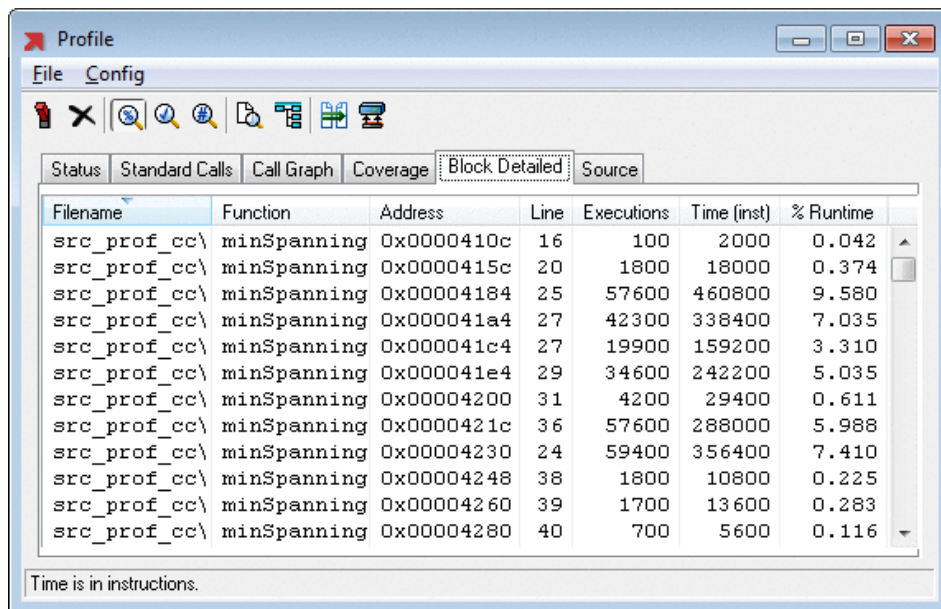


The columns of the coverage report are described in the following table.

Header	Meaning
Filename	The location of the function.
Function	The function name.
% Covered	The percentage of bytes executed.
Bar Graph	Graphs the value of the % Covered column.

Block Detailed Report

The block detailed report gives the program code coverage per basic block. It is available if coverage analysis data is available and if you did not use trace data. For information about the availability of coverage analysis data, see “Overview of Profiling Methods” on page 355.



The columns of the block detailed report are described in the following table.

Header	Meaning
Filename	The location of the function.
Function	The name of the function.
Address	The starting address of the block.

Header	Meaning
Line	The file-relative source line corresponding to the block.
Executions	The number of times the block was entered or a NOT REACHED message if zero.
Time (<i>unit</i>)	The total time in milliseconds (ms), seconds (s), or instructions (inst) spent in the block.
% Runtime	The percentage of total time spent in the block.

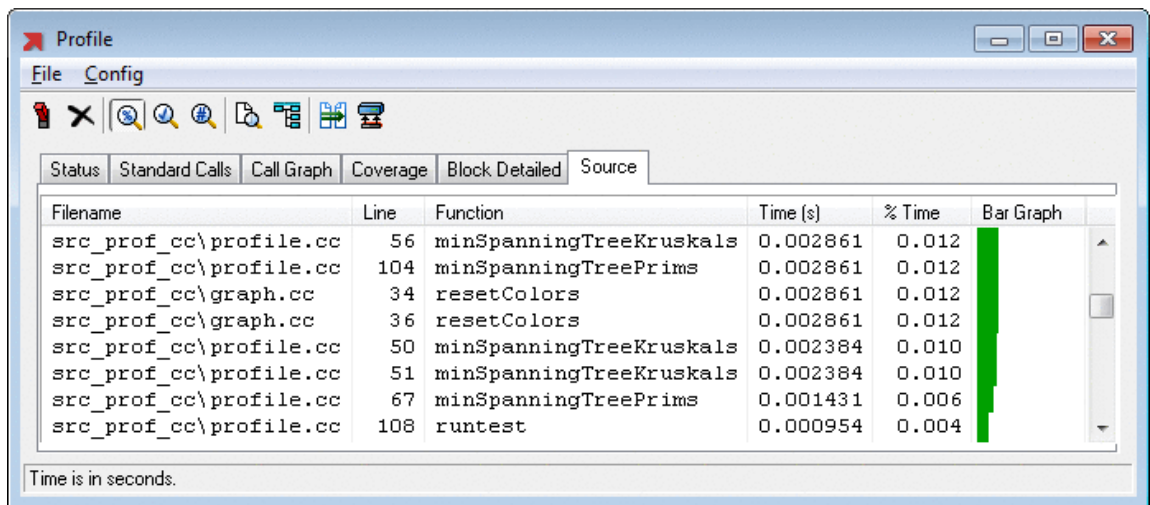
Source Report

The source report is a list of all the source lines, along with how long each took to execute. A source line is uniquely determined by filename and file-relative line number. Only lines with positive times are displayed. This report is available if PC samples are available. For information about the availability of PC samples, see “Overview of Profiling Methods” on page 355.



Note

The time it takes to process this report is proportional to the size of what you are profiling. For example, it takes longer to generate this report for very large programs than for small ones.



Filename	Line	Function	Time (s)	% Time	Bar Graph
src_prof_cc\profile.cc	56	minSpanningTreeKruskals	0.002861	0.012	
src_prof_cc\profile.cc	104	minSpanningTreePrims	0.002861	0.012	
src_prof_cc\graph.cc	34	resetColors	0.002861	0.012	
src_prof_cc\graph.cc	36	resetColors	0.002861	0.012	
src_prof_cc\profile.cc	50	minSpanningTreeKruskals	0.002384	0.010	
src_prof_cc\profile.cc	51	minSpanningTreeKruskals	0.002384	0.010	
src_prof_cc\profile.cc	67	minSpanningTreePrims	0.001431	0.006	
src_prof_cc\profile.cc	108	runtest	0.000954	0.004	

Time is in seconds.

The columns of the source report are described in the following table.

Header	Meaning
Filename	The location of the source line.
Line	The file-relative line number of the source line.
Function	The function in which the source line appears.
Time (<i>unit</i>)	The time in milliseconds (ms), seconds (s), instructions (inst), or cycles (cyc) spent on the given line.
% Time	The percentage of the total time spent on the line.
Bar Graph	Graphs the value of the % Time column.

The Profile Window Reference

The following sections provide a comprehensive description of the menu items and buttons that appear in the **Profile** window. For information about the tab views within the **Profile** window, see “Profiling Reports” on page 363.

The File Menu

The following table describes the **File** menu items and lists their command equivalents.

For information about the **profilereport** and **profilemode** commands, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning	Equivalent Command
Save Report	Saves the text of the report that is currently displayed in the Profile window. The default file extension given to the saved text file is .rep .	profilereport save <i>filename</i>
Append Report	Appends the text of the report currently displayed in the Profile window to a specified, pre-existing file.	profilereport append <i>filename</i>
Print Report	Prints the text of the report currently displayed in the Profile window.	profilereport print

Menu Item	Meaning	Equivalent Command
Close	Closes the Profile window, which halts the collection of profiling data and clears existing profiling data.	profilemode close

The Config Menu

The following table describes the **Config** menu items and lists their command equivalents, if any. These menu items are context sensitive; they may not all be available every time you open this menu. Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

For information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning	Equivalent Command
Config → New Data → Added to Old	Appends new data to old data when a new set of profiling data is processed (such as data from a new run of the program). For more information, see “Adding to or Overwriting Existing Profiling Data” on page 380.	profilemode add
Config → New Data → Replaces Old	Replaces old data with new data when a new set of profiling data is processed (such as data from a new run of the program). For more information, see “Adding to or Overwriting Existing Profiling Data” on page 380.	profilemode replace
Config → Data Processing → Automatic	Processes and stores profiling data automatically when it is dumped. This is the default unless you are using INTEGRITY. For more information, see “Manually Processing Profiling Data” on page 383.	profilemode automatic
Config → Data Processing → Manual	Does not process profiling data automatically when it is dumped. This is the default if you are using INTEGRITY. For more information, see “Manually Processing Profiling Data” on page 383.	profilemode manual

Menu Item	Meaning	Equivalent Command
Config → Function Names → Short	Omits C++ qualifiers from the function names displayed in profiling reports. (You can view fully qualified function names in the tooltips.) This option has no effect on the display of C functions.	profilemode short
Config → Function Names → Long	Displays fully qualified function names in profiling reports. Fully qualified function names include all C++ qualifiers, such as namespace and class names, function arguments, and template information. This option has no effect on the display of C functions.	profilemode long
Config → Time Units → Milliseconds	Displays all times in milliseconds.	profilemode time milliseconds
Config → Time Units → Seconds	Displays all times in seconds.	profilemode time seconds
Config → Time Units → Instructions	Displays all times in instructions.	profilemode time instructions
Config → Time Units → Cycles	Displays all times in cycles.	profilemode time cycles
Config → Bar Graphs → Linear	Draws all Bar Graph columns in profiling reports as linear graphs.	(no equivalent command)
Config → Bar Graphs → Logarithmic	Draws Bar Graph columns in the Standard Calls , Call Graph , and Source reports as logarithmic graphs. The Bar Graph column in the Coverage report is always drawn as a linear graph regardless of the setting for this option. This is the default.	(no equivalent command)

The Toolbar

The following table describes the buttons available in the **Profile** window and lists their command equivalents, if any. These buttons are context sensitive; different buttons are available in different contexts.

For information about the **profilemode** and **profdump** commands, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **browse** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Button	Meaning	Equivalent Command
 Stop collecting profile information	Indicates whether profiling data is being collected and, when clicked, toggles profiling to the opposite state (on or off).	
 Start collecting profile information	When this button is in the  position, profiling data is being collected. Clicking the button when it is in this position disables profiling collection and clears any current profiling data (but does not close the Profile window).	profilemode stop
	When this button is in the  position, profiling data is not being collected. Clicking the button when it is in this position enables profiling.	profilemode start
 Pause updating of profile display	Pauses the continual updating of information in the Profile window. For more information, see “Continual Updates in the Profile Window” on page 363.	no equivalent command
 Clear current profile information	Deletes any existing profiling data. You can use this button in conjunction with the Dump Profiling Info button () to examine profiling data gathered between two points of execution. For more information, see “Manually Dumping Profiling Data” on page 381.	profilemode clear

Button	Meaning	Equivalent Command
 Merge new data	Toggles whether new data is added to old data or whether new data replaces old data. When this button appears to be pushed down and when a new set of profiling data is processed (such as data from a new run of the program), new data is added to old data. This is the default. When this button does <i>not</i> appear to be pushed down and when a new set of profiling data is processed, the new data replaces the old data. For more information, see “Adding to or Overwriting Existing Profiling Data” on page 380.	profilemode add and profilemode replace
 Percentage View	Displays, to the left of each source code line in the Debugger, the percentage of time spent in each source line. (In assembly display mode, displays the percentage of time spent on each instruction.) This is the default view in the Debugger.	profilemode percent
 Coverage View	Highlights dead code (lines that were never executed during the profiling run) in the Debugger.	profilemode coverage
 Count View	If coverage analysis profiling data is available, clicking this button displays in the Debugger the total number of times each line (or instruction) was executed. If coverage analysis profiling data is unavailable, but call count profiling data is available, clicking this button displays the number of times each function was called. This information is displayed at the beginning of each function in the Debugger. For information about the availability of these data types, see “Overview of Profiling Methods” on page 355.	profilemode count
 Range Analysis	Opens a Range Analysis window. See “Performing Range Analyses” on page 379.	profilemode range <i>start_addr end_addr</i>

Button	Meaning	Equivalent Command
 Dynamic Call Graph	Opens a dynamic call graph Browse window centered on the function currently selected in the Profile window. If nothing is selected in the Profile window the graph is centered on the function being examined in the Debugger. For more information, see “Browsing Dynamic Calls by Function” on page 246.	browse dcalls
 Refresh Report	Updates the report with new data.	no equivalent command
 Process Data	Processes profiling data. For more information, see “Manually Processing Profiling Data” on page 383.	profilemode process
 Dump Profiling Info	Retrieves any currently available profiling data from the target. You can use this button in conjunction with the Clear current profile information button () to examine profiling data gathered between two points of execution. For more information, see “Manually Dumping Profiling Data” on page 381.	profdump
 Close	Closes the Profile window, which halts the collection of profiling data and clears existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)	profilemode close

Viewing Profiling Information in the Debugger

Once you have collected profiling data, you can use **Profile** window buttons or the **profilemode** command to view some basic profiling information directly in the source pane of the Debugger, as described below. If you profile all tasks in your system, or if you profile a trace-enabled target, the information is continually updated.



Note

The following display modes are mutually exclusive (only one type of information can be shown in the Debugger at a time). By default, the Debugger displays time percentage information, if it is available.

- *Time percentage information* — To view the percentage of time spent in each source line, click the **Percentage View** button (🔍) in the **Profile** window or run the **profilemode percent** command in the Debugger command pane. The percentage is shown to the left of each source line. (If you are in assembly display mode, the percentage of time spent on each instruction is displayed.) This feature is only available if PC samples or coverage analysis profiling data is available. For information about the availability of these data types, see “Overview of Profiling Methods” on page 355.
- *Coverage information* — To have lines of dead code (lines that were never executed) highlighted in the Debugger, click the **Coverage View** button (🔍) in the **Profile** window or run the **profilemode coverage** command in the Debugger command pane. This feature is only available if coverage analysis profiling data is available. For information about the availability of coverage analysis profiling data, see “Overview of Profiling Methods” on page 355.
- *Count information (line executions)* — To view the total number of times each line (or instruction) was executed, click the **Count View** button (🔍) in the **Profile** window or run the **profilemode count** command in the Debugger command pane. The number of executions is shown to the left of each source line. This feature is only available if coverage analysis profiling data is available. For information about the availability of coverage analysis profiling data, see “Overview of Profiling Methods” on page 355.
- *Count information (function calls)* — To view the total number of times each function was called, click the **Count View** button (🔍) in the **Profile** window or run the **profilemode count** command in the Debugger command pane. The call count for each function is shown to the left of the beginning of each function. This feature is only supported if coverage analysis profiling data is not available and call count profiling data is available. For information about the availability of these data types, see “Overview of Profiling Methods” on page 355.

**Note**

If there is no profiling data for a particular line, a question mark appears to the left of the line instead of actual count or time percentage information.

For more information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

If you profile all tasks in your system, profiling information is also displayed in the Debugger's target list in a column labeled **CPU %**. This column displays the percentage of the CPU that each task and AddressSpace is currently using. For more information, see “Profiling All Tasks in Your System” on page 603.

Performing Range Analyses



Note

This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program, *and* if PC samples are available. For information about the availability of PC samples, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 356.

Improving the performance of an application often requires isolating small portions of large functions that account for the majority of the function's execution time, such as computationally intensive nested loops. MULTI allows you to perform range analyses to collect profiling information for particular sections of code.

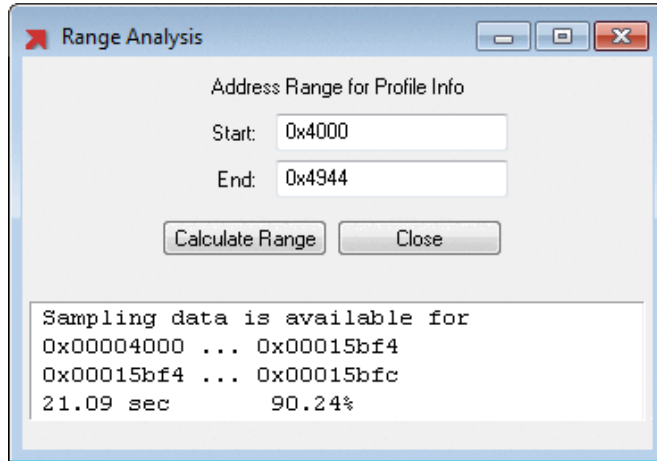
You can perform a range analysis from the **Range Analysis** window or from the Debugger command pane. To open a **Range Analysis** window, click the **Range Analysis** button () in the **Profile** window. To specify the range you want to analyze, enter a hexadecimal **Start** and **End** address in the fields of the **Range Analysis** window.



Tip

In assembly display mode (see “Source Pane Display Modes” on page 23), hexadecimal addresses are displayed to the left of the corresponding instructions in the Debugger source pane.

After you have specified the range of hexadecimal addresses, click the **Calculate Range** button to display the amount of time that elapsed during execution of the specified range, as well as the percentage of the total execution time required to execute the range. The time is displayed in milliseconds, seconds, instructions, or cycles depending on the current configuration (see **Config** → **Time Units** in the **Profile** window).



If no range or an inappropriate range is specified when you click the **Calculate Range** button, the **Range Analysis** window displays an error message.

To perform a range analysis from the Debugger command pane, issue the following command:

```
profilemode range start_addr end_addr
```

where *start_addr* and *end_addr* are the beginning and end of your range, respectively. The result of the analysis appears in the Debugger command pane.

Managing Profiling Data

You can manage collected profiling data in a number of ways: by choosing whether or not to overwrite existing profiling data with new data, by manually dumping and processing your profiling data, and by clearing existing data. Support for some of these capabilities depends on what you are profiling and on the method of profiling. The following sections provide more information.

Adding to or Overwriting Existing Profiling Data

When new profiling data is collected and processed, it can be added to profiling data that already exists, or it can replace old profiling data.

Accumulating data across multiple executions may be useful for getting a more accurate understanding of application performance. This is the default behavior (with one exception*). To append new profiling data to existing profiling data:

- If you are profiling all tasks in your system or if you are profiling a trace-enabled target — In the **Profile** window, click the **Merge new data** button () so that it appears to be pushed down, or enter **profilemode add** in the Debugger command pane.
- If you are profiling a task, AddressSpace, or stand-alone program — Select **Config** → **New Data** → **Added to Old** in the **Profile** window, or enter **profilemode add** in the Debugger command pane. (*Note that you cannot add new data to existing data if you are profiling a task or AddressSpace, are using INTEGRITY version 10 or later, and have established a run-mode connection.)

To replace existing profiling data with new profiling data:

- If you are profiling all tasks in your system or if you are profiling a trace-enabled target — In the **Profile** window, click the **Merge new data** button () so that it does *not* appear to be pushed down, or enter **profilemode replace** in the Debugger command pane.
- If you are profiling a task, AddressSpace, or stand-alone program — Select **Config** → **New Data** → **Replaces Old** in the **Profile** window, or enter **profilemode replace** in the Debugger command pane.

For information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Manually Dumping Profiling Data



Note

This information is only relevant if one of the following is true:

- You are profiling a run-mode task or AddressSpace on an INTEGRITY target, or you are profiling a stand-alone program, *and* you instrumented your code to generate profiling data (that is, you compiled your program with Builder or driver profiling options). For more information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 356.

- You collected PC samples over a freeze-mode connection.

By default, MULTI processes collected profiling data automatically each time your program exits. In some situations, however, you may want to halt your process and dump the collected profiling data at some other point. The most common scenarios for manually dumping profiling data are:

- When you are debugging a process, such as an operating system, that does not terminate.
- When you want to dump profiling data gathered between two points of execution so that you can profile specific areas of execution.



Note

MULTI's ability to dump profiling data depends on command line procedure calls. If you are not using INTEGRITY, interrupts may need to be disabled before command line procedure calls and, consequently, dumps will work. If you need to disable interrupts on your target, first halt your target, and then disable interrupts. After dumping profiling data (explained next), turn interrupts back on and run the program.

To dump all available forms of profiling data (PC samples, call count data, and coverage analysis data), halt your process and then do one of the following:

- Click the **Dump Profiling Info** button ()
- In the Debugger command pane, enter the **profdump** command. For information about the **profdump** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

To dump profiling data gathered between two specific points of execution:

1. Run the program to the first point of execution and then clear any profiling data gathered prior to reaching that point. To clear profiling data, do one of the following:
 - Click the **Clear current profile information** button ()
 - In the Debugger command pane, enter the **profilemode clear** command. For information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

2. Run the program to the second point of execution and then dump the profiling data gathered between the two points.

If automatic data processing is disabled, you must manually process dumped profiling data. For more information, see “Manually Processing Profiling Data” on page 383.

For information about the limitations associated with dumping profiling data, see “Caveats for Dumping and Clearing Profiling Data” on page 385.

Manually Processing Profiling Data



Note

This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program.

Unless you are using INTEGRITY, MULTI is set by default to automatically process and store profiling data when it is dumped. If you want more control over when profiling data is processed, you can disable automatic processing and process your profiling data manually.

To disable the automatic processing of profiling data, do one of the following:

- In the **Profile** window, select **Config** → **Data Processing** → **Manual**.
- In the Debugger command pane, enter the **profilemode manual** command.

After you have disabled automatic processing, you will have to process your profiling data manually. To successfully process profiling data (automatically or manually), you should be connected to your target with the program loaded.



Note

To avoid inadvertently merging old data into a profiling report, MULTI deletes profiling data files after processing them. To save these files, copy the ***.out** files to a separate directory before manually processing profiling data.

To start the manual processing of data, do one of the following:

- Click the **Process Data** button (.
- In the Debugger command pane, enter the **profilemode process** command.

To load a **.pro** file output by **protrans** (see the documentation about the **protrans** utility in the *MULTI: Building Applications* book), perform the following steps:

1. Make sure you are connected to your target and the program is loaded.
2. Copy saved profiling data files to your run directory.
3. Rename ***.pro** files in the format *program.pro*, where *program* is the name of your program.
4. Open the **Profile** window (for instructions, see “Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 359).
5. Enter the **profilemode import** command in the Debugger command pane.
6. Follow the preceding instructions for processing data manually.

To re-enable automatic processing of profiling data, do one of the following:

- Choose **Config** → **Data Processing** → **Automatic**.
- In the Debugger command pane, enter the **profilemode automatic** command.

For information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Clearing Profiling Data

To clear existing profiling data, do one of the following:

- In the **Profile** window, click the **Clear current profile information** button (.
- In the Debugger command pane, enter the **profilemode clear** command. For information about the **profilemode** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

- In the **Profile** window, click the **Close** button (✖) to close the **Profile** window, halt the collection of profiling data, and clear existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)

For information about the limitations associated with clearing profiling data, see “Caveats for Dumping and Clearing Profiling Data” on page 385.

Caveats for Dumping and Clearing Profiling Data

- MULTI uses command line procedure calls to dump and clear profiling data. In some cases command line procedure calls may not be possible or may have adverse effects. If you are using INTEGRITY, command line procedure calls are unsafe for tasks that have been halted while performing a system call. MULTI attempts to detect this situation and prevent command line procedure calls in this context, but it is not always able to do so, nor is it able to detect every situation in which it is unsafe to make command line procedure calls. To guarantee correct operation, you must ensure that the task you are profiling is halted in a safe location prior to dumping or clearing profiling data. For more information about the limitations of command line procedure calls, see “Caveats for Command Line Procedure Calls” on page 306.
- If you are not using INTEGRITY, interrupts may need to be disabled before command line procedure calls and, consequently, dump or clear operations will work. If you need to disable interrupts on your target, first halt your target, and then disable interrupts. After dumping or clearing profiling data, turn interrupts back on and run the program.
- If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program, *and* if you instrumented your code to generate profiling data, dumping or clearing the profiling data may cause the program, task, or tasks inside the AddressSpace to run for the duration of the dump or clear operation. This brief run may cause MULTI to generate PC samples.

Chapter 18

Using Other View Windows

Contents

Viewing Call Stacks	388
Viewing Native Processes	390
Viewing Caches	392

Viewing Call Stacks

The **Call Stack** window displays call stacks (also known as *call stack traces* or simply *stack traces*), which consist of stack frames that are currently active in your program. Each stack frame typically represents a function call. Stack frames are shown in order from most to least recently created.

To open the **Call Stack** window, do one of the following:

- Click the **Call Stack** button () .
- Choose **View** → **Call Stack**.
- In the command pane, enter **callsvi**ew. For information about the **callsvi**ew command, see Chapter 5, “Call Stack Command Reference” in the *MULTI: Debugging Command Reference* book.

The following table describes the buttons available from the toolbar of the **Call Stack** window.

Button	Effect
 Hide Parameters	Toggles the display of parameters in function calls.
 Hide Positions	Toggles the display of the filename and line number of the function call.
 Display stack frames before main()	Toggles whether to display stack frames before <code>main()</code> .
 Copy Window Contents	Copies the contents of the window to the clipboard.
 Freeze Window	Toggles whether the window is refreshed.
 Edit Function	Opens an Editor on the selected function, if it has source code.
 View Locals	Opens a Data Explorer showing all the local variables of the selected function. If the selected function is a C++ instance method, the Data Explorer also shows information about the <code>this</code> pointer.
 Print	Prints the text contents of the Call Stack window to your printer. To print call stack information to the command pane, enter the calls command. For information about the calls command, see Chapter 5, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Close	Closes the Call Stack window. Whether or not this button appears on your toolbar depends on the setting of the option Display close (x) buttons .

At the far right of the toolbar is the **Max Depth** field, which controls the maximum number of stack levels which the window will display. You can change it according to your preference. For example, if you are debugging a program on a very slow target and you only want information about the first few levels of the call stack, you can decrease the number so that the window is refreshed more quickly.

Below the toolbar is the call stack pane, where the call stack is displayed. The following table lists the mouse and keyboard operations you can perform in the call stack pane.

To	Do this
Display a function in the source pane	Click the function
Open an Editor on a function	Double-click the function
Copy the entire call stack pane to the clipboard	Press Ctrl+Shift+C
Search forward in the call stack pane	Press Ctrl+F
Search backward in the call stack pane	Press Ctrl+B

During a debugging session, if you change the attributes of a **Call Stack** window, the changes will affect subsequently created **Call Stack** windows. You can also change these attributes with the **cvconfig** command. For information about this command, see Chapter 5, “Call Stack Command Reference” in the *MULTI: Debugging Command Reference* book.

The Call Stack Window and Command Line Procedure Calls

If a breakpoint is hit during the execution of a command line procedure call, the call stack pane of the **Call Stack** window displays a separator line corresponding to the location where the target was stopped when you made the procedure call. The lower portion is the call stack from before the function call; the upper portion is the call stack starting from the command line procedure call.

The Call Stack Window and Procedure Prologues and Epilogues

At the beginning and end of every procedure are special code sequences called the prologue and epilogue, respectively. These special code sequences save and restore registers and modify the stack pointer. Full source-level debugging is not possible within these regions, so MULTI does not display source-level breakpoints for these lines. You can single-step through this code at the machine level, but many other tasks such as tracing the stack and examining variables are not supported until you are outside this region. The **Call Stack** window may display incorrect information when a process is stopped inside a prologue, an epilogue, or a function called by a prologue or epilogue.

The Call Stack Window and Interrupt/Exception Handlers

Interrupt and exception handlers typically employ nonstandard stack frames. Attempting to generate a stack trace when you are stopped in an interrupt or exception handler may result in the **Call Stack** window displaying incorrect or incomplete information, as well as attempts to access invalid memory addresses on your target.

Viewing Native Processes

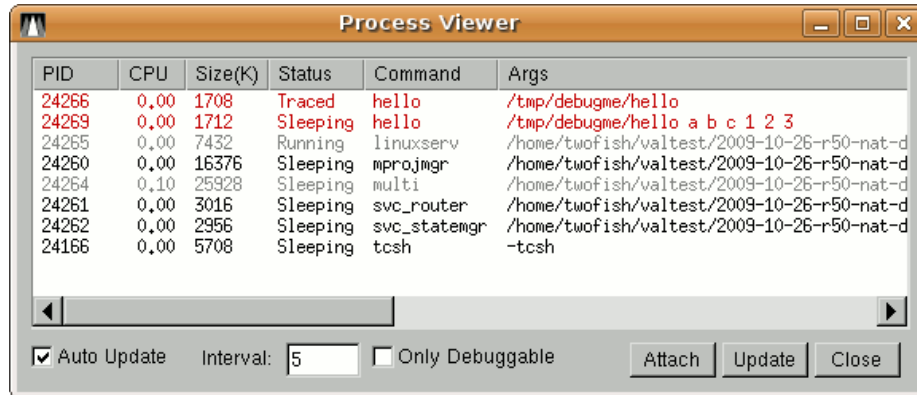
The **Process Viewer** displays a snapshot of the processes on your native Linux or Solaris target, much like the Linux/Solaris `top` and `ps` utilities. The primary purpose of this window is to allow you to attach to native processes. The **Process Viewer** also provides an easy way to identify PIDs for use with certain Debugger commands or dialog boxes.

To open a **Process Viewer**, do one of the following from your native debugging environment:

- Select **View** → **Task Manager** from the Debugger menu bar. (If you are in a non-native environment and in run mode, this menu selection will open a Task Manager instead of a **Process Viewer**.)
- Issue the **top** command from the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

- Issue the command **multi-top** from your shell (see Appendix C, “Command Line Reference” on page 721).

A sample **Process Viewer** for Linux is displayed below:



The **Process Viewer** displays all of the processes that you own, color-coded depending on their type, where:

- Red — Indicates a process that carries debugging information and contains source code that can be debugged.
- Gray — Indicates a process or child process of the MULTI Debugger. Attaching to this process will cause MULTI to stop functioning correctly.
- Black — Indicates a process that does not have available debugging information.

To attach to a process and debug it, either select the process and click **Attach**, or simply double-click the process. If the Debugger cannot locate the executable image associated with the process, you will be prompted to choose it from the disk.

If the **Process Viewer**'s contents become out of date and you want to see the current list of processes available on your native target, simply click **Update** to refresh the list. You can also enable automatic periodic refreshing by selecting the **Auto Update** check box and entering an integer in the **Interval** text box, which specifies how often (in seconds) updates should occur. The default interval is 5 seconds.

Viewing Caches

MULTI includes two display windows that allow you to view the contents of the caches on your processor. You can use the **Cache View** and **Cache Find** windows to browse cache contents and search all of the caches at a specific address.



Note

Support for cache viewing and searching varies according to processor type. Not all processors support these features, and availability also depends on your debug server connection.

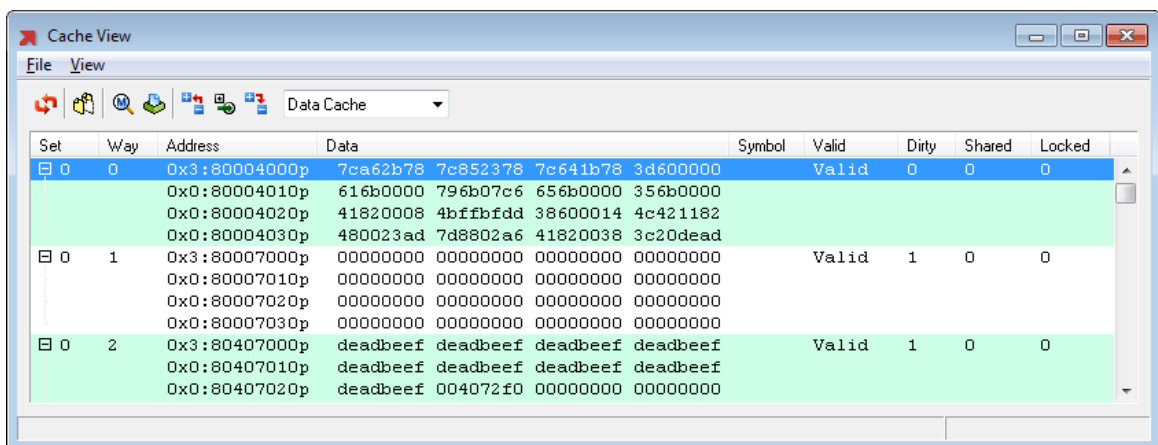
The Cache View Window

The **Cache View** window displays all of the entries from a single cache. You can use this window to browse through the cache data to see what addresses and data are in the cache, what addresses are in the cache but have been invalidated, and other information.

To open a **Cache View** window, do one of the following:

- Select **View** → **Caches**.
- Enter the **cacheview** command in the Debugger command pane. For more information about this command, see “Cache View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Cache View** window is shown next.



The drop-down box on the toolbar allows you to select which cache to view.

By default, the **Cache View** window hides invalid cache entries. You can change this default behavior by clicking the  button.

The columns in the **Cache View** window also vary depending on your processor and cache type. The following table describes all possible columns, which are ordered alphabetically below. See your processor's manual for more information about the data in the caches.

Column	Description
A0, A1	Displays the allocate bit for the first/second cache block of this cache entry.
Address	Displays the address of the cache line. This address may be a virtual or physical address, depending on your processor, cache and whether the memory management unit is enabled.
D0, D1	Indicates whether the first/second half of the cache line is dirty. Each dirty bit corresponds to 32 bytes of cache data.
Data	Displays the data associated with this cache line. The number of bytes displayed depends on your processor and cache type.
Data Locked	Indicates whether the cache line has been locked for data.
Dirty*†	Indicates whether the cache line is dirty. On ARM920/922, ARM946, and ARM1136, each dirty bit corresponds to 16 bytes of cache data. On PowerPC 440, each of the 4 bits corresponds to a double word of data in the cache line.
Instruction Locked	Indicates whether the cache line has been locked for instructions.
Lock, Locked	Indicates whether the cache line has been locked.
LRU	On Intel XScale IXP2350, indicates which way of the cache line is next in line for replacement on a line eviction. The following list provides the mapping of LRU bit values to cache ways: • 01x — Way 0 • 11x — Way 1 • x00 — Way 2 • x01 — Way 3 On PowerPC 405, indicates whether the way will be the next out of 2 available ways to be replaced at line eviction.

Column	Description
MESI0, MESI1	Displays the <code>MESI</code> bits for the first/second cache block of this cache entry.
N	Indicates whether the cache line has been loaded incoherently. If set, only instructions hit in this line (data accesses will miss).
Set	Displays the set of the cache line. Each address maps to a single set in the cache.
Shared[‡]	Indicates whether the cache line is shared in a multiprocessor system.
Stale	Indicates whether the cache line is stale (that is, whether the data is invalid).
State	Displays the <code>MOESI</code> state for the cache line. Available states are: • 0 — Invalid • 1 — Shared • 3 — Owned • 5 — Exclusive • 7 — Modified
S[X]:Alloc	Displays the allocate bit for cache block <i>x</i> .
S[X]:MESI	Displays the <code>MESI</code> bits for cache block <i>x</i> of this cache entry.
SX:Parity	Displays the parity bit for cache block <i>x</i> .
Symbol	Displays the symbol associated with the address at the start of the cache line.
TD	Displays the TID disable field for the memory page associated with this cache entry.
TERA	Displays the tag extended real address (TERA) of this cache entry. This is the top 4 bits of the physical address associated with this cache entry.
TID	Displays the translation ID field for the memory page associated with this cache entry.
TS	Displays the translation space for the memory page associated with this cache line.
Used	Indicates whether the cache line has been recently used.
User	Displays the 4 user bits associated with this cache entry.
V0, V1	Indicates whether the first/second half of the cache line is valid. Each valid bit corresponds to 32 bytes of cache data.
Valid^{*‡}	Indicates whether the cache line is valid.

Column	Description
Way	Displays the way of the cache line. Any address that maps to the current set can be found in any way within that set. This column does not apply for direct-mapped caches.

*: On PowerPC 6xx, 7xx, 51xx, 52xx, 7400, 7410, 82xx, and 83xx, MESI states are encoded in the **Valid** and **Dirty** bits as follows:

State	Valid	Dirty
Modified	1	1
Exclusive	1	0
Shared	0	1
Invalid	0	0

‡: On PowerPC 744x, 745x, 85xx, 86xx, and QorIQ, MESI states are encoded in the **Valid**, **Dirty**, and **Shared** bits as follows:

State	Valid	Dirty	Shared
Modified	1	1	0
Exclusive	1	0	0
Shared	1	0	1
Invalid	0	x	x

You can sort the data in the **Cache View** window by clicking the column header above the column that you want to sort by. You can also change the display, refresh the window, or open other windows using the buttons described in the following table.

Button	Effect
	Refreshes the current view by reloading cache data from the target.
	Hides or shows all invalid cache entries. When the button is pushed down, invalid caches entries are not displayed. Otherwise, all cache entries, including invalid entries, are visible.
	Opens a Memory View window at the address of the selected cache line.
	Opens a Cache Find window at the address of the selected cache line. See “The Cache Find Window” on page 396.

Button	Effect
	Contracts all entries in the Cache View window so that each cache line takes up a single line in the cache list.
	Expands all valid entries and contracts all invalid entries in the Cache View window so that each valid cache line is fully displayed and each invalid entry takes up a single line.
	Expands all entries in the Cache View window so that each cache line is fully displayed.

The Cache Find Window

The **Cache Find** window allows you to view the contents of all of your processor's caches, as well as the memory underneath the caches, at a specific address. To open a **Cache Find** window, do one of the following:

- Select **View** → **Find Address in Cache**.
- In the Debugger command pane, enter the **cachefind** command. For information about this command, see “Cache View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the **Cache View** window, click .

The **Cache Find** window is shown next.

Cache Find - 0x380004000					
Address: 0x380004000 Go					
Instruction Cache:					
	Set	0	Way	0	Valid
Locked: 0					
0x3:80004000p:	7ca62b78	7c852378	7c641b78	3d600000	
0x3:80004010p:	616b0000	796b07c6	656b0000	356b0000	
0x3:80004020p:	41820008	4bffbffd	38600014	4c421182	
0x3:80004030p:	480023ad	7d8802a6	41820038	3c20dead	
Data Cache:					
	Set	0	Way	0	Valid
Dirty: 0					
Shared: 0					
Locked: 0					
0x3:80004000p:	7ca62b78	7c852378	7c641b78	3d600000	
0x3:80004010p:	616b0000	796b07c6	656b0000	356b0000	
0x3:80004020p:	41820008	4bffbffd	38600014	4c421182	
0x3:80004030p:	480023ad	7d8802a6	41820038	3c20dead	
L2 Cache:					
	Set	256	Way	0	Valid
Dirty: 0					
Shared: 0					
Locked: 0					
N: 1					
0x3:80004000p:	48000000	48000000	48000000	48000000	
0x3:80004010p:	48000000	48000000	48000000	48000000	
0x3:80004020p:	48000000	48000000	48000000	48000000	
0x3:80004030p:	48000000	48000000	48000000	48000000	
Raw Memory:					
0x3:80004000p:	7ca62b78	7c852378	7c641b78	3d600000	
0x3:80004010p:	616b0000	796b07c6	656b0000	356b0000	
0x3:80004020p:	41820008	4bffbffd	38600014	4c421182	
0x3:80004030p:	480023ad	7d8802a6	41820038	3c20dead	

The **Cache Find** window displays the valid line that contains the specified address for each cache. If the address is not found in a cache, the set that was searched will be displayed and the window will indicate that the address was not found in that cache. If a cache line is found that contains the address, the cache tags associated with the cache line are displayed along with the data from the cache line.

The data in the **Cache Find** window is colored to indicate the state of the data. This makes the display easy to read and can help you to find cache coherency problems. A cache coherency problem occurs when there is data in the cache that is clean (not dirty) and it differs from data in memory or a cache further from the processor. In this case, the data that makes up the cache coherency conflict is colored in red. In addition, data in memory that has a dirty cache line associated with it is colored gray.

To refresh the cache data in the **Cache Find** window, click . To change the address to search for, enter a new address in the **Address** field and click **Go**.

To freeze the cache data in the **Cache Find** window so that it does not update when the target changes state, click . (The window is frozen when this button appears pushed down. Click the button again to unfreeze the window.)

To open a **Memory View** window at the specified address, click . (See Chapter 15, “Using the Memory View Window” on page 323 for information about using this window.)

To open a **Cache View** window, click .

Part IV

TimeMachine Debugging

Chapter 19

Analyzing Trace Data with the TimeMachine Tool Suite

Contents

Quick Start	403
Overview of Trace Analysis Tools	404
Managing Trace Data	405
The TimeMachine Debugger	413
The PathAnalyzer	424
Viewing Trace Data in the Trace List	434
Bookmarking Trace Data	444
Searching Trace Data	446
Saving and Loading a Trace Session	448
Browsing Trace Data	451
Viewing Trace Statistics	460
The TimeMachine API	466
Viewing Trace Events in the EventAnalyzer	474
Using Trace Data to Profile Your Target	475
Viewing Reconstructed Register Values	476

The MULTI Debugger is capable of collecting and storing trace data from supported trace-capable targets. MULTI and TimeMachine currently support trace collection and analysis with the following architectures:

- ARM
- Power Architecture
- 68K/ColdFire
- Renesas V850E
- MIPS

There are a number of interfaces that you can use to collect trace data, including:

- ARM Embedded Trace Macrocell
- PowerPC 405 and 440 built-in trace port
- PowerPC 55xx Nexus trace port
- Freescale MAC71xx Nexus trace port
- ColdFire built-in trace port
- Renesas V850E IECUBE
- V850E RTE
- Green Hills Simulators for ARM, MIPS, Power Architecture, V800, 68K/ColdFire

You can use the Green Hills SuperTrace Probe to collect trace data from any of the trace ports listed above. No additional hardware is required for collecting trace data from the simulators listed above.

Once you have collected trace data, you can analyze it with an array of tools provided in the MULTI Debugger and with the powerful TimeMachine tool suite. The TimeMachine tool suite expands MULTI's standard trace analysis capabilities to allow you to step and run backward through time, examine process flow over significant periods of time, analyze high-level execution patterns, and debug RTOS applications from trace data. This chapter describes how to collect trace data and how to analyze it with the TimeMachine tool suite and with the standard tools that ship with the MULTI Debugger.

**Note**

TimeMachine is licensed separately from MULTI. For information about purchasing a TimeMachine license, contact your Green Hills sales representative.

Quick Start

When you establish a trace-enabled debug connection, trace collection is automatically enabled. At any point when the target is halted, you can use the TimeMachine run-control buttons (, , , and ) to run or step backwards through the trace data as though you were debugging your code while it executed. For example, if your target hits an unexpected breakpoint, you can simply step backwards through the code to determine what caused the breakpoint to be hit. If your target supports data trace, you can even examine variables in the Data Explorer as you step backwards. For information about the TimeMachine run-control buttons, which are disabled if you do not have a TimeMachine license, see “TimeMachine Run-Control Buttons” on page 416.

In addition to making it possible to run and step backwards, trace data enables many other powerful tools, which are described in the next section. To use any of these tools, you must retrieve trace data from your trace-enabled probe or target. Trace data is automatically retrieved when you click one of the TimeMachine run-control buttons. You can also retrieve it at any time by selecting **TimeMachine** → **Retrieve Trace** or by clicking the **Retrieve Trace** button () located in the Trace List or PathAnalyzer. In addition, you can configure MULTI to automatically retrieve trace data each time your target halts. See the description of **Retrieve trace when target halts** in “The Collection Tab” on page 481.

Overview of Trace Analysis Tools

Once you have collected trace data, you can analyze it with any of the TimeMachine or trace tools listed next.

TimeMachine is a separately licensed tool suite that dramatically extends MULTI's trace analysis and debugging capabilities. The TimeMachine tool suite includes the following trace analysis tools and features:

- *TimeMachine Debugger* — Utilizes trace data to allow you to step backward in time while you are debugging. The TimeMachine Debugger is similar to the MULTI Debugger window and supports most of the main Debugger's features, so you can continue to use breakpoints, view registers and memory, and examine the call stack as you move backward and/or forward through the process. TimeMachine makes it easy to quickly isolate and debug problems that are otherwise hard to reproduce. For more information, see “The TimeMachine Debugger” on page 413.
- *PathAnalyzer* — Displays the sequence of function calls that occurred in call stack order. The PathAnalyzer allows you to examine the flow of execution at a high level or zoom in to see individual function calls at a low level. You can also search for sequences of events that point out anomalies such as a failure to meet a real-time deadline.
- *TimeMachine API* — Enables you to write custom analysis tools to solve problems that may not be easily solved using the standard tool suite. This API offers a DLL (Windows) or a shared object (Linux/Solaris) that allows you to write a program or script that can interface with trace data to track down any type of problem. Examples of problems that have been investigated with the TimeMachine API are high power consumption, cache coherency problems, and illegal memory writes. For more information, see “The TimeMachine API” on page 466.
- *EventAnalyzer Integration* — Displays a time line of task activity, OS API calls, and system events. For more information, see “Viewing Trace Events in the EventAnalyzer” on page 474.
- *MULTI Profiling Integration* — Allows you to convert trace data into performance analysis data. Profiling data generated from trace data is much more complete, in terms of which instructions are represented, than profiling data generated by profiling methods that use sampling. For more information, see “Using Trace Data to Profile Your Target” on page 475.

Both the MULTI Debugger and the TimeMachine tool suite include the following standard trace analysis tools:

- *Trace List* — Allows you to view and explore trace data at the function and assembly levels and control trace collection. You can also access advanced trigger, search, filter, and bookmark controls, in addition to various trace data analysis tools. For more information, see “Viewing Trace Data in the Trace List” on page 434.
- *Trace Browsers* — Allow you to quickly locate events in your trace data by finding similar events, such as instructions and memory accesses. For more information, see “Browsing Trace Data” on page 451.
- *Trace Statistics* — Calculates and displays statistical information about trace data. For more information, see “Viewing Trace Statistics” on page 460.

Managing Trace Data

Trace collection, retrieval, and deletion is highly configurable. By default, trace collection is automatically enabled, and collected trace data is automatically retrieved when you start using the TimeMachine Debugger by stepping or running backwards. Collected trace data is also retrieved automatically when the program being debugged exits, and it can be retrieved manually at any time. MULTI keeps the trace data as long as the accumulated trace data files do not exceed the size specified.

When trace configuration options are set to their defaults, it is usually possible to use TimeMachine to step or run backward in time from the current location. This default trace configuration presents a convenient way to use trace data, but sometimes you may require more precise control over when trace data is collected, retrieved, and discarded. The following sections describe the many ways that you can manage trace data in MULTI.

Enabling and Disabling Trace Collection

By default, MULTI automatically enables trace collection when you connect to a trace-capable target. To disable this default behavior, clear the **Automatically enable trace collection** option, which appears on the **Collection** tab of the **Trace Options** window. To open the **Trace Options** window, select **TimeMachine** →

Trace Options from the Debugger. For more information, see “The Trace Options Window” on page 480.

You can also manually enable and disable trace collection. When you manually enable trace collection, MULTI clears any previously collected data on the target. Data that has already been retrieved is not cleared, but if trace retrieval is currently in progress, it is aborted. To manually enable trace collection, do one of the following:

- In the MULTI Debugger, select **TimeMachine** → **Enable Trace**.
- In the Trace List or the PathAnalyzer, click the **Enable Trace** button () so that it appears to be pushed down.
- In the Trace List or the PathAnalyzer, select **File** → **Enable Trace**.
- In the Debugger command pane, enter the **trace enable** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

To manually disable trace collection, do one of the following:

- In the MULTI Debugger, select **TimeMachine** → **Disable Trace**.
- In the Trace List or the PathAnalyzer, click the **Disable Trace** button () so that it does not appear to be pushed down.
- In the Trace List or the PathAnalyzer, select **File** → **Disable Trace**.
- In the Debugger command pane, enter the **trace disable** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

When you disable trace collection, all the trace data currently stored on the trace collection device is retrieved.

You can also control trace collection via trace triggers. For information about trace triggers, see “Configuring Trace Collection” on page 489.

Collecting Operating System Trace Data

On the INTEGRITY, velOSity, and u-velOSity operating systems, TimeMachine can record the operating system tasks and AddressSpaces that were executed. On

INTEGRITY, it can also filter out operating system AddressSpaces that you are not interested in tracing, provided that your BSP supports this.

**Note**

Even if you have one of these operating systems and hardware trace support, the operating system trace features may still be unavailable. For fully functional use of TimeMachine with INTEGRITY, velOSity, or u-velOSity, special hardware that records the process ID (PID) register must be made available. Additionally, not all versions of these operating systems support these features. For updated information about which hardware and operating system versions support these enhanced features, contact your Green Hills sales representative.

**Note**

Some target processors limit the number of unique AddressSpaces that can be traced. Tracing a target on which the number of AddressSpaces exceeds this limit is not supported.

MULTI uses the state of the operating system at the time when trace data is retrieved for trace decoding. If a task or AddressSpace was traced, but is no longer present on the target when trace data is retrieved, MULTI will be unable to decode the trace data for that task or AddressSpace. Additionally, MULTI may not be able to decode some or all of the subsequent trace data.

On INTEGRITY, establishing a run-mode partner connection alongside your freeze-mode connection allows you to specify which AddressSpaces are traced. By not tracing the entire system, you can more efficiently use your limited trace buffer. A common use for tracing specific AddressSpaces is to disable trace collection for any idle tasks you may have on your system. Typically, idle tasks take up a large amount of trace buffer space or time and do not contain very interesting data. For information about establishing a run-mode partner, see “Automatically Establishing Run-Mode Connections” on page 69.

After you have established a run-mode partner, you can enable and disable the tracing of specific AddressSpaces by right-clicking a task in the target list and selecting **Trace**. This toggles trace collection for the AddressSpace that encapsulates the selected task. Once you have toggled trace collection for an AddressSpace, any new AddressSpaces that are created will not be traced until you enable tracing of them.



Note

If you manually start your run-mode connection, the **Trace** menu entry may not be available until MULTI offers to partner your connections the first time you halt your freeze-mode connection or retrieve trace data.



Note

Interrupt and exception handlers in the kernel are traced if the interrupt or exception occurs while executing in a traced AddressSpace.

Retrieving Trace Data

By default, MULTI automatically retrieves trace data when you first step or run backwards, when you enable TimeMachine, or when the program being debugged exits. You can also configure MULTI to automatically retrieve trace data when the target halts:

1. In the Debugger, select **TimeMachine** → **Trace Options** to open the **Trace Options** window.
2. Select the **Retrieve trace when target halts** option, which appears on the **Collection** tab of the **Trace Options** window.

You can manually retrieve trace data by doing one of the following:

- In the Debugger, select **TimeMachine** → **Retrieve Trace**.
- In the Debugger command pane, enter the **trace retrieve** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Trace List or PathAnalyzer, click the **Retrieve Trace** button () at any time.

To abort the retrieval of trace data, do one of the following:

- In the Debugger, select **TimeMachine** → **Abort Trace Retrieval**.
- In the Debugger command pane, enter the **trace abort** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

- In the Trace List or PathAnalyzer, click the **Abort Trace Retrieval** button () so that it no longer appears to be pushed down.

You can also control trace retrieval via trace triggers. For more information about trace triggers, see “Configuring Trace Collection” on page 489.

Retrieving Operating System Trace Data

TimeMachine may need to halt the kernel in order to retrieve system information that aids in trace analysis. Halting the kernel allows MULTI to associate trace data with corresponding AddressSpaces and tasks. The amount of time that the system is halted varies depending on your hardware. If TimeMachine needs to halt the system, a warning dialog box asking for your permission appears. If you decide not to halt the system, MULTI may abort trace retrieval upon encountering instructions from an AddressSpace it is unaware of, thereby making much of the trace analysis unavailable. See also the **Assume static OSA** trace option in “The Trace Options Window” on page 480.

Retrieving Trace Data from a SuperTrace Probe v3

Green Hills SuperTrace Probes can collect a large amount of trace data. Retrieving 4 GB of trace data from a SuperTrace Probe v3 and decoding it takes at least five minutes and in many cases may take much longer. In some cases, it may take an hour or more. Decoding speed depends on many factors including your trace configuration, the type of target being traced, the code traced, the performance of the host machine, and local network conditions.

In many cases, the most interesting trace data is the most recent data near the end of the trace buffer. With SuperTrace Probe v3, all available trace RAM is always used regardless of the **Target buffer size** setting (see “The Collection Tab” on page 481). However, it is still possible to step or run backwards quickly because MULTI quickly retrieves the configured **Target buffer size** from the end of the SuperTrace Probe's trace buffer. You can manually retrieve more data later if the originally configured amount is insufficient.

If you have already retrieved some of the trace data, do one of the following things to retrieve twice as much:

- In the Debugger, select **TimeMachine** → **Retrieve Trace**.

- In the Debugger command pane, enter the **trace retrieve** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Trace List or PathAnalyzer, click the **Retrieve Trace** button (.

To choose how much data is retrieved:

- In the Trace List or PathAnalyzer, click and hold the **Retrieve Trace** button (). In the drop-down menu that appears, choose how much data to retrieve.

To retrieve all of the trace data from the SuperTrace Probe, do one of the following:

- In the Trace List or PathAnalyzer, click and hold the **Retrieve Trace** button (). In the drop-down menu that appears, choose to retrieve all of the trace data.
- In the Debugger command pane, issue the **trace retrieve -all** command.



Note

If you retrieve more trace data than is configured by **Target buffer size**, all of the data in the tools is cleared and retrieved from the SuperTrace Probe again. This also ends TimeMachine Debugger sessions and clears all bookmarks.

Discarding Trace Data

Trace collection devices such as the SuperTrace Probe are capable of quickly collecting very large amounts of trace data. The trace data files become even larger when they are retrieved by your PC, decompressed, and indexed. Because MULTI always appends new trace data to the end of older trace data, the amount of disk space used to store trace data can be very large. When the size of the accumulated trace data files exceeds the **Host buffer size** setting, MULTI discards old trace data. (For more information about the **Host buffer size** setting and other trace configuration settings, see “The Trace Options Window” on page 480.)

MULTI attempts to discard unnecessary trace data by basing its deletion on the age of the trace data and the locations of triggers, bookmarks, and the current selection. As trace data is discarded over time, you may encounter side effects such as gaps in the timestamps displayed in the Trace List and Path Analyzer.

Sometimes you may want to discard all your trace data and start trace collection over again. When you perform one of the following operations, MULTI clears all current trace data on the host, trace probe, and target:

- In the Debugger, select **TimeMachine** → **Clear Data**.
- In the Trace List or the PathAnalyzer, select **File** → **Clear Data**.
- In the Debugger command pane, enter the **trace clear** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

Dealing with Incomplete Trace Data

An ideal trace target (such as a Green Hills instruction set simulator) generates trace data that includes complete information about every instruction and data access that occurs on the target. When used with such an ideal trace target, the TimeMachine Debugger is capable of determining the value of almost every register and memory location used by your program at any point during its execution. With an ideal trace target, the behavior of the TimeMachine Debugger is almost identical to that of the MULTI Debugger on a live target. The only differences are that you can run and step backwards in the TimeMachine Debugger but cannot modify registers or memory.

Unfortunately, most hardware devices compromise on trace capabilities because of price, power, pin count, and performance constraints. Data trace is often one of the first things to be sacrificed. Some architectures, such as PowerPC 4xx, do not support tracing data accesses at all. Other architectures, such as ColdFire, trace values that are loaded and stored to memory but do not trace the associated memory addresses. ARM ETMv3 targets typically drop data trace packets when on-chip trace buffers begin to fill. Very few devices support full data trace and have the trace port bandwidth to output data trace without dropping packets when the processor is running from cached memory. Some devices are capable of stalling the processor when necessary to allow the trace port to keep up, but even this cannot always prevent overflows.

Most trace tools, such as the PathAnalyzer, **Profile** window, and EventAnalyzer, are unaffected by incomplete data trace; however, data trace is very important for the TimeMachine Debugger. Incomplete data trace can prevent the TimeMachine Debugger from determining the values of registers, memory, and, by extension,

local and global variables. Values that the TimeMachine Debugger cannot determine given the available trace data are displayed as <<Unreadable/Unknown>>. Incomplete data trace can also prevent read/write hardware breakpoints from being hit.

Some trace architectures also drop instruction trace when on-chip trace buffers overflow. MULTI attempts to fill in the missing instruction trace by inferring it from the trace data collected before and after the overflow (see the option **Attempt to reconstruct gaps in trace data** in “The Trace Options Window” on page 480). Unfortunately, reconstruction is not always possible, and incomplete instruction trace impacts all trace analysis tools. It can cause discontinuities in the PathAnalyzer call stack, the exclusion of events from the EventAnalyzer, and problems when running and stepping in the TimeMachine Debugger. In the Trace List window, missing instruction trace is indicated by a line with the text **FIFO Overflow**. If the TimeMachine Debugger encounters missing instruction trace while running or stepping, it stops and prints:

```
Stopped by discontinuity in trace data
```

to avoid skipping over breakpoints that may have been set on instructions that were not traced.

Many trace architectures can be configured to only generate instruction trace data of certain functions or when certain conditions are met. All of the MULTI trace tools attempt to deal with the sparse trace data that results from this trace suppression, but as more instruction trace is disabled, the usefulness of high-level tools such as the TimeMachine Debugger and the PathAnalyzer degrades rapidly. If instruction trace is suppressed to the point that only short bursts of instructions are traced, the TimeMachine API and the instruction pane in the Trace List are likely to be the only useful trace analysis tools. In the Trace List window, suppressed instruction trace is indicated by a line with the text **Trace Disabled**.

Most trace architectures provide some mechanism for configuring when data accesses are traced or for completely disabling data trace. You can control this through the **Set Triggers** window (see “The Set Triggers Window” on page 493). Limiting or disabling data trace can prevent instruction trace from being lost. You may therefore want to disable data trace if you are primarily using trace data for performance analysis or for the PathAnalyzer, where data trace is not important. If you are collecting trace data over Nexus, you may alternatively enable the option **Stall Processor to Avoid Overflows** to prevent instruction trace from being lost. For

information about this option, see the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide*.

It is also possible for trace data to include incomplete context switch markers. For example, on some INTEGRITY targets, the raw trace data includes sufficient information for MULTI to determine the active AddressSpace, but not enough information for it to determine the active task. On some architectures, MULTI attempts to infer the active task by leveraging data trace and knowledge of the kernel. In this case, missing data trace may prevent MULTI from determining the task associated with some context switches. Such context switches are marked as switching into an unknown task. On other architectures, debug builds of the kernel include special instrumentation that allows MULTI to extract active task information from the trace data. In some cases, active task information is not available and trace data for all tasks in the same AddressSpace is lumped together. In other cases, even active AddressSpace information may not be available due to incomplete trace data.

The TimeMachine Debugger

With TimeMachine, you can step and run backward through your software, replaying the execution of your trace data through the familiar Debugger interface. The TimeMachine Debugger is completely integrated with the MULTI Debugger so that you can also run and step forward and use most other features available in the standard Debugger.

There are certain operations, however, that cannot be performed in a TimeMachine Debugger (including writing registers and memory and invoking command line procedure calls) and certain features for which support in TimeMachine is dependent on MULTI's ability to reconstruct register and memory values.

If your trace data includes information about data accesses, MULTI uses that information to infer as much as possible about the state of the target at each point recorded in the trace log. If you have a complete log of the address and value of every data access, TimeMachine is usually able to completely reconstruct the state of all registers and memory accessed by your program. This allows features such as the Data Explorer, the **Register View** window, and the **Memory View** window to be used in the TimeMachine Debugger just as they are used in the standard MULTI Debugger.

However, if your trace log does not contain a complete log of the address and value of every data access, or if the trace log does not go back far enough to include necessary data accesses, MULTI may be unable to reconstruct some parts of the target state. Values that MULTI cannot infer from the available trace data are displayed as << **Unreadable/Unknown** >>. For more information, see “Dealing with Incomplete Trace Data” on page 411.

**Note**

MULTI cannot infer register and memory values from trace data collected on a MIPS target.

Enabling and Disabling TimeMachine

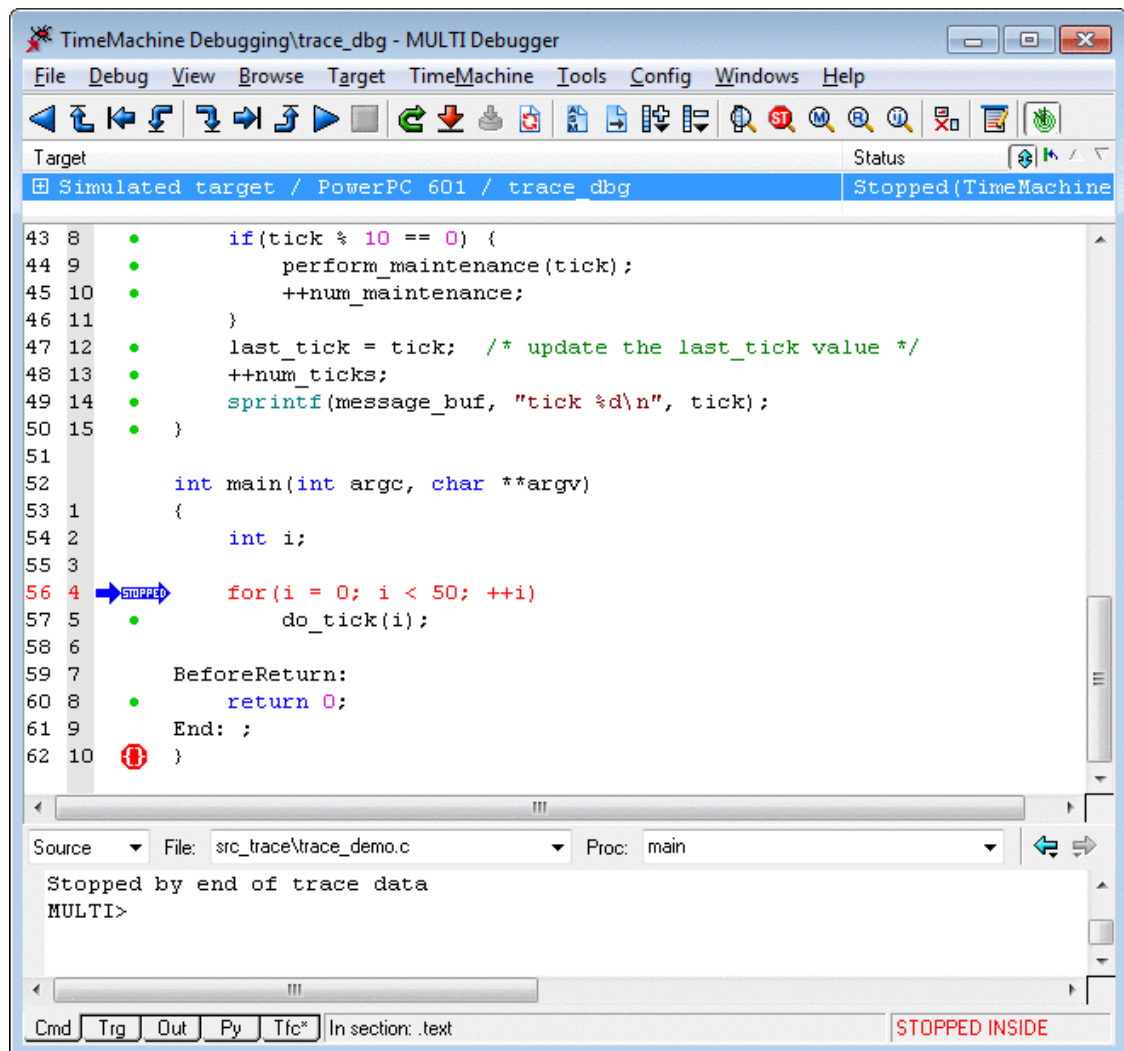
TimeMachine can only be used after you have collected trace data. (For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.) After you have run your program and collected trace data, you can launch the TimeMachine Debugger on selected target list entries by doing one of the following:

- In the Debugger window, click the **TimeMachine Debugger** button () or select **TimeMachine** → **TimeMachine Debugger**.
- In the Debugger window, click any one of the TimeMachine run-control buttons (, , , or ) to step or run backward in TimeMachine. For more information, see “TimeMachine Run-Control Buttons” on page 416.
- In the Debugger command pane, enter the **timemachine** command. For more information, see the **timemachine** command in Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Trace List or the PathAnalyzer, double-click an instruction or a function, respectively.
- In the Trace List or the PathAnalyzer, select **Tools** → **TimeMachine Debugger**.

The following things happen to indicate that you are in TimeMachine mode:

- The PC pointer turns blue () (It is red otherwise.)
- All run-control buttons turn blue.
- The **TimeMachine Debugger** button () appears to be pushed down.

- The **Status** column in the target list indicates that you are in TimeMachine mode.



To disable TimeMachine mode for selected target list entries, do one of the following:

- In the Debugger window, click the **TimeMachine Debugger** button (🔍) or select **TimeMachine** → **TimeMachine Debugger**.
- In the Debugger command pane, enter the **timemachine** command. For more information, see the **timemachine** command in Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

The Location of the Program Counter in TimeMachine

When you first launch the TimeMachine Debugger, the blue program counter arrow () indicates the last instruction recorded during the trace. You can change which instruction the TimeMachine Debugger begins at by selecting the instruction in the Trace List or by selecting a function in the PathAnalyzer (for more information, see “Viewing Trace Data in the Trace List” on page 434 and “The PathAnalyzer Window” on page 425).

If you enter TimeMachine by running or stepping backwards, TimeMachine begins running backwards from the end of the trace data and not from a previous position of TimeMachine.

TimeMachine Run-Control Buttons

The four buttons that are unique to the TimeMachine Debugger window are described in the following table. (The rest of the buttons and the window function similarly to a regular Debugger window. For more information, see “The Debugger Window Toolbar” on page 691 and Chapter 2, “The Main Debugger Window” on page 11.)



Tip

You can also issue TimeMachine Debugger commands in the command pane to move backward through your code. The command equivalents for each button are given in the following table. For more information about these commands, see Chapter 13, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

Button	Command equivalent	Description
	bc	Runs backward to the previous breakpoint or, if no previous breakpoint exists, to the first instruction in the trace data.
	bcU	Steps back up to the caller of the current function.
	bprev	Steps back one line. If the TimeMachine Debugger is in source display mode, clicking this button causes the TimeMachine Debugger to step back a single source line. If the TimeMachine Debugger is in an assembly mode, it steps back a single machine instruction. For more information about display modes, see “Source Pane Display Modes” on page 23.

Button	Command equivalent	Description
	bs	Steps back one line, stepping into a function if the previous line is in a different function. For more information, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

If running or stepping backward in the TimeMachine Debugger brings you to an unexpected location in your source code, you can issue the **trace history -** command to undo the run or step operation. To redo the operation, enter the **trace history +** command. For information about the **trace** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book. For information about adding toolbar buttons that perform the same operations as the **trace history -** and **trace history +** commands, see “Adding, Removing, and Rearranging Toolbar Buttons” on page 698.

Breakpoint Sharing

Because TimeMachine is so closely integrated with the MULTI Debugger, TimeMachine and the live target share breakpoints. That is, if you set a breakpoint while using TimeMachine and then disable TimeMachine, the breakpoint is applied to the real target. Breakpoint sharing works both ways, so if you set a breakpoint on the target and then launch TimeMachine, the breakpoint is also present in TimeMachine.

An unlimited number of hardware breakpoints can be set in TimeMachine, but only a target-specific number on the live target. If you set hardware breakpoints in TimeMachine mode and then disable TimeMachine, MULTI applies as many of those hardware breakpoints to the live target as are supported; the rest are retained but disabled.

For information about setting, hitting, or sharing breakpoints while using TimeMachine with an OS or while using Separate Session TimeMachine, see “Using TimeMachine with OS Tasks” on page 418, “OSA Breakpoints in TimeMachine” on page 420, or “Using Separate Session TimeMachine” on page 422.

Using TimeMachine with OS Tasks

When you retrieve INTEGRITY, velOSity, or u-velOSity trace data for the first time, a list of the tasks on the system appears in the target list.

The first time you want to launch TimeMachine on a task, you must launch it manually. After you have launched TimeMachine on the first task, TimeMachine may be automatically launched on other tasks if hardware breakpoints are hit in those tasks while TimeMachine is running. To launch TimeMachine on a task, do one of the following:

- Select the task in the target list and then click the **TimeMachine Debugger** button () or any one of the TimeMachine run-control buttons (, , , or ) . For information about the run-control buttons, see “TimeMachine Run-Control Buttons” on page 416.
- Select the task in the target list and then enter the **timemachine** command in the Debugger command pane.
- In the Debugger command pane, enter **timemachine -tid *task_id*** to launch TimeMachine on the specified task.
- In the Debugger command pane, enter **timemachine -as_name *AddressSpace*** to launch TimeMachine on the first task in the specified AddressSpace.

For more information about the **timemachine** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

You may notice that even when you launch TimeMachine from a run-mode task, TimeMachine is associated with the corresponding freeze-mode task in the target list. This is because trace data is retrieved from the freeze-mode connection.



Note

You can only launch TimeMachine on tasks that exist on the target. If a task has been removed or unloaded from the target, you will not be able to launch TimeMachine on it even if it has trace data associated with it. You can work around this behavior by saving the trace data to a file and then loading it back into MULTI (see “Saving and Loading a Trace Session” on page 448). Once the trace file is loaded, TimeMachine is automatically launched on all tasks that were traced.

After TimeMachine has been launched on two or more tasks, the mode of these tasks is synchronized. As a result, if you disable TimeMachine mode for one of the tasks, it is also disabled for the other(s). Similarly, if you enable TimeMachine again for one of the tasks, it is also enabled for the other(s). Tasks on which you never launched TimeMachine are unaffected by these actions.

Once TimeMachine has been launched on several tasks, you can step or run forward or backward from any of them. If you do so from a task that is not the currently executing one, the step/run operation begins from the location of the blue program counter arrow () in the currently executing task. The end location of a successful step is relative to the location of the blue program counter displayed in the selected task.

If you set a hardware breakpoint on a virtual address when TimeMachine is enabled, and the target processor supports data trace, the hardware breakpoint is applied to the corresponding physical address. This means that the hardware breakpoint could be hit for any virtual or physical address in any AddressSpace that translates to the physical address of the hardware breakpoint. If you set a hardware breakpoint on a virtual address when TimeMachine is enabled, but the target processor does not support data trace, the breakpoint is set on the virtual address and is triggered whenever any task executes that virtual address.

**Note**

OS-awareness in TimeMachine requires specific parts of OS execution to be reconstructed. If kernel trace data is incomplete, MULTI may discard trace data for some tasks. For more information, see “Dealing with Incomplete Trace Data” on page 411.

OSA Tasks and the Master Process in TimeMachine

This section provides information about behavior that you can expect when TimeMachine is enabled for a freeze-mode connection. It describes running through traced interrupts and exceptions from the master process and running OSA tasks. For information about freeze-mode debugging, see Chapter 26, “Freeze-Mode Debugging and OS-Awareness” on page 605.

When you are debugging the OSA master process in TimeMachine mode, you can single-step and run through traced interrupts and exceptions. TimeMachine treats all execution inside of interrupts and exceptions as a separate logical task, even

though it does not correspond to any physical task. Note that this behavior differs from that of the OSA master process when not in TimeMachine mode. When not in TimeMachine mode, you can use the OSA master process to run through all the different tasks on the target.

When you run an OSA task for which TimeMachine is enabled, other OSA tasks on the same processor, and for which TimeMachine is also enabled, run as well.

If your freeze-mode target halts, and you want to step or run backward in the current task, select the OSA task in the target list before stepping or running backward. If you do not select the task first (that is, you leave the OSA master process selected), stepping or running backward steps/runs you back in the task with the most recently traced data. To step/run back through interrupts and exceptions, select the TimeMachine target list entry associated with the OSA master process.



Tip

If you want the current task to be automatically selected whenever the target halts, ensure that the **osaSwitchToUserTaskAutomatically** configuration option is set to `on` (the default). See the **osaSwitchToUserTaskAutomatically** option in “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

OSA Breakpoints in TimeMachine

Software breakpoints set on tasks in TimeMachine mode are task specific. This differs from software breakpoints set on tasks in freeze mode, which may be task specific or AddressSpace specific (see “Working with Freeze-Mode Breakpoints” on page 618). AddressSpace-specific breakpoints are referred to as *any-task* breakpoints.

Even though you cannot *set* any-task breakpoints on tasks in TimeMachine mode, you can hit them. When TimeMachine is launched on an OSA task, TimeMachine inherits any breakpoints set in the task. Additionally, any-task breakpoints existing in TimeMachine tasks are copied to other tasks contained in the same AddressSpace when TimeMachine is launched on these new tasks.

Because any-task breakpoints are inherited from OSA tasks, you can set an any-task breakpoint in freeze mode and then enable TimeMachine if you want to hit the

breakpoint while in TimeMachine mode. Alternatively, you can simulate the behavior of an any-task breakpoint without actually using one by setting a breakpoint in each task of an AddressSpace.

In addition to encountering any-task breakpoints as a result of inheritance, you may, in some limited cases, encounter other types of breakpoints behaving like any-task breakpoints. Some targets do not provide enough information in the trace stream for MULTI to determine the active task at each point in the trace. If such a target traces data accesses, MULTI may attempt to infer task switch information by utilizing knowledge of the kernel and watching for traced writes to certain addresses. This is not infallible and sometimes results in incorrect task switch markers in the trace data. If the trace data does not include task information or if the task information may be unreliable, all breakpoints set on tasks behave like any-task breakpoints in TimeMachine mode.

Unlike breakpoints set on tasks while in TimeMachine mode, breakpoints set on the master process function as any-task breakpoints in the kernel AddressSpace within TimeMachine.

In TimeMachine mode, breakpoints that are not task specific are only hit when the breakpointed instruction is executed at the same virtual address and AddressSpace where the breakpoint was originally set. This is in contrast to the way these breakpoints behave on the live target, where regardless of how they are set, they are hit when the breakpointed instruction is executed, even if that instruction has been mapped to a different virtual address in a different AddressSpace and is executed there.



Note

In TimeMachine mode, hardware breakpoints are only displayed in the OSA master process; however, they still take effect in OSA tasks. Before you can set a hardware breakpoint in an OSA task for which TimeMachine is enabled, TimeMachine must be enabled for the master process. If TimeMachine is not enabled for the master process, MULTI prompts you to enable it.

When TimeMachine is disabled, any changes that you have made to software or hardware breakpoints while in TimeMachine mode are applied to the live target. All breakpoints in the master process revert to normal breakpoints on the target.

Using Separate Session TimeMachine

Usually, you debug an OS task or a stand-alone application either in TimeMachine mode or in live mode. In live mode, you can write registers and memory, invoke command line procedure calls, collect trace data, run the real target, etc. However, you cannot step or run backward. In TimeMachine mode, you can step and run backward, but the live target must be halted. You cannot simultaneously debug a process in TimeMachine mode and live mode.

For some applications, it is not practical or desirable to halt a live target. If you want the capability to run and step backward while your live target is running, you may debug the process by launching TimeMachine as a separate session (called *Separate Session TimeMachine*). Doing so creates an additional entry in the target list. Selecting the original target list entry allows you to run and step the process in live mode. Selecting the new target list entry allows you to run or step the process backward in Separate Session TimeMachine.

Separate Session TimeMachine functions like regular TimeMachine except for one key difference: breakpoints are not shared between Separate Session TimeMachine and live mode. Breakpoints that exist in live mode are copied over to Separate Session TimeMachine when it first starts, but from that point on, each has distinct breakpoint sets. If you set a breakpoint on a process while in Separate Session TimeMachine mode, the breakpoint does not appear in live mode, and vice versa.

To launch Separate Session TimeMachine on an application or on a freeze-mode task, do one of the following:

- Select the process in the target list and then enter the command **timemachine –newsession** in the Debugger command pane. See also the **timemachine** command in Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
- While the target is running, launch TimeMachine in any of the usual ways. In the dialog box that appears, choose to launch TimeMachine as a separate session.

To launch Separate Session TimeMachine from a run-mode task, perform the following steps:

1. Establish a freeze-mode connection, and maintain this connection for the remaining steps.

2. Download and run the kernel.
3. Establish a run-mode connection via a debug server such as **rtserv** or **rtserv2**. The run-mode connection should be established in the same Debugger window as the freeze-mode connection.
4. Collect trace data.
5. Launch TimeMachine while the kernel is running. In the dialog box that appears, choose to launch TimeMachine as a separate session.



Note

Even when you launch TimeMachine from a run-mode task, TimeMachine is associated with the corresponding freeze-mode task in the target list. This is because trace data is retrieved from the freeze-mode connection.

All TimeMachine processes on a particular CPU are synchronized. When you debug a process in Separate Session TimeMachine, any processes that are already in TimeMachine mode are switched to Separate Session TimeMachine. Once a process is in Separate Session TimeMachine, attempts to enter TimeMachine mode on any other process automatically enable Separate Session TimeMachine. It is not possible to simultaneously debug one process in TimeMachine mode and another in Separate Session TimeMachine.

To disable Separate Session TimeMachine for a process, right-click the corresponding target list entry and select **Remove** from the shortcut menu. If you want to debug a process in TimeMachine mode, remove all Separate Session TimeMachine entries from the target list and then launch TimeMachine in the normal way.

The PathAnalyzer

The PathAnalyzer shows your call stack over time so that you can analyze the execution path of your program. In addition to displaying the relationship between all functions, the PathAnalyzer keeps track of the number of times each function is called and the execution time of every function called.

The PathAnalyzer is a great tool both for debugging and for optimizing for speed. The graphical display allows you to look for anomalies in your program's execution path. You might find functions calling functions they should not or taking much longer than they should. Or you might discover the converse: functions not making calls they should be making or exiting more quickly than expected. Because the PathAnalyzer is linked with the TimeMachine Debugger, you need only double-click a function in the PathAnalyzer to examine it more closely in the TimeMachine Debugger. The PathAnalyzer's graphical display also makes the distribution of work clear by showing the amount of work involved for each function. Once you know what functions take longest, you can examine them in the TimeMachine Debugger to see what you can do to speed them up.

The PathAnalyzer is also a great tool to use for getting a rough idea of how unfamiliar code works. It usually only takes a couple minutes of exploring in the PathAnalyzer to figure out what functions you need to examine in more detail to solve a problem.

Opening the PathAnalyzer

To open the PathAnalyzer window, do one of the following:

- In the Debugger window, select **TimeMachine** → **PathAnalyzer**.
- In the Debugger command pane, enter the **tracepath** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Trace List, click the **PathAnalyzer** button () located on the toolbar.

The PathAnalyzer Window

This section describes the basic parts of the PathAnalyzer window, how function calls are colored in the PathAnalyzer, and how to display information for a function. Additional information about features you can access from this window is located in:

- “Navigating Path Analysis Data” on page 428
- “Using Fast-Find in the PathAnalyzer” on page 430
- “Browsing References of a Function” on page 431
- “Viewing, Editing, and Adding Bookmarks in the PathAnalyzer” on page 432
- “Analyzing Operating System Trace Data” on page 432

Depending on the target you are tracing, the horizontal axis of the PathAnalyzer's central pane can represent time, instruction counts, or cycle counts. If your trace data includes timestamps, the horizontal axis represents time. If your trace data includes cycle counts, but does not include timestamps, the horizontal axis uses cycle counts. If neither timestamps nor cycle counts are available, the horizontal axis uses instruction counts.

Some hardware targets support more than one type of data collection. To set which type of trace data you want to collect, perform the following steps:

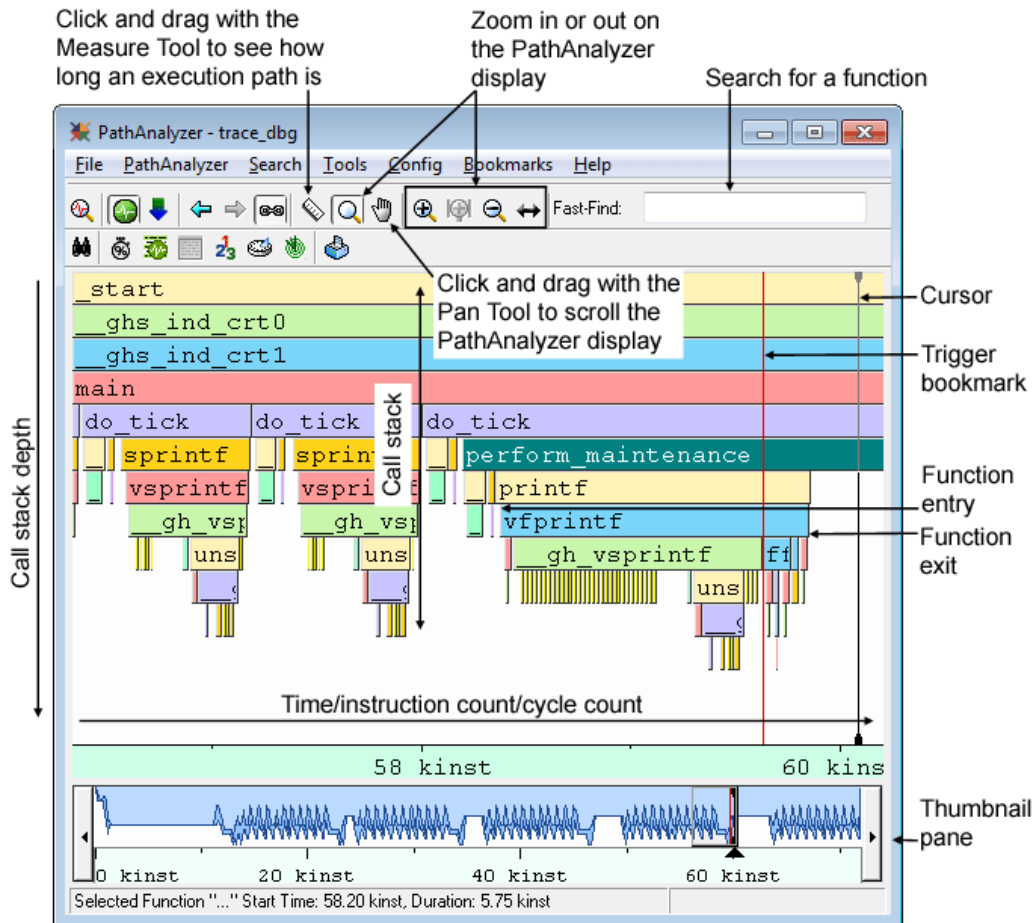
1. In the PathAnalyzer, select **Config** → **Options**.
2. In the **Trace Options** window that appears, click the **Target Specific Options** button located on the **Collection** tab, or click the target-specific tab. (Only one or the other will appear.)



Tip

Typically, if you want to analyze performance, you should collect trace data with timestamps. However, if you are tracking down a difficult bug or race condition or are trying to get a better understanding of your code flow, collecting trace data by instruction count is most helpful.

The vertical axis in the PathAnalyzer's central pane represents call stack depth. As function calls are made, the stack depth increases downward.



Each colored bar in the PathAnalyzer represents a single function call. The bar's length is proportional to the length of time (or the number of instructions) it has executed. By default, the PathAnalyzer colors function calls by hashing the name of the function. However, you can also color functions by their filename or class name. To change the method by which functions are colored, select **PathAnalyzer** → **Color by Filename** or **PathAnalyzer** → **Color by Class**.

You can display information about a function in any one of the following ways:

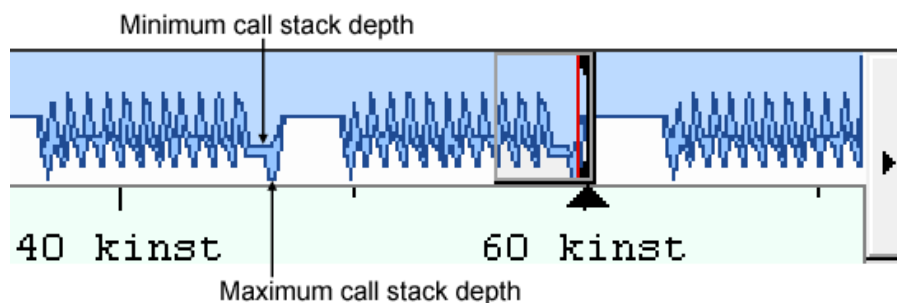
- Hover your mouse over a function to display a tool-tip with information about the function.
- Click a function to view information about it in the status bar (located at the bottom of the PathAnalyzer window).
- Click the  button or press the number 1 on your keyboard to access the **Measure Tool**. To measure the difference between two points, click a point

in the PathAnalyzer and drag your mouse right or left to create a selection. To zoom in on your selection, click the **Zoom to Selection** button ()

The Thumbnail Pane

The thumbnail pane (located at the bottom of the PathAnalyzer window) displays PathAnalyzer data in summary, giving you a bird's-eye view of your call stack over the entire span of your trace data.

The minimum call stack depth at a particular time is represented by the point where the light blue color meets the dark blue line. The maximum call stack depth at a particular time is represented by the point where the dark blue line meets the background color. In the following graphic, the background color is white.



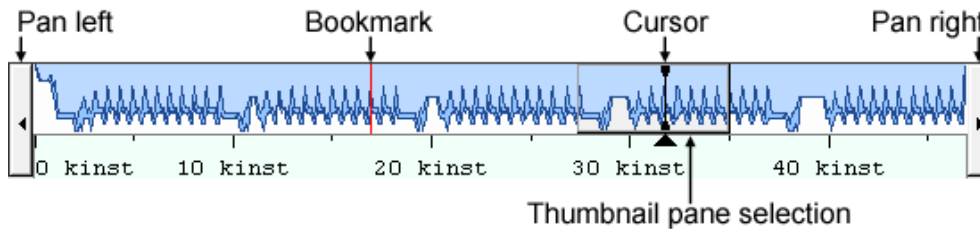
If you have not collected much trace data, the difference between the minimum and the maximum values may not be very great, resulting in what appears to be a single line.

You can zoom the PathAnalyzer in and out by clicking a point in the thumbnail pane and dragging your mouse right or left. The wider your selection, the more function calls are displayed in the central part of the window (equivalent to zooming out). The narrower your selection, the fewer the function calls displayed in central part of the window (equivalent to zooming in). After you have made your initial selection, you can zoom in or out by dragging the right and left edges of your selection.

The thumbnail pane also acts as the horizontal scroll bar for the PathAnalyzer. To pan right or left from the thumbnail pane, do one of the following:

- Click the center of the thumbnail pane selection and drag it right or left.

- Click a new point in the thumbnail pane. If you click outside your current thumbnail pane selection, the selection moves to your new location.
- Click the **Pan Right** or **Pan Left** button. These buttons are located at either end of the thumbnail pane, as shown in the following graphic.



Bookmarks are displayed as short vertical lines in the thumbnail pane. Each bookmark is colored according to the color set in the **Trace Bookmarks** window. For more information about bookmarks, see “Viewing, Editing, and Adding Bookmarks in the PathAnalyzer” on page 432.

Navigating Path Analysis Data

Embedded processors generate massive amounts of trace data. As a result, it is completely typical to analyze a path analysis run with hundreds of millions of instructions. When that much complexity is involved, simply looking, function by function, at your function flow is not feasible. To help you efficiently interact with the generated data, the PathAnalyzer provides a variety of navigational tools, which are described next. (See also “Searching Path Analysis Data” on page 430.)

- *Cursor* — When data is available in the PathAnalyzer, a cursor, which indicates the position of the currently selected trace packet, is also available. The cursor is the black vertical line displayed in the graphic in “The PathAnalyzer Window” on page 425. By default, the cursor's location is shared among other TimeMachine tools. As a result, clicking a different location in the PathAnalyzer changes not only the position of the cursor in the PathAnalyzer, but also the selection in the Trace List and the state of the target in the TimeMachine Debugger. Likewise, if you run or step in the TimeMachine Debugger, the PathAnalyzer's cursor also moves. If the Debugger is not in TimeMachine mode, clicking a different location in the PathAnalyzer will warp the Debugger to show that location in the source pane and will highlight the associated source line or instruction.

If you do not want the cursor to be synchronized with the Debugger and the Trace List, you can “unlink” the tools by clicking the pushed down **Unlink Selection** button (.

- *Bookmarks* — When you find some interesting data in the PathAnalyzer, you can bookmark it so that you can easily find it again later. You can assign each bookmark a name and color. Bookmarks appear in the PathAnalyzer as colored vertical lines. In addition to any bookmarks you might create yourself, MULTI automatically selects and bookmarks the trigger packet when trace data is first collected. The default name for the trigger bookmark is “Trigger,” and it is colored red by default (see the graphic in “The PathAnalyzer Window” on page 425). For more information about using bookmarks with the PathAnalyzer, see “Viewing, Editing, and Adding Bookmarks in the PathAnalyzer” on page 432.
- *Smooth Scrolling* — When the location of the PathAnalyzer cursor is synchronized with other TimeMachine tools (see preceding bullet point), and the cursor moves because of a location change in another window, the adjustment can be disorienting. To make the transition less sudden, the PathAnalyzer has a smooth-scrolling feature. When the location of the cursor is updated to reflect a change in another window, the Path Analyzer gradually scrolls you to the new destination.

You can toggle smooth scrolling on and off by selecting **PathAnalyzer → Enable Smooth Scrolling**.

- *One-Click Zoom Tool* — The easiest way to zoom in is by using the zoom tool. To access this tool, click the **Zoom Tool** button () or press the number 2 on your keyboard. To zoom in, click a point in the PathAnalyzer and drag your mouse right or left to create a selection. For more information about zooming methods, see the following table.
- *Pan Tool* — To pan right and left, you can use the pan tool. To access this tool, click the **Pan Tool** button () or press the number 3 on your keyboard. Click a spot in the PathAnalyzer and drag right or left. For more information about panning methods, see the following table.

The following table lists different ways to zoom and pan.

Desired Action	Method
Zoom in	• Click the Zoom in button (. • Press Ctrl+UpArrow. • Press Ctrl while using the mouse scroll wheel to scroll up.
Zoom in on a selection	• Click the Zoom Tool button (). Then click a point in the PathAnalyzer and drag your mouse right or left to create a selection. • Press the number 2 on your keyboard. Then click a point in the PathAnalyzer and drag your mouse right or left to create a selection.
Zoom out	• Click the Zoom out button (. • Press Ctrl+DownArrow. • Press Ctrl while using the mouse scroll wheel to scroll down.
Zoom out completely	• Click the Show entire timeline button (. • Press Ctrl+Home.
Pan right	• Click the Pan Right button located in the bottom-right corner of the PathAnalyzer window. (See the second graphic in “The Thumbnail Pane” on page 427.) • Press Ctrl+RightArrow.
Pan left	• Click the Pan Left button located in the bottom-left corner of the PathAnalyzer window. (See the second graphic in “The Thumbnail Pane” on page 427.) • Press Ctrl+LeftArrow.

Searching Path Analysis Data

To help you find important information, the PathAnalyzer provides two different search mechanisms. These mechanisms are described in the following sections. (See also “Navigating Path Analysis Data” on page 428.)

Using Fast-Find in the PathAnalyzer

The **Fast-Find** feature provides you with an easy way to search through trace data. Simply type search term(s) into the **Fast-Find** text field located in the top-right

corner of the PathAnalyzer window. Everything in the PathAnalyzer that does not match the search string turns gray.

You may enter as many search strings as you want. All search strings are processed through a logical AND. That is, a function turns gray unless it contains all of the search strings.

To eliminate functions from your search, use the - (minus) operator. A function is eliminated when its name matches the string following the minus sign.

Searches are case-insensitive unless you use capital letters in the search string.

Example search strings follow:

- **Fast-Find:** `foo -bar`
 - This string is case-insensitive.
 - Entering this string displays all functions whose names contain the string `foo` but not the string `bar`.
- **Fast-Find:** `sendMsg_ -sync`
 - This string is case-sensitive.
 - Entering this string displays all functions whose names contain the string `sendMsg_` but not the string `sync`.

Browsing References of a Function

It is often useful to see a list of every call to a function with call sites and durations. This can be accomplished with the **Trace Call Browser**. To open a **Trace Call Browser** directly from the PathAnalyzer, right-click a single instance of a function, and select **Find Function References**. If you cannot easily find the function you are looking for in the PathAnalyzer, there are many other ways to open it in a **Trace Call Browser**. For more information, see “The Trace Call Browser” on page 457.

Viewing, Editing, and Adding Bookmarks in the PathAnalyzer

The TimeMachine bookmark system is tightly integrated with the PathAnalyzer, through which you can view, edit, and add bookmarks.

Bookmarks appear in the PathAnalyzer as colored vertical lines. The color of the line corresponds to the bookmark's color as set in the **Trace Bookmarks** window. For more information, see “Bookmarking Trace Data” on page 444.

To view a list of available bookmarks, select the PathAnalyzer's **Bookmarks** menu. To jump to a particular bookmark, select the bookmark in the **Bookmarks** menu.

The easiest way to edit bookmarks is by using the **Trace Bookmarks** window. Select **Bookmarks** → **Edit Bookmarks**. For more information, see “Bookmarking Trace Data” on page 444.

You can easily add a bookmark by right-clicking a function in the PathAnalyzer. You have the option of adding the bookmark to the beginning of the selected function (**Add Bookmark at Entry**), to the end of the selected function (**Add Bookmark at Exit**), or to the point of selection (**Add Bookmark at Cursor**).

Analyzing Operating System Trace Data

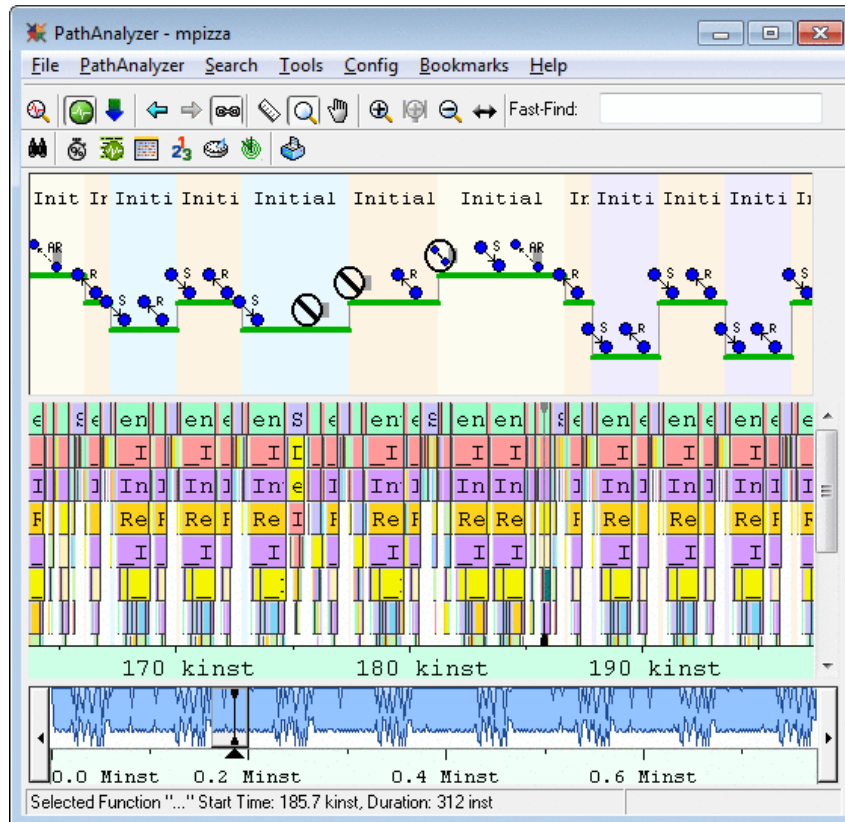
The PathAnalyzer has two features that help you analyze operating system trace data:

- An EventAnalyzer pane within the PathAnalyzer
- The ability to view a single AddressSpace in the PathAnalyzer

These features automatically become available when you trace an operating system that is supported by the trace tools.

Viewing Data in the EventAnalyzer Pane

The EventAnalyzer pane appears directly above the PathAnalyzer. It displays operating system events and context switches.



The name of each task is displayed in the topmost portion of the EventAnalyzer pane. The current task is displayed as a green horizontal line. To highlight a task, hover your mouse over it. When a task is highlighted, its name is prominently displayed in the top portion of the EventAnalyzer pane and its context switches turn a brighter shade of green.

The EventAnalyzer pane is not as fully featured as the MULTI EventAnalyzer. To access the full MULTI EventAnalyzer, click the  button. For information about the MULTI EventAnalyzer, see the *EventAnalyzer User's Guide* or the documentation about using the MULTI EventAnalyzer for ThreadX in the *MULTI: Developing for ThreadX* book.



Note

Certain architectures support AddressSpace tracing but not task tracing. If this is the case, you are able to see context changes between different AddressSpaces but not context switches within a single AddressSpace. For more information, see “Dealing with Incomplete Trace Data” on page 411.

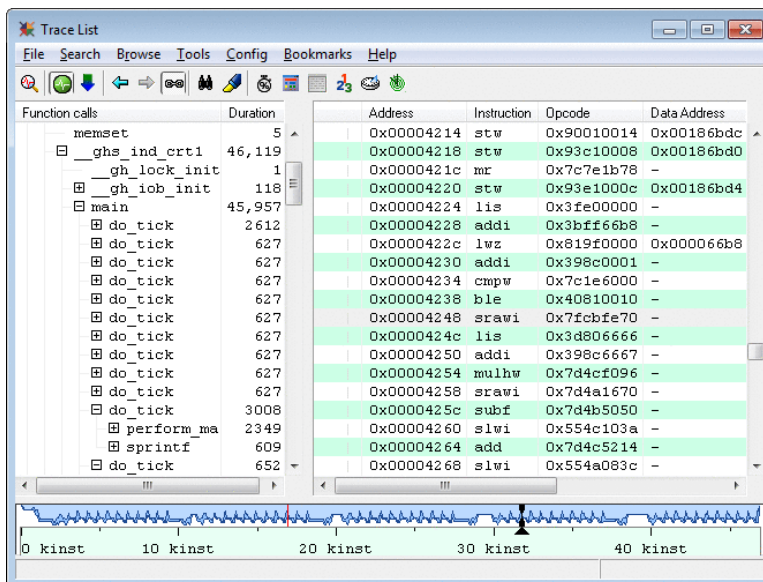
Viewing a Single AddressSpace in the Path Analyzer

When you trace an entire operating system, a large amount of information is displayed in the PathAnalyzer. If you are only interested in seeing one AddressSpace, you can launch a new PathAnalyzer that only displays function calls in that particular AddressSpace. To view a single AddressSpace in a new PathAnalyzer, select the AddressSpace you want to view from the **PathAnalyzer** menu.

Viewing Trace Data in the Trace List

The Trace List allows you to view and explore trace data at the function and assembly levels and to control trace collection. It also provides access to advanced trigger, search, filter, and bookmark controls, in addition to various trace data analysis tools. To open the Trace List, do one of the following:

- In the Debugger, select **TimeMachine** → **Trace List**.
- In the Debugger command pane, enter the **trace list** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



The Trace List

The Trace List consists of three panes that make it easy to browse through trace data:

- The *tree pane*, which appears on the left side of the window, displays the call stack in a tree view. Each node in the tree represents a function call. Expanding a node displays additional, chronologically ordered nodes for all the functions called during the associated function call. When you are tracing an application with multiple tasks or AddressSpaces, top-level function nodes also display the name of the associated task or AddressSpace.
- The *instruction pane*, which appears on the right side of the window, displays a list of executed instructions. This pane shows all the instructions in a single list, regardless of what function or AddressSpace the instructions reside in.
- The *thumbnail pane*, which appears in the bottom of the window, displays the call stack of your program over time. This pane provides an overview that shows when your program was calling many functions and when it was executing a relatively shallow call stack. The thumbnail pane also displays the amount of time you spent gathering trace data, any bookmarks you may have set, and your current location in the trace buffer.

The Tree Pane

The tree pane groups trace data into function calls, enabling you to see where you are browsing relative to other function calls. This eases the process of browsing complicated code by allowing you to jump to a point in time by selecting a function.

To enable you to see what functions called what other functions, the tree pane's **Function Calls** column displays executed functions in tree form. At each level, you can expand a function to display the functions it called. To navigate to the trace data corresponding to a function, select the function in the tree list.

The tree pane's **Duration** column displays the amount of time that the given function took to execute. Depending upon your trace collection settings, the duration is displayed in instruction counts, cycles, or elapsed time. The duration includes the time taken by the given function as well as the time taken by functions it called.

The splitter between the tree pane and the instruction pane allows you to resize the tree pane for the appropriate amount of detail.

When you are tracing a target with multiple tasks, context switches between tasks appear as multiple top-level entries in the tree pane. If the target operating system supports multiple AddressSpaces, top-level entries include the AddressSpace name. If task-aware trace is supported, they also include the task name. Colons are used as delimiters between the AddressSpace name, task name, and function name. When all three are available, top-level nodes have names of the form *AddressSpace:Task:Function*. Alternating colors ease readability and highlight context switches.

The Instruction Pane

The instruction pane provides up to eleven columns of information for each instruction. Some columns only apply to certain instructions.

If you want to focus only on certain columns of information in the instruction pane, you can hide columns by right-clicking the column header. Then select the **Hide column** menu option that corresponds to the column you want to hide. The menu items corresponding to hidden columns appear ticked. To show the column again, select the menu item and the column appears.

To rearrange the columns, click and drag the column header of the column you want to move. Move it horizontally to the desired location and release the mouse button. To resize a column, click and drag the right border of the column header. To automatically resize a column so that all data in the column is shown, double-click the right border of the column header. Column order and size are automatically saved between sessions.

Each column and the information it contains is described in the following table.

Unlabeled left-most column	Shows whether or not the instruction is bookmarked. When a bookmark is present, a small flag is drawn in the color of the bookmark. When no bookmark is present, a light gray line is displayed. You can add or remove a bookmark by clicking the vertical gray line or the flag, respectively.
Address	The address of the instruction. If the instruction was in a known function, the function name and offset into the function is also displayed.
Instruction	The disassembled instruction.
Opcode	The opcode of the instruction.

Data Address	The address that the instruction loaded from or stored to. This column only applies if the instruction performed a traced load or store.
Access	The type of memory access performed by the instruction. If the instruction read data from memory, this column contains an R . If the instruction wrote data to memory, this column contains a W . This column only applies if the instruction performed a traced load or store.
Value	The value loaded from or stored to memory. This column only applies if the instruction performed a traced load or store.
Executed	Yes if the instruction met its condition code requirements and was executed, No if it did not. This column only applies if your target has conditional execution.
Cycles	The number of cycles the instruction took to execute. This column only applies if your target supports cycle count tracing and cycle counts are enabled.
Time	The time taken to execute the instruction. This column only applies if your target supports trace timestamps and timestamps are enabled.
Total Instructions, Total Cycles, or Total Time	The name of this column depends upon whether you have timestamps, cycle counts, or neither enabled. If you have timestamps, the cumulative time to that line is displayed. If you have no timestamps but you do have cycle counts, this column displays the cumulative cycle count up to this line. If you have neither timestamps nor cycle counts, this column displays the total number of instructions to this point in the trace data. All values displayed in this column are relative to one line—initially the trigger—with negative numbers indicating that an instruction was executed before the zero line.

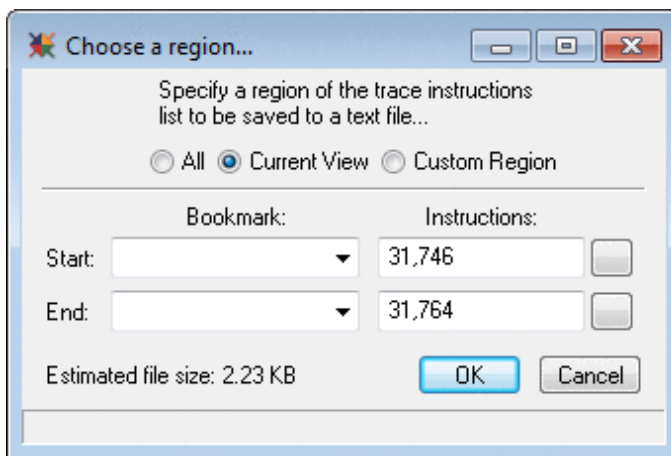
If an instruction performed multiple data accesses, only the first data access is listed on the same line as the instruction. The additional data accesses are shown on separate lines that are initially hidden. To show the additional data accesses, click the plus sign (+) to the left of the instruction.

In addition to lines representing instructions, functions, and AddressSpaces, the Trace List also includes lines that represent trace events. Trace events include exceptions, trace overflows, markers that indicate that the trace was disabled, markers that indicate that the target entered debug mode, and trace processing errors. Each of these events is displayed on a separate line in red text in the **Address** column.

With the default color scheme, normal instructions and data accesses are displayed with black text. On some targets, you may see blocks of instructions and data accesses highlighted in blue. This indicates that those instructions and data accesses were not present in the raw trace data because of an on-chip trace FIFO overflow, but that MULTI was able to determine with a high degree of certainty that those instructions and data accesses actually occurred. MULTI accomplishes this by examining the trace data both before and after the overflow and simulating the code in between. A line with red text reading **FIFO Overflow** indicates that MULTI was unable to reconstruct all of the instructions lost as a result of a FIFO overflow. For more information, see the option **Attempt to reconstruct gaps in trace data** in “The Trace Options Window” on page 480. It is also possible to create filters that color lines in the Trace List in any color. For more information, see “Filtering Trace Data in the Trace List” on page 442.

You can print the contents of the instruction pane by selecting **File** → **Print** from the Trace List menu bar. To print the contents of the instruction pane to a text file, select **File** → **Print To Text File**. In both cases, the instruction pane is formatted as it is currently displayed. If a column is not wide enough to display the full contents of a cell, that cell's contents are truncated in the printed output as well. If you would like to output the trace data in a program readable format, select **File** → **Export As CSV**. This outputs the trace data in Comma Separated Value format to a text file. You can control which columns appear in the CSV file by showing and hiding them in the instruction pane. Columns that are displayed in the instruction pane are exported to the CSV file, columns that are hidden are not.

All three of the preceding menu options open the **Choose a region** dialog box, which asks you to select a region of trace data to act upon.



By default, only the visible portion of the instruction pane is printed or exported, but you can select the **All** radio button to print or export the entire contents of the instruction pane. Alternatively, select the **Custom Region** radio button to print or export a specific range of data. To identify a custom region, select bookmarks that delimit the region or specify start and end times by manually typing them into the **Start** and **End** fields or by clicking the box to the right of either field and then clicking an instruction in the Trace List. If you selected **File** → **Print**, the dialog box indicates the number of lines of text that will be printed when you click **OK**. If you selected **File** → **Print To Text File**, the dialog box shows the estimated size of the resulting file.

The Thumbnail Pane

The thumbnail pane is a graphical representation of your call stack over the entire time span of your trace buffer. The thumbnail pane in the Trace List is equivalent to that in the PathAnalyzer except that the Trace List's thumbnail pane lacks pan and zoom controls. See “The Thumbnail Pane” on page 427.

The thumbnail pane allows you to quickly navigate to any data point by clicking a point in the pane. The thumbnail pane indicates the position of the selected instruction with a vertical cursor.

Any bookmarks that you set are displayed as short vertical lines in the thumbnail pane. Each bookmark is colored according to the color set in the **Trace Bookmarks** window. To display a bookmark in the instruction pane, click the bookmark's vertical line in the thumbnail pane. For more information about bookmarks, see “Bookmarking Trace Data” on page 444.

Time Analysis

One of the major benefits of collecting trace data is that it allows you to non-intrusively determine what the processor is doing while the processor is running at full speed. Analyzing trace data allows you to determine the time between two events. In this context, an event can be anything that appears in the Trace List, including instructions, function calls, memory accesses, and even interrupts.

The **Total** column in the Trace List always displays the cumulative time before the instruction on a line is executed. This column displays the following information in each of these cases:

- When you have time tags enabled on your trace collection device, this column is called **Total Time** and displays the total time before each instruction.
- When you have no time tags enabled, but you have cycle-accurate mode enabled, this column is called **Total Cycles** and displays the total number of cycles elapsed before each instruction is executed.
- When you have neither time tags nor cycle-accurate mode enabled, this column is called **Total Instructions** and displays the total number of instructions executed before each instruction is executed.

This information allows you to find two points in your trace data and determine the total amount of time, number of cycles, or number of instructions elapsed, between two points.

In addition, you can set the **Total** column to 0 at any line so that the Trace List can do the required subtraction. This makes it easy, for example, to find what was happening exactly 10 milliseconds after an interrupt. It also allows you to easily find the time between 2 events of interest. For example, if you want to know exactly how long the interrupt handler for a high-priority interrupt takes, you can set the **Total** column to 0 when the interrupt occurs, then find the return from the interrupt handler and see what the elapsed time is.

To set the **Total** column to 0 on a line, simply right-click the line and select **Set Total Time to 0 on this Line**. When this option is selected, the Trace List will refresh and the selected line will have its **Total** column set to 0. All other instructions will now have their **Total** value relative to the selected line. Negative numbers indicate that an instruction executed before the selected line and positive numbers indicate that an instruction executed after the selected line.

Navigating through Trace Data

The Trace List has many features to make it easy to navigate through large trace buffers. These features allow you to easily find occurrences of instructions, functions and variables in your trace data as well as recall previous lines you have viewed.

Navigating Based on Trace Data

If you are looking for an event of interest, you may not find the specific execution of an address or data access that you are looking for on the first try. The Trace List allows you to easily navigate to the previous or next execution of an instruction or access to a data address.

- To navigate to the previous execution of an instruction, right-click the instruction of interest. In the shortcut menu that appears, select **Go to Previous Execution**. The Trace List selects the previous execution of the selected instruction.
- To navigate to the next execution of an instruction, right-click the instruction of interest. In the shortcut menu that appears, select **Go to Next Execution**. The Trace List selects the next execution of the selected instruction.
- To navigate to the previous data access at an address, right-click a line that shows an access of the data address of interest. In the shortcut menu that appears, select **Go to Previous Data Access**. The Trace List selects the instruction that last accessed the selected data address.
- To navigate to the next data access at an address, right-click a line that shows an access of the data address of interest. In the shortcut menu that appears, select **Go to Next Data Access**. The Trace List selects the next instruction that accessed the selected data address.

If no previous or next instance of an event exists, the Trace List prints an error message in its status bar.

By default, the selected instruction is shared among other TimeMachine tools. As a result, clicking a different line in the Trace List also moves the cursor in the PathAnalyzer and changes the state of the target in the TimeMachine Debugger. Likewise, if you click a different location in the PathAnalyzer or if you run or step in the TimeMachine Debugger, the selection in the Trace List also changes. If the Debugger is not in TimeMachine mode, clicking a different line in the Trace List will warp the Debugger to show that location in the source pane and will highlight the associated source line or instruction.

If you do not want the selection to be synchronized with the Debugger and the PathAnalyzer, you can “unlink” the tools by clicking the pushed down **Unlink Selection** button ().

Browse History

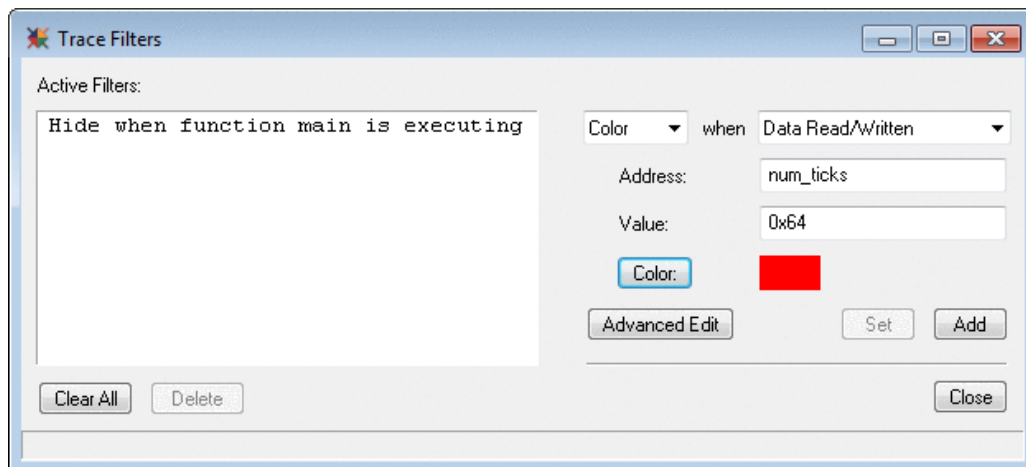
The  and  buttons allow you to move to the previous and next instructions that have been selected in the Trace List. These buttons work just like similar buttons on standard Web browsers. This allows you to easily move between various instructions that you have examined in your trace data.

Filtering Trace Data in the Trace List

The Trace List provides the ability to filter trace data, which makes it easier for you to find the data that you are looking for. You can hide, show or color groups of instructions based on various criteria. Filtering only affects the instruction pane so that you do not lose your context in the tree pane.

To hide all executions of a function in the instruction pane, simply right-click an instruction from the function you want to hide and select **Hide Function** *function_name*. The data redisplay with the selected function hidden.

In addition, you can define more complex filters using the **Trace Filters** dialog. To access this dialog, select **Search** → **Filters** from the menu bar or click  on the toolbar.



The **Trace Filters** dialog consists of a list of currently active filters as well as controls to add new filters. To add a filter, select the action you would like the filter to have and the type of event to filter. Then enter a function name, source line, address, or variable name and an optional value for data accesses. When you select

the **Color** action, you can also specify the color for the matching lines. Then click the **Add** button. You can add as many filters as you would like in this manner.



Note

On INTEGRITY, you may only use kernel space symbols when creating filters. Also note that Trace List filters currently match virtual addresses in all AddressSpaces. For example, if you create a filter on a **Data Read** at **Address** 0x1000, MULTI triggers the filter when 0x1000 is read in any AddressSpace.

Filters are applied in the order that they are added and once a line matches a single filter, processing for that line stops. This means that if multiple filters match on a given line, the action specified by the first one in the **Active Filters** list is performed. For example, if two filters match on a line and the first is to **Color** and the second is to **Hide** that line, the line is colored rather than hidden.

You can edit an existing filter by selecting it in the **Active Filters** list and changing the values associated with it. Then click the **Set** button to confirm the new filter settings. The **Add** button remains active, allowing you to add a new filter based on the previous filter.

In addition, you can edit a filter to specify a complex event by selecting an existing filter and clicking the **Advanced Edit** button. This opens the **Advanced Event Editor** window, which allows you to define a complex event. When you click **OK** in the **Advanced Event Editor** window, the changes are automatically applied to the selected filter. For more information about using the **Advanced Event Editor** window, see “Editing Complex Events” on page 497.

You can also manage your list of filters either by clearing all filters or by deleting an individual filter.

- To clear all existing filters, click the **Clear All** button.
- To delete an individual filter, select it in the **Active Filters** list and click the **Delete** button.

Accessing TimeMachine Analysis Tools

The Trace List's menu items and toolbar buttons allow you to access advanced TimeMachine functions and tools. The following list outlines how to access TimeMachine features from the Trace List:

- **TimeMachine** — Double-click a line in the Trace List. A TimeMachine Debugger opens at the selected instruction. Alternatively, select **Tools** → **TimeMachine Debugger** or press the  button. For more information about TimeMachine, see “The TimeMachine Debugger” on page 413.
- **Profile** window — Select **Tools** → **Profile** or click the  button. For more information, see “Using Trace Data to Profile Your Target” on page 475 and Chapter 17, “Collecting and Viewing Profiling Data” on page 353.
- **PathAnalyzer** — Select **Tools** → **PathAnalyzer** or click the  button. For more information, see “The PathAnalyzer” on page 424.
- **EventAnalyzer** — Select **Tools** → **EventAnalyzer** or click the  button. For more information, see “Viewing Trace Events in the EventAnalyzer” on page 474.

Bookmarking Trace Data

When you find an interesting point in a trace run, you may want to bookmark it so that you can go back to it later. To bookmark a point in your trace data, do one of the following.

- In the Trace List, right-click the instruction you would like to bookmark and then select the **Add Bookmark** menu item.
- In the PathAnalyzer, right-click the function you would like to bookmark and then select **Add Bookmark at Entry**, **Add Bookmark at Exit**, or **Add Bookmark at Cursor**.
- In the Trace List or in any of the **Trace Browsers**, click the gray line that appears to the left of the instruction.



Note

The right-click menu items are also available via the **Bookmarks** menu.

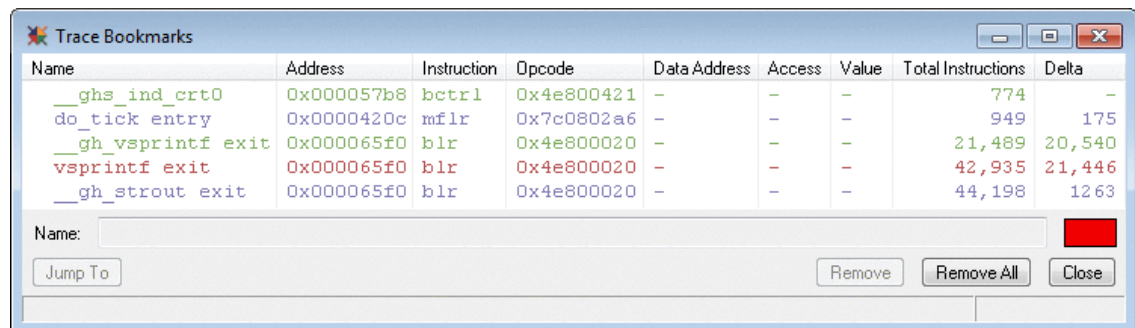
In windows such as the Trace Browsers and Trace List, a flag is drawn next to bookmarked instructions. In the Trace List, the text of bookmarked instructions is also colored. In windows such as the Trace List and PathAnalyzer, vertical lines representing bookmarks are added to the thumbnail pane at the bottom of the window.

In addition to the manually added bookmarks, when a trigger is found in the trace data, a bookmark is automatically added to allow you to easily navigate to the trigger position. The default name for trigger bookmarks is “Trigger,” and they are colored red by default.

Once there are one or more bookmarks, you can view all of the defined bookmarks through the **Trace Bookmarks** window. To open the **Trace Bookmarks** window, do one of the following:

- In the Trace List or the PathAnalyzer, select **Bookmarks** → **Edit Bookmarks**.
- In the Debugger command pane, enter the **trace bookmarks** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

The following graphic displays the **Trace Bookmarks** window.



Each bookmark has a name and a color associated with it. By default, bookmarks are named for the function they correspond to.

The **Name** column in the **Trace Bookmarks** window displays the bookmark's name. The **Delta** column displays the difference (in instructions, time, or cycles) between a bookmark and the previous bookmark. For a description of the remaining data columns, see “The Instruction Pane” on page 436.

To change the name of a bookmark, select it in the **Trace Bookmarks** window. Then change the value in the **Name** field and press **Enter**.

To change the color of a bookmark, select it in the **Trace Bookmarks** window. Then double-click the color box in the lower-right corner of the window. Select a new color in the color chooser dialog that appears. When you click **OK**, the bookmark's color is updated.

To jump to a bookmarked instruction, do any one of the following.

- In the **Trace Bookmarks** window, double-click a bookmark.
- In the **Trace Bookmarks** window, select a bookmark and click **Jump To**.
- In the Trace List or the PathAnalyzer, choose a bookmark from the **Bookmarks** menu.
- In the Trace List or the PathAnalyzer, click a bookmark line in the thumbnail pane.

To remove selected bookmarks, do any one of the following.

- In the **Trace Bookmarks** window, select a bookmark and click **Remove**.
- In the Trace List or in one of the Trace Browsers, click the flag next to an instruction.

To remove all bookmarks, do one of the following:

- In the **Trace Bookmarks** window, click **Remove All**.
- In the Trace List or the PathAnalyzer, select **Bookmarks → Delete All Bookmarks**.

Searching Trace Data

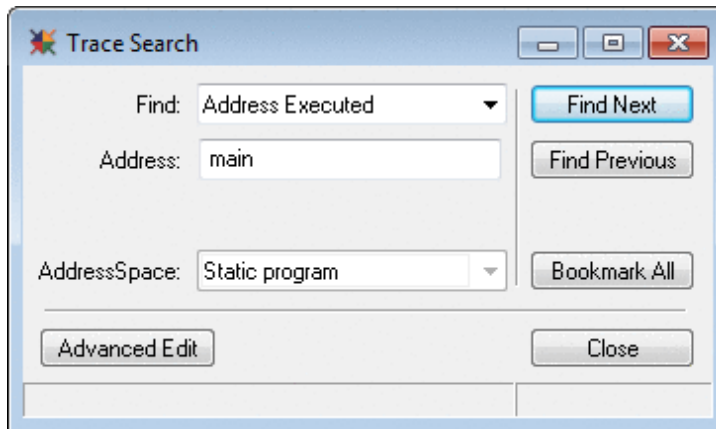
The Trace List has powerful search capabilities to enable you to quickly find specific information in a large trace run.

The simplest searching capability allows you to search through the data that currently appears in either the tree or instruction pane of the Trace List. To perform a search of this type, click in the pane you want to search, press **Ctrl+F**, and then enter a string. Matching strings are highlighted as you type. To search again for the same

string, press **Ctrl+F** again. To search backward for a string, press **Ctrl+B**. To abort the current search, press **Esc**.

In addition to this simple searching mechanism, you can use the **Trace Search** dialog box to search for various events in your trace data. To open this dialog box, do one of the following:

- In the Trace List or the PathAnalyzer, select **Search → Search**.
- In the Trace List or the PathAnalyzer, click the  button.



The **Trace Search** dialog, shown above, allows you to search your trace data for the events listed in the following table.

Event	Description
Address Executed	Searches for execution of a specific instruction. The address of the instruction may be specified by entering a function name and offset, function name and line number, or the address.
Function Executing	Searches for execution of any instruction in a specific function.
Function Not Executing	Searches for execution of any instruction not in a specific function.
Data Read/Written	Searches for reads from or writes to a variable or address. You can also search only for accesses that read or wrote a specific value.
Data Read	Searches for reads from a variable or address. You can also search only for reads that read a specific value.
Data Written	Searches for writes to a variable or address. You can also search only for writes that wrote a specific value.

Event	Description
Event	Searches for a specific type of trace event. The available events are described in “Event Type Descriptions” on page 501.
Exception	Searches for a specific type of exception. The available exceptions are described in “Exception Type Descriptions” on page 502. The Exception option is not available on all targets.
Custom Event	Searches for a complex event. You can specify the complex event with the Advanced Edit button. See “Editing Complex Events” on page 497 for details.

To search for an event, select the type of event to search for from the **Find** drop-down list. Then enter the appropriate information for the event and click the **Find Next** button to find the next occurrence, or click the **Find Previous** button to find the previous occurrence.

If you have collected trace data from multiple AddressSpaces, you can limit your event search to only one of the AddressSpaces by selecting one from the **AddressSpace** drop-down list. Any symbols specified in the search are looked up in the selected AddressSpace or, if **All** is selected, in the kernel AddressSpace.

You can also bookmark all matches to a search with the **Bookmark All** button. Clicking this button adds bookmarks for all matches, indicating their locations in the thumbnail pane and allowing you to view them in the **Trace Bookmarks** window. When there are more than 100 search matches, you are offered the choice of adding bookmarks for all search matches, for only the first 100 search matches, or for none of the search matches.

In addition to the predefined events described above, you can find arbitrarily complex events by using the **Advanced Edit** button. The interface to specify these complex events is described in “Editing Complex Events” on page 497.

Saving and Loading a Trace Session

MULTI provides a simple mechanism to save your trace data and everything needed to analyze it in a single file. Imagine that you have captured trace of a bug that is extremely difficult to reproduce and is only triggered by a specific environment that takes a long time to replicate. You have tracked the problem to part of the system that is maintained by a colleague. Now regardless of whether that colleague is located down the hall or across the world, you can save the trace session, including

all applicable ELF files and debug information, and send it to her. She can load the trace session and use the TimeMachine Debugger to step through her code and find the problem, or use the PathAnalyzer or any of the other trace tools to see what went wrong. Perform one of the following actions to save a trace session (note that source files are not saved):

- In the Trace List or PathAnalyzer, select **File** → **Save Session**.
- In the Debugger command pane, enter the **tracesave** command. You may optionally specify a file to save the trace data to. For more information, see the **tracesave** command in Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

If you are concerned for intellectual property reasons about saving your ELF image and/or debug information, you can optionally not include them in the trace session. To save only the trace data, enter the command **tracesave --data**. To save the trace data and ELF image but not the debug information, enter the command **tracesave --data_elf**. (For more information about the **tracesave** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.) Note that certain trace tools, such as the PathAnalyzer, require debug information in order to display useful data.



Note

Large temporary files may be created during the process of saving a trace session. By default, these files are created in the directory where the trace session file is created. As a result, more space will be needed on that file system than is needed to store just the trace session file. You can specify an alternative location for the temporary files with the **tracesave** command (see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book).

The information you save in the trace session file determines how you load the file. For details, see the following table.

If you saved:	To load the trace file, you must:
A trace session that includes the ELF file(s) (This is the default.)	• Open the Debugger, and then load the trace session as described below*, or • Start the Debugger from the command line, and specify the trace session file on the command line as described in “Starting the Debugger in GUI Mode” on page 6.

If you saved:	To load the trace file, you must:
Only the trace data	1. If you are using INTEGRITY, locate and debug the kernel executable. If you are not using INTEGRITY, locate and debug the ELF file that was used when you saved the trace data. 2. Load the trace session as described below.* Note: If you have rebuilt the program since you collected the trace data, loading and/or using the saved trace data may produce unexpected behavior.
Trace data saved with MULTI 4.x	1. If you are using INTEGRITY, locate and debug the kernel executable, and connect to a simulator that can run the executable. If you are not using INTEGRITY, locate and debug the ELF file that was used when you saved the trace data, and connect to a simulator that can run the ELF file. 2. Load the trace session as described below.* Note: If you have rebuilt the program since you collected the trace data, loading and/or using the saved trace data may produce unexpected behavior.

*: To load the trace session:

- In the Trace List or PathAnalyzer, select **File** → **Load Session**.
- In the Debugger command pane, enter the **traceload** command. You may optionally specify a file to load. For more information about the **traceload** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

Because MULTI does not save source files when you save a trace session, you will be unable to see source if MULTI cannot locate the original source files when you load the trace session. If MULTI can locate the source files, but they have been modified since originally being traced, they will not accurately match the saved trace data. To load source files that you have saved, use the **source** or **sourceroot** command. For information about these commands, see “General Configuration

Commands” in Chapter 6, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.



Tip

If a large number of tasks will be loaded from trace, it may be useful to set the configuration option **osaTaskAutoAttachLimit**, which allows you to specify the maximum number of OSA tasks that are displayed in the target list (the default is 32). If the number of tasks exceeds this limit, one task per AddressSpace is displayed in the target list. To manually display additional tasks loaded from trace, open the Trace List or PathAnalyzer and select an instruction or function call corresponding to the task you want to view. For information about opening the Trace List, see “Viewing Trace Data in the Trace List” on page 434. For information about opening the PathAnalyzer, see “Opening the PathAnalyzer” on page 424. For more information about the configuration option **osaTaskAutoAttachLimit**, see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

To exit a loaded trace session, do one of the following:

- Disconnect from the simulated target.
- Right-click the simulated program or each individual OS task and select **Remove**.

Browsing Trace Data

The Trace Browsers allow you to quickly locate events in your trace data by finding similar events, such as instructions and memory accesses. There are four types of Trace Browsers:

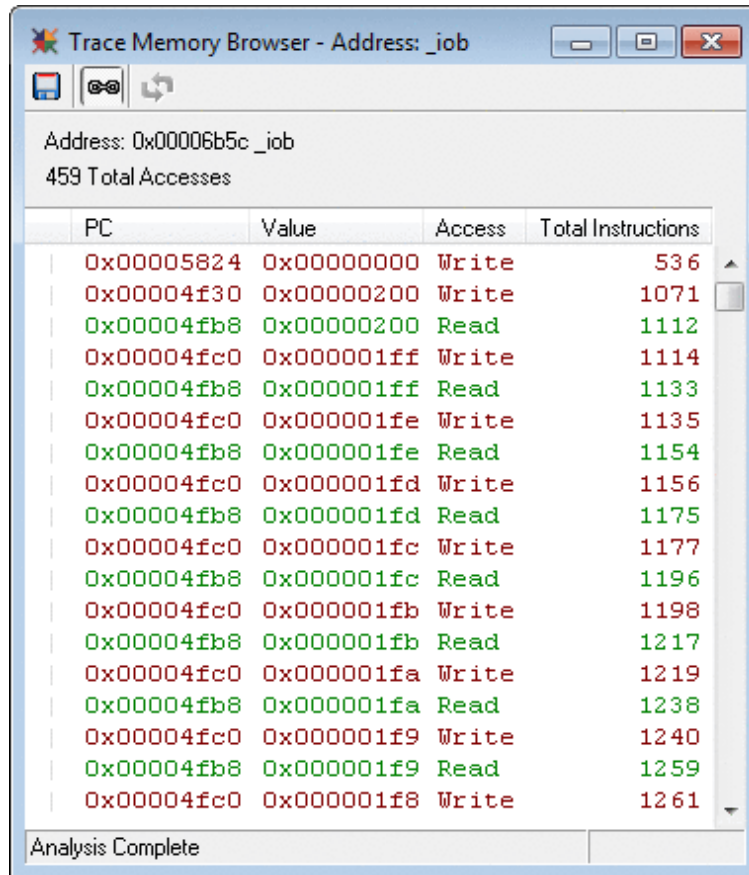
- **Trace Memory Browser** (see “The Trace Memory Browser” on page 452)
- **Trace Branch Browser** (see “The Trace Branch Browser” on page 454)
- **Trace Instruction Browser** (see “The Trace Instruction Browser” on page 456)
- **Trace Call Browser** (see “The Trace Call Browser” on page 457)

The Trace Memory Browser

The **Trace Memory Browser** allows you to quickly view each time that a specific memory address is referenced in your trace data.

To open a **Trace Memory Browser**, do one of the following:

- In the **Trace Statistics** window, double-click one of the memory locations listed in the **Memory** tab.
- In the Trace List, right-click a line that shows an access of the data address of interest. In the shortcut menu that appears, select **Browse Accesses of *data_address***.
- In the Debugger window, find a global or static variable that you are interested in accesses to. Right-click the variable name and select **Trace → Browse Traced Accesses**. Note that this only displays accesses of the first address of the variable. If the variable is a struct or class with multiple member variables, the **Trace Memory Browser** only displays accesses of the first one.
- In the Debugger command pane, enter the **tracebrowse** command. For information about the **tracebrowse** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



The screenshot shows a window titled "Trace Memory Browser - Address: _job". It contains a table of memory access data. The table has four columns: PC, Value, Access, and Total Instructions. The data is sorted by Total Instructions in descending order. The first row shows PC 0x00005824, Value 0x00000000, Access Write, and Total Instructions 536. The last row shows PC 0x00004fc0, Value 0x000001f8, Access Write, and Total Instructions 1261. The window also displays the address 0x00006b5c _job and 459 Total Accesses. At the bottom, it says "Analysis Complete".

Trace Memory Browser - Address: _job

Address: 0x00006b5c _job
459 Total Accesses

PC	Value	Access	Total Instructions
0x00005824	0x00000000	Write	536
0x00004f30	0x00000200	Write	1071
0x00004fb8	0x00000200	Read	1112
0x00004fc0	0x000001ff	Write	1114
0x00004fb8	0x000001ff	Read	1133
0x00004fc0	0x000001fe	Write	1135
0x00004fb8	0x000001fe	Read	1154
0x00004fc0	0x000001fd	Write	1156
0x00004fb8	0x000001fd	Read	1175
0x00004fc0	0x000001fc	Write	1177
0x00004fb8	0x000001fc	Read	1196
0x00004fc0	0x000001fb	Write	1198
0x00004fb8	0x000001fb	Read	1217
0x00004fc0	0x000001fa	Write	1219
0x00004fb8	0x000001fa	Read	1238
0x00004fc0	0x000001f9	Write	1240
0x00004fb8	0x000001f9	Read	1259
0x00004fc0	0x000001f8	Write	1261

Analysis Complete

The **Trace Memory Browser** contains a list of all the memory accesses to a specific memory location. From left to right, this list displays the following information for each memory access:

- Whether or not the instruction that caused the memory access is bookmarked. In the narrow, left-most column, you can add or remove a bookmark by clicking the vertical gray line or the flag, respectively.
- The PC and function in which the access occurred.
- The value that was read or written to this memory location.
- The type of access. This indicates whether the access was a read or a write.
- The cumulative time at which the access occurred. This column corresponds to the **Total Time**, **Total Cycles**, or **Total Instructions** column of the Trace List.

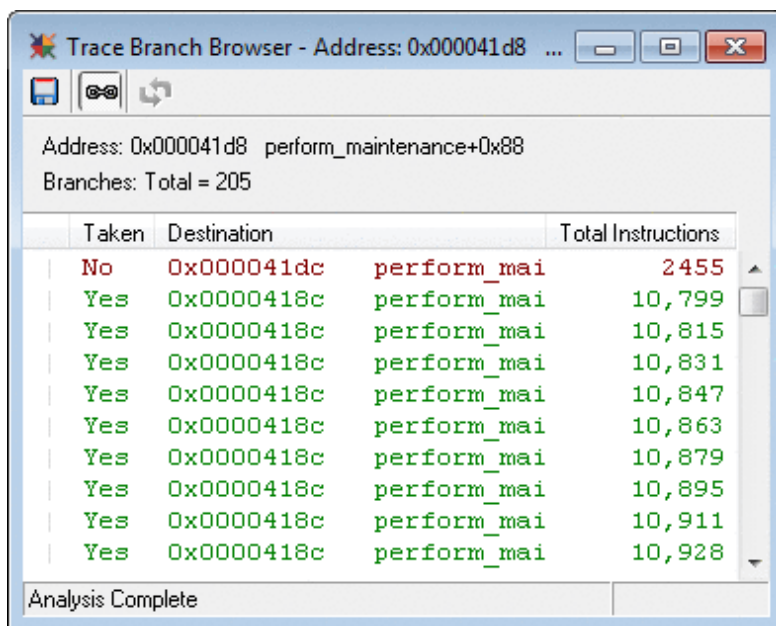
Initially, MULTI sorts the memory accesses by the cumulative time column. You can sort by any column other than the first one.

For information about the **Trace Memory Browser** toolbar, see “Trace Browsers Toolbar” on page 459.

The Trace Branch Browser

The **Trace Branch Browser** displays a list of all executions of a single branch instruction from your trace data.

To open a **Trace Branch Browser**, open the **Trace Statistics** window and double-click one of the branches listed in the **Branches** tab.



From left to right, the **Trace Branch Browser** displays the following information for each execution of the branch:

- Whether or not the instruction is bookmarked. In the narrow, left-most column, you can add or remove a bookmark by clicking the vertical gray line or the flag, respectively.
- Whether the branch was taken or not. A branch that is taken indicates that the execution flow of your program was altered at this point. For conditional branches, it generally indicates that the condition evaluated to true.
- The destination address of the branch. This is the PC of the next instruction that executed after each instance of the branch.
- The cumulative time at which the branch occurred. This column corresponds to the **Total Time**, **Total Cycles**, or **Total Instructions** column of the Trace List.

Initially, MULTI sorts the executions of the branch by the cumulative time column. You can sort by any column other than the first one.

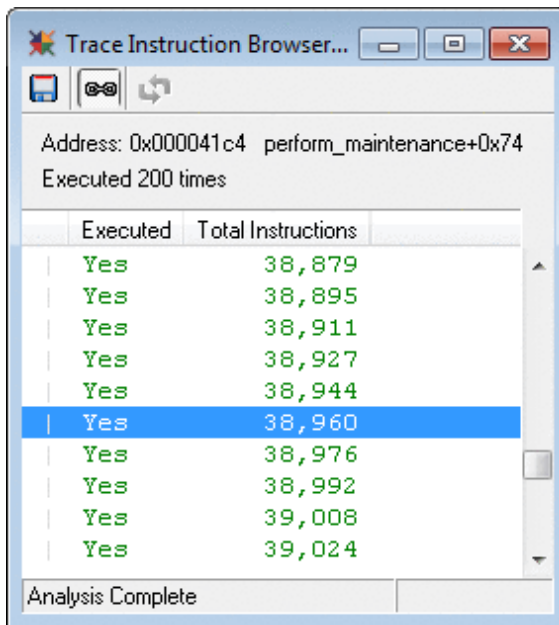
For information about the **Trace Branch Browser** toolbar, see “Trace Browsers Toolbar” on page 459.

The Trace Instruction Browser

The **Trace Instruction Browser** displays a list of all executions of a single instruction in the trace data.

To open a **Trace Instruction Browser**, do one of the following:

- In the Trace List, right-click the instruction you want to browse and select **Browse Executions**.
- In the Debugger, right-click the instruction of interest and select **Trace → Browse Traced Executions**.
- In the Debugger command pane, enter the **tracebrowse -line** command. For information about the **tracebrowse** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



From left to right, the **Trace Instruction Browser** displays the following information for each execution of the instruction:

- Whether or not the instruction is bookmarked. In the narrow, left-most column, you can add or remove a bookmark by clicking the vertical gray line or the flag, respectively.

- Whether the instruction was executed or not. This column only applies if the target supports conditional execution of instructions.
- The cumulative time at which the instruction was traced. This column corresponds to the **Total Time**, **Total Cycles**, or **Total Instructions** column of the Trace List.

Initially, MULTI sorts the executions of the instruction by the cumulative time column. You can sort by any column other than the first one.

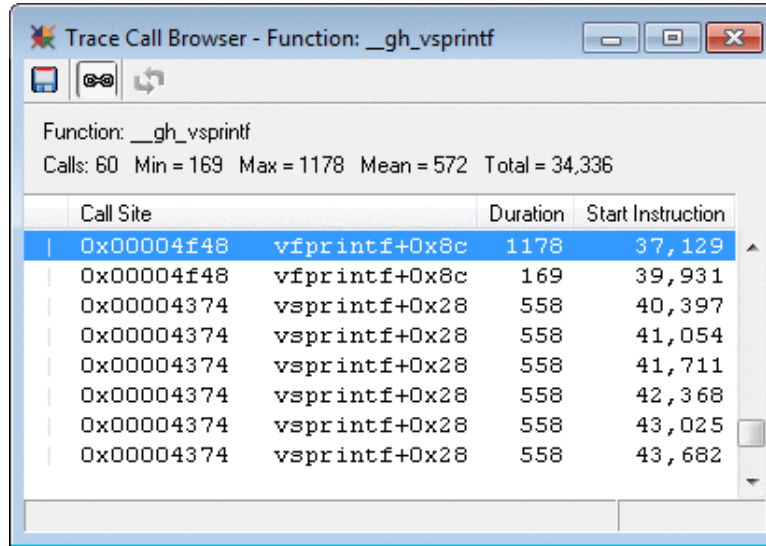
For information about the **Trace Instruction Browser** toolbar, see “Trace Browsers Toolbar” on page 459.

The Trace Call Browser

The **Trace Call Browser** displays a list of each call site of a single function in the trace data.

To open a **Trace Call Browser**, do one of the following:

- In the Trace List, right-click a function node and select **Browse Function Calls**.
- In the PathAnalyzer, right-click a function and select **Find Function References**.
- In the **Functions** tab of the **Trace Statistics** window, double-click a function, or select a function and click the **Details** button (.
- In the Debugger, right-click a function name and select **Trace → Browse Traced Calls**.
- In the Debugger command pane, enter the **tracebrowse** command. For information about the **tracebrowse** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



Trace Call Browser - Function: `__gh_vsprintf`

Function: `__gh_vsprintf`
 Calls: 60 Min = 169 Max = 1178 Mean = 572 Total = 34,336

Call Site	Duration	Start Instruction
<code>0x00004f48</code> <code>vfprintf+0x8c</code>	1178	37,129
<code>0x00004f48</code> <code>vfprintf+0x8c</code>	169	39,931
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	40,397
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	41,054
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	41,711
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	42,368
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	43,025
<code>0x00004374</code> <code>vsprintf+0x28</code>	558	43,682

From left to right, the **Trace Call Browser** displays the following information for each call of the function:

- Whether or not the call site of the function is bookmarked. In the narrow, left-most column, you can add or remove a bookmark by clicking the vertical gray line or the flag, respectively.
- The call site, which is the location that the function was called from. This will include an address and, if available, the function and offset of that address.
- The duration, which is the total time that the function was on the stack. This is counted from the time the function or its called functions were first observed to the time the function or its called functions were last executed. If trace data is available for the function's entire execution, this value is simply the time elapsed between the function's call and its return.
- The cumulative time at the start of the function. If the start of the function is not in the trace data, the time at which the function was first known to be on the call stack is used instead. This column corresponds to the **Total Time**, **Total Cycles**, or **Total Instructions** column of the Trace List.

Initially, MULTI sorts the calls of the function by the cumulative time column. You can sort by any column other than the first one.

**Note**

The following situations can cause the function start time, duration, and call site to be incorrect:

- If the trace buffer starts after the call to the function.
- If the trace buffer ends before the function returns.
- If an exception is taken while the function is executing. In this case, a single function call may be broken up into two separate calls in the list. The call site of the second one will be unknown.
- If there are gaps in the trace data due to on-chip trace buffer overflow or trace filtering.

For information about the **Trace Call Browser** toolbar, see the next section.

Trace Browsers Toolbar

The toolbar that appears at the top of the Trace Browsers contains the following buttons:

-  — Saves the list of instruction, branch, or function executions (whichever the Trace Browser lists) to a text file.
-  — Toggles synchronization of the selection in this window with the current position in TimeMachine.
-  — Resumes the search for instruction, branch, or function executions (whichever the Trace Browser lists) in the trace data if processing was stopped before completion and was not automatically resumed.

Viewing Trace Statistics

The **Trace Statistics** window generates and displays statistical information about your trace data. To open the **Trace Statistics** window, do one of the following:

1. Select **TimeMachine** → **Trace Statistics**.
2. In the Trace List or the PathAnalyzer, click the  button.
3. In the Trace List or the PathAnalyzer, select **Tools** → **Statistics**.
4. In the Debugger command pane, enter the **trace stats** command. For information about the **trace** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Trace Statistics** window is broken up into five tabs: **Summary**, **AddressSpaces**, **Memory**, **Branches**, and **Functions**. The **AddressSpaces** tab is only available if you collected trace data from an INTEGRITY application. The **Memory** tab is only available if your trace data includes the data addresses associated with memory access instructions.

When you trace an INTEGRITY application, there will be one or two drop-down lists at the bottom of the window. These drop-down lists allow you to select an AddressSpace and a task that apply to all the tabs (except the **AddressSpaces** tab, which displays summary information about all traced AddressSpaces and tasks).

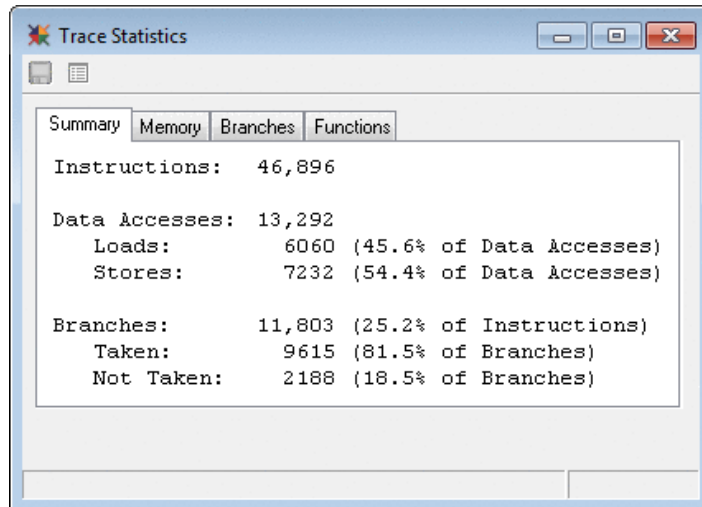


Note

The trace data collected from some targets does not include sufficient information to determine which INTEGRITY task was executing at any particular time (see “Dealing with Incomplete Trace Data” on page 411). With those targets there will not be a task drop-down list.

Summary

The **Summary** tab of the **Trace Statistics** window contains summary statistics for either the entire trace buffer or for traced instructions from the currently selected task or AddressSpace.



The information that this tab contains is detailed in the following table.

Item	Description
Instructions	The total number of instructions found in the trace buffer.
16 bit	The total number of 16-bit instructions that were executed as well as the percentage of total instructions that were 16-bit instructions. This item only appears when trace data was collected from a target that has both 32-bit and 16-bit instruction sets.
Cycles	The total number of cycles elapsed as well as the average number of cycles per instruction. This item only appears when cycle-accurate mode is enabled.
Data Accesses	The total number of data accesses.
Loads	The total number of loads as well as the percentage of data accesses that were loads.
Stores	The total number of stores as well as the percentage of data accesses that were stores.
Branches	The total number of branch instructions.
Taken	The total number of branch instructions that were taken, meaning that they caused a change-of-flow in your process. The percentage of total branch instructions that were taken is also printed.
Not Taken	The total number of branch instructions that were not taken, meaning that their conditions failed and execution continued on the next instruction. The percentage of total branch instructions that were not taken is also printed.

AddressSpace Statistics

When tracing an INTEGRITY target, the **AddressSpaces** tab will display statistics about each AddressSpace in which a task executed during the trace run.

Name	Instructions	Accesses	Loads	Stores	Branches	Taken
kernel	716422	255577	154005	101572	158905	107520
system	716350	255559	153993	101566	158881	107503
Idle	72	18	12	6	24	17
mpizzahut	10560	2998	1618	1380	2515	2251
Initial	10560	2998	1618	1380	2515	2251
mphonecompany	7015	1673	813	860	1630	1313
Initial	7015	1673	813	860	1630	1313
mengineer	37274	10794	5865	4929	9924	8025
Initial	37274	10794	5865	4929	9924	8025
minformation	3916	1058	609	449	980	825
Initial	3916	1058	609	449	980	825

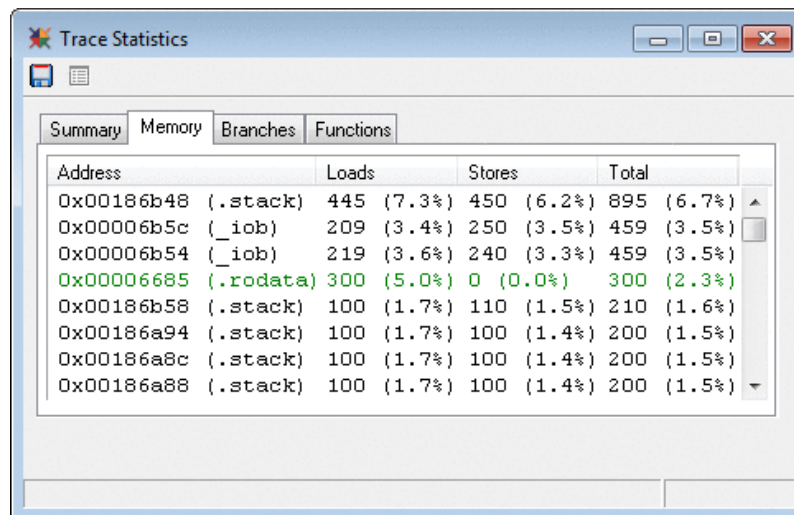
The following information will be displayed for each AddressSpace and, if task information is available, for each task:

Column	Description
Instructions	The total number of traced instructions from the task or AddressSpace.
16 bit	The total number of 16-bit instructions that were traced from the task or AddressSpace. This column only appears when trace data was collected from a target that has both 32-bit and 16-bit instruction sets.
Cycles	The total number of cycles spent executing instructions from the task or AddressSpace. This item only appears when cycle-accurate mode is enabled.
Accesses	The total number of data accesses performed by instructions from the task or AddressSpace.
Loads	The total number of loads performed by instructions from the task or AddressSpace.
Stores	The total number of stores performed by instructions from the task or AddressSpace.
Branches	The total number of branch instructions traced from the task or AddressSpace.
Taken	The total number of branch instructions from the task or AddressSpace that were taken, meaning that they caused a change-of-flow in your process.

Memory Statistics

The **Memory** tab of the **Trace Statistics** window contains a list of all the addresses that were read or written during the trace run. This list displays the following information for each memory location accessed:

- The number of loads from this address as well as the percentage of all loads that were from this address. For addresses that were only loaded from, the text of the row is green.
- The number of stores to this address as well as the percentage of all stores that were to this address. For addresses that were only stored to, the text of the row is red.
- The total number of accesses to this address as well as the percentage of all accesses that were to this address.



Address	Loads	Stores	Total
0x00186b48 (.stack)	445 (7.3%)	450 (6.2%)	895 (6.7%)
0x00006b5c (_iob)	209 (3.4%)	250 (3.5%)	459 (3.5%)
0x00006b54 (_iob)	219 (3.6%)	240 (3.3%)	459 (3.5%)
0x00006685 (.rodata)	300 (5.0%)	0 (0.0%)	300 (2.3%)
0x00186b58 (.stack)	100 (1.7%)	110 (1.5%)	210 (1.6%)
0x00186a94 (.stack)	100 (1.7%)	100 (1.4%)	200 (1.5%)
0x00186a8c (.stack)	100 (1.7%)	100 (1.4%)	200 (1.5%)
0x00186a88 (.stack)	100 (1.7%)	100 (1.4%)	200 (1.5%)

To display detailed information about the memory accesses to a specific location, do one of the following:

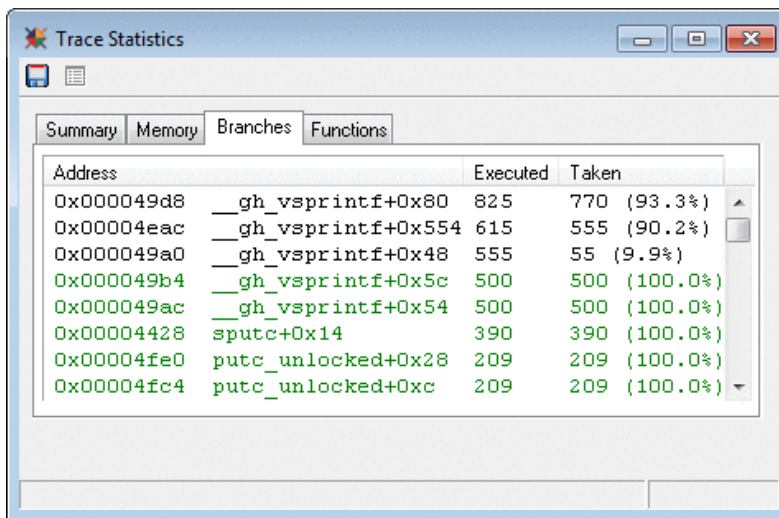
- Double-click the corresponding line.
- Select an address and click the **Details** button ().

A **Trace Memory Browser** opens for the selected memory location. See “The Trace Memory Browser” on page 452.

Branch Statistics

The **Branches** tab of the **Trace Statistics** window contains a list of all of the branches that were collected during the trace run. This list displays the following information about each branch executed:

- The number of times that the branch was executed.
- The number of times that the branch was taken. This indicates the number of times that this branch changed the execution flow of your program. For branches that were never taken, the text of the row is red. For branches that were always taken, the text of the row is green.



Address		Executed	Taken	
0x000049d8	__gh_vsprintf+0x80	825	770 (93.3%)	
0x00004eac	__gh_vsprintf+0x554	615	555 (90.2%)	
0x000049a0	__gh_vsprintf+0x48	555	55 (9.9%)	
0x000049b4	__gh_vsprintf+0x5c	500	500 (100.0%)	
0x000049ac	__gh_vsprintf+0x54	500	500 (100.0%)	
0x00004428	sputc+0x14	390	390 (100.0%)	
0x00004fe0	putc_unlocked+0x28	209	209 (100.0%)	
0x00004fc4	putc_unlocked+0xc	209	209 (100.0%)	

To display detailed information about a specific branch, do one of the following:

- Double-click the corresponding line.
- Select a branch and click the **Details** button ().

A **Trace Branch Browser** opens for the selected branch. See “The Trace Branch Browser” on page 454.

Function Statistics

The **Functions** tab of the **Trace Statistics** window contains a list of all of the functions that were called during the trace run. This list displays the following information about each function called:

- The number of times the function was called.
- The function's shortest run time during the trace run.
- The function's average run time during the trace run. (This is a simple average.)
- The function's longest run time during the trace run.

All run times are reported in the units available from the trace data.



Note

Function run times shown on this tab may be less than the actual function run times in the following cases:

- If the trace buffer starts after the call to the function.
- If the trace buffer ends before the function returns.
- If an exception is taken while the function is executing. In this case a single function call may be broken up into two separate function calls in the list.
- If there are gaps in the trace data due to on-chip trace buffer overflow or trace filtering.

Function	Calls	Min	Mean	Max
0x00004958 __gh_vspr	60	169 inst	572 inst	1.18 kinst
0x00004850 unsout	55	195 inst	225 inst	238 inst
0x00004570 __gh_stro	55	171 inst	185 inst	196 inst
0x00004538 __gh_outs	110	20 inst	20 inst	20 inst
0x00004508 __gh_xxde	55	5 inst	20 inst	23 inst
0x0000442c printf	10	494 inst	994 inst	1.50 kinst
0x00004414 sputc	390	6 inst	6 inst	6 inst
0x0000438c sprintf	50	574 inst	603 inst	609 inst

To display detailed information about a specific function call, do one of the following:

- Double-click the corresponding line.
- Select an address and click the **Details** button ().

A **Trace Call Browser** opens for the selected function. See “The Trace Call Browser” on page 457.

The TimeMachine API

The TimeMachine API is a TimeMachine component that enables you to perform custom analysis on your trace data. The TimeMachine API consists of a dynamic library that you can link against a C program, a C++ program, or a script written in any scripting language that can import functions from a DLL. You can access your trace data either from a saved trace file or by connecting to a running trace session.

The TimeMachine API files are located at *ide_install_dir/timemachine_api*. In this directory, you will find an example C/C++ program and Python scripts that use the TimeMachine API. These examples may be useful to you both as a reference and as a starting point for creating your own custom analysis tools. For more information about these examples, see “Example Python Scripts” on page 471 and “The Example C/C++ Application” on page 473. For information about specific API functions, see the **timemachine_api.h** header file. For information about data stored in the trace stream, see **ts_packet.h**.

The TimeMachine API provides two interfaces that allow you to access your trace data:

- A live interface (`tm_*` functions) — Connects to a running MULTI Debugger session and accesses trace data by reading it from the active MULTI trace session.
- A file interface (`tm_file_*` functions) — Allows you to save MULTI trace data to a file and then analyze it later. Alternatively, you can create files of saved trace data outside of MULTI and then load them in MULTI for analysis.

Both of these interfaces allow you to access the trace packet stream and then perform custom analysis. In addition, the live interface enables you to look up symbol values for your program, which provide useful trace analysis information. Applications and scripts using the live interface are usually launched from the MULTI Debugger with the **trace api** command. (For information about the **trace** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.) Applications and scripts using the file interface are usually launched outside of MULTI because they do not require a connection to an active trace session.



Note

If you want to access the TimeMachine API using Python, you must install Python and the ctypes Python module. Some of the Python examples also require that you install the Tkinter Python module and Tcl/Tk.

Accessing Trace Data via the Live TimeMachine Interface

Before you can run an application or script that uses the live interface, you must first collect some trace data in the MULTI Debugger. For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.

To launch a C/C++ application, enter the following command in the Debugger command pane:

```
> trace api application_name
```

To launch a Python script, enter the following command in the Debugger command pane:

```
> trace api path_to_Python_application path_to_script
```

where:

- *path_to_Python_application* is the full path to the installed Python interpreter (a Python 2.3.3 installation is included with the MULTI IDE installation).
- *path_to_script* is the full path to the Python script.

The list below specifies the functions that your application or script must call to establish a live TimeMachine connection. The functions must be called in the specified order.

1. Initialize the TimeMachine API by:
 - C/C++ — Calling `tm_init_arg(&argc, argv)`.
 - Python — Instantiating the `timemachine_api` class (`my_timemachine_api = timemachine_api()`).
2. Establish a connection to the MULTI Debugger by:
 - C/C++ — Calling `tm_connect(0)`.
 - Python — Calling the `live_connect()` function of the `timemachine_api` class instance.

When the connection is established, you can call live interface functions to access the number of trace packets, access an array of trace packets, and control the trace collection process. In addition, you can query the Debugger for information about symbols and addresses in your program, which enables you to easily correlate your results back to your system's source code. You can find a comprehensive list of live interface functions in the **timemachine_api.h** header file located at *ide_install_dir/timemachine_api*. For information about data stored in the trace stream, see **ts_packet.h**.

Your application must call `tm_process_events()` regularly—typically in the main event loop—to ensure that events are handled and callbacks are called in a timely manner. TimeMachine API live interface applications that do not call

`tm_process_events()` regularly can cause the MULTI Debugger to appear to hang.

Accessing Trace Data via the TimeMachine File Interface

To run an application or script that accesses saved trace data from MULTI, perform the following steps:

1. Collect some trace data in the MULTI Debugger. For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.
2. Save the trace data. For information about saving trace data, see “Saving and Loading a Trace Session” on page 448.
3. You may optionally close the Debugger at this point.
4. Launch the application or script. You can do this outside of MULTI.

The list below specifies the functions that your application or script must call to open a file of saved trace data. The functions must be called in the specified order.

1. Initialize the TimeMachine API by:
 - C/C++ — Calling `tm_init(ide_install_dir)`.
 - Python — Instantiating the `timemachine_api` class (`my_timemachine_api = timemachine_api()`).
2. Open the trace file by:
 - C/C++ — Calling `tm_file_open(trace_file_name)`.
 - Python — Calling the `trace_file_open(trace_file_name)` function of the `timemachine_api` class instance.

When the trace file is opened, you can call file interface functions to access the number of trace packets and access an array of trace packets. For a comprehensive list of file interface functions, see the **timemachine_api.h** header file located at *ide_install_dir/timemachine_api*. For information about data stored in the trace stream, see **ts_packet.h**.

Creating Trace Data via the TimeMachine File Interface

To run an application or script that creates trace data, perform the following steps:

1. Launch the application or script. You can do this outside of MULTI.
2. Launch the MULTI Debugger on the program associated with the trace data you created.
3. Connect to a simulator.
4. Load the trace data file in the Debugger. For information about loading trace, see “Saving and Loading a Trace Session” on page 448.

The list below specifies the functions that your application or script must call to create a file of saved trace data. The functions must be called in the specified order.

1. Initialize the TimeMachine API by:
 - C/C++ — Calling `tm_init(ide_install_dir)`.
 - Python — Instantiating the `timemachine_api` class (`my_timemachine_api = timemachine_api()`).
2. Create a trace file by:
 - C/C++ — Calling `tm_file_create(trace_file_name)`.
 - Python — Calling the `trace_file_create(trace_file_name)` function of the `timemachine_api` class instance.
3. Specify the trace data properties of the file by:
 - C/C++ — Calling `tm_file_set_info(trace_file, info, sizeof(TM_DATA_INFO))`.
 - Python — Calling the `set_info(info)` function of the `timemachine_trace_file` class instance.
4. Add trace packets to the file by:
 - C/C++ — Calling `tm_file_append_packet(trace_file, trace_packet)` for each packet to be added to the file.
 - Python — Calling the `append_packet(trace_packet)` function of the `timemachine_trace_file` class instance for each packet to be added to the file.

5. Save and close the trace file by:

- C/C++ — Calling `tm_file_save(trace_file)` and then calling `tm_file_close(trace_file)`.
- Python — Calling the `close()` function of the `timemachine_trace_file` class instance.

For a comprehensive list of file interface functions, see the **timemachine_api.h** header file located at *ide_install_dir/timemachine_api*. For information about data stored in the trace stream, see **ts_packet.h**.

Example Python Scripts

The MULTI IDE installation includes Python script examples that use the TimeMachine API:

- **timemachine_api_test.py**
- **timemachine_api_test2.py**
- **timemachine_api_test3.py**
- **timemachine_api_file_test.py**

These example scripts are located at:

ide_install_dir/timemachine_api/example_python

For information about how to launch a Python script, see “Accessing Trace Data via the Live TimeMachine Interface” on page 467.

The following sections provide more detailed instructions about how to run these examples and information about what each one demonstrates.

Live Interface Python Examples

The live interface examples demonstrate how you can use Python scripts to perform custom analysis from the MULTI Debugger.

Before you can run these examples, you must first collect some trace data in the MULTI Debugger. For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405. If you do not already have a program in

which to collect trace data, you can use the Project Manager to access the **TimeMachine Debugging** demo project. For information about creating a project, see Chapter 1, “Creating a Project” in the *MULTI: Managing Projects and Configuring the IDE* book.

After you have collected trace data, you can launch the following example scripts:

- **timemachine_api_test.py** — Displays trace packets and counts packets, instructions, and data accesses.
- **timemachine_api_test2.py** — Displays trace packets using Tcl/Tk.
- **timemachine_api_test3.py** — Looks up the address of a symbol.

For information about how to launch a Python script, see “Accessing Trace Data via the Live TimeMachine Interface” on page 467.

The File Interface Python Example

The file interface example demonstrates how you can use a Python script to perform custom analysis on saved trace data. This example prints out trace packets stored in the trace file, properties of the trace data, and statistics calculated on the trace data.

To run this example, perform the following steps:

1. Collect some trace data in the MULTI Debugger. For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.
2. Save the trace data as **timemachine_api_file_test.trs**. Save the file to the **ide_install_dir/timemachine_api/example_python** directory. For information about saving trace data, see “Saving and Loading a Trace Session” on page 448.
3. You may optionally close the Debugger at this point.
4. Launch the application or script in one of the following ways:
 - Windows — From the command prompt, run the following command in the **ide_install_dir\timemachine_api\example_python** directory:

```
> python timemachine_api_file_test.py
```

- Linux/Solaris — In a Linux/Solaris shell, run the following command in the *ide_install_dir/timemachine_api/example_python* directory:

```
> python timemachine_api_file_test.py
```

The Example C/C++ Application

The C/C++ example is located at:

ide_install_dir/timemachine_api/example_c/timemachine_api_test.cc.

You can use this example source code to build a native application that links against the TimeMachine API library. To build a native application, you must use native development tools for your host operating system, such as Green Hills MULTI for x86 Linux Native, Microsoft Visual C++, or GNU GCC. After building the native application, you can use MULTI to invoke it and to print basic statistics for any trace data you have collected.

The C/C++ example provides you with a good starting point from which to create a TimeMachine API application for your own custom analysis of trace data.

To set up this example, perform the following steps:

1. Use your native development tools to create a project. Then adapt the following steps to your native tools:
 - Add the **timemachine_api_test.cc** C++ source file to your project. This file is located at *ide_install_dir/timemachine_api/example_c*.
 - Add *ide_install_dir/timemachine_api* as an include directory. This is the location of the **timemachine_api.h** and **ts_packet.h** header files, which are included by **timemachine_api_test.cc**.
 - Windows — Link against the **timemachine_api.lib** library file, which is located at *ide_install_dir*.
 - Linux/Solaris — Link against the **timemachine_api.so** library file, which is located at *ide_install_dir*.
2. Build the native project you just created.

3. Set aside the native application you just built, and use the MULTI Debugger to collect trace data from your target. For information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.
4. Retrieve the trace data. For more information, see “Retrieving Trace Data” on page 408.
5. Enter the following command in the MULTI Debugger command pane to run the application built in step 2:

```
> trace api application_name
```

Viewing Trace Events in the EventAnalyzer

MULTI can convert trace data into an EventAnalyzer log that you can view in the MULTI EventAnalyzer. This allows the MULTI EventAnalyzer to be used without instrumenting the code, which can often cause changes in process behavior.



Note

Generating EventAnalyzer information is only available when using an operating system that supports the MULTI EventAnalyzer with trace.



Note

The EventAnalyzer requires that your trace data either include data accesses or task switch markers. If neither is available, all events are gathered in a common task denoted **Unknown Context**. **Unknown Context** is also used to display events at the beginning of the trace before the current task is known.

To generate EventAnalyzer information from the current trace data and open the EventAnalyzer on that information, do one of the following:

- In the Trace List or the PathAnalyzer, click the **EventAnalyzer** button (). (If the operating system does not support the MULTI EventAnalyzer, this button will be dimmed.)
- In the Debugger, select **TimeMachine** → **EventAnalyzer**. (If the operating system is not supported by the EventAnalyzer, this option will be dimmed.)

- In the Debugger command pane, enter the **tracemevsys** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

MULTI will display a progress window while it converts the trace data. When the conversion is complete, the MULTI EventAnalyzer will launch automatically. For information about the MULTI EventAnalyzer, see the *EventAnalyzer User's Guide* or the documentation about using the MULTI EventAnalyzer for ThreadX in the *MULTI: Developing for ThreadX* book.



Note

The EventAnalyzer does not display relevant information in the **More Info** field of the **Event View** window or in the tooltip that is displayed when you click an event. All event parameters except for PC are displayed as 0 (zero). To view these parameters, use the TimeMachine Debugger.

In addition, task statuses are not displayed. Tasks are either in a running or non-running state.

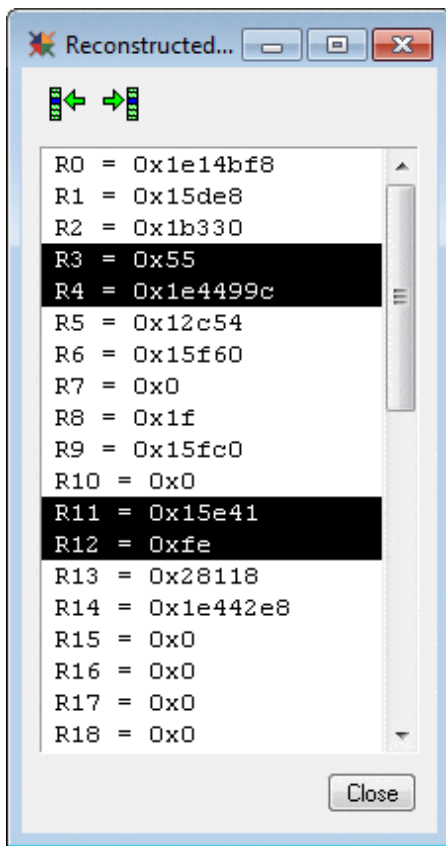
Using Trace Data to Profile Your Target

MULTI can convert trace data into performance analysis data that can be displayed in the **Profile** window. Profiling data generated from trace data is much more complete, in terms of which instructions are represented, than data generated by profiling methods that use sampling. (Sampling methods are usually timer-driven and can miss important hot spots in a process.) MULTI can generate program counter samples, call count with call graph data, and coverage analysis data from trace data. For information about these data types, see “Types of Profiling Data” on page 354.

For information about generating profiling data from trace data, see “Collecting Profiling Data for a Trace-Enabled Target” on page 358.

Viewing Reconstructed Register Values

If your trace data includes the addresses and values associated with traced memory access instructions, MULTI is capable of inferring register values from the trace data (see “Dealing with Incomplete Trace Data” on page 411). To view the register values that MULTI was able to reconstruct at a particular point in your trace data, select an instruction and either click the  button or select **Tools** → **Registers** from the Trace List or the PathAnalyzer. This opens the **Reconstructed Registers** window.



The **Reconstructed Registers** window displays a list of all the registers that MULTI can reconstruct from the trace data.

If a register's value prior to executing the selected instruction can be inferred, the value will be displayed. If not, ? will be displayed to indicate that the register value is unknown.

In the **Reconstructed Registers** window, you can navigate to the previous or next write to the selected register:

- To navigate to the next write to the selected register, either click  or right-click the register to display a shortcut menu and select **Next Write to this Register**.
- To navigate to the previous write to the selected register, either click  or right-click the register to display the shortcut menu and select **Previous Write to this Register**.

Chapter 20

Advanced Trace Configuration

Contents

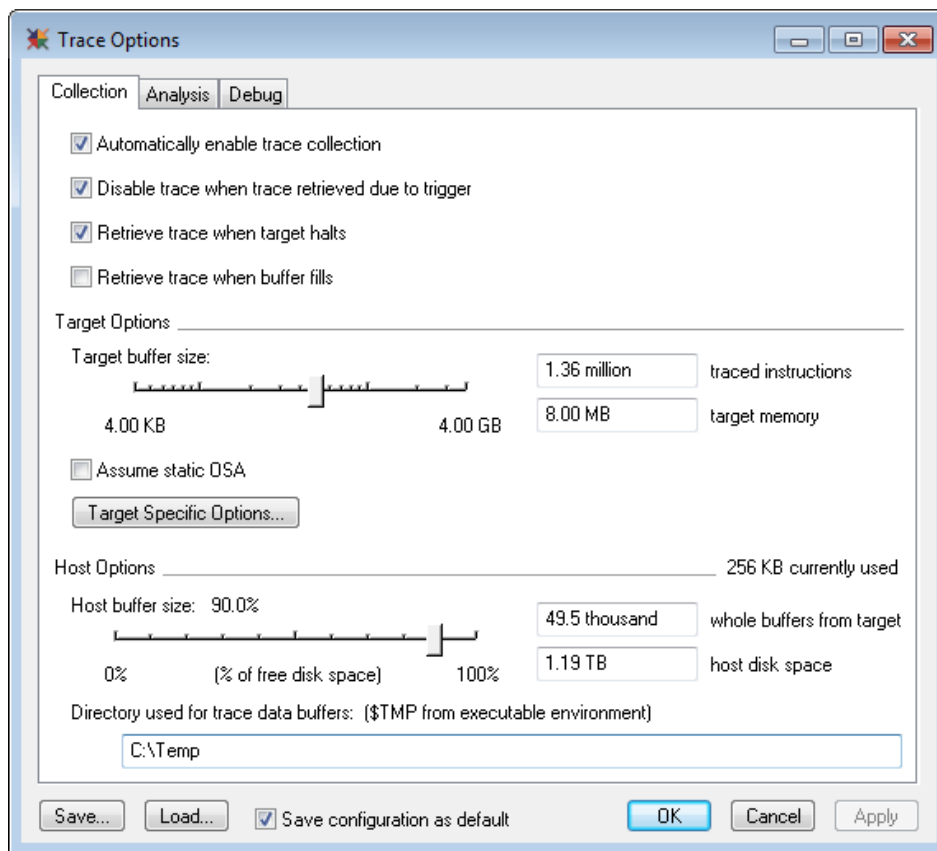
The Trace Options Window	480
Configuring Target-Specific Trace Options	488
Viewing Target-Specific Information	489
Configuring Trace Collection	489

The Trace Options Window

The **Trace Options** window allows you to configure many aspects of trace collection, retrieval, display, and deletion. To open the **Trace Options** window, do one of the following:

- In the Trace List or the PathAnalyzer, select **Config** → **Options**.
- In the Debugger, select **TimeMachine** → **Trace Options**.
- In the Debugger command pane, enter the **trace options** command. For more information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

A sample **Trace Options** window is shown next.



The Collection Tab

The following table describes each option available from the **Collection** tab of the **Trace Options** window. Unless otherwise stated, all descriptions of **Trace Options** window toggle options explain the behavior of the option when enabled.

Collection Option	Description
Automatically enable trace collection	Automatically enables trace collection when you establish a connection to a target that supports trace. By default, this option is selected.
Disable trace when trace retrieved due to trigger	Disables additional trace collection when trace is automatically retrieved due to a trigger event. Disabling this option causes MULTI to automatically re-enable trace collection after trace is retrieved due to a trigger. This can be useful for collecting a large amount of trace data consisting of independent trace buffers each centered around a trigger. By default, this option is selected.
Retrieve trace when target halts	Automatically downloads collected trace data when the target halts. This means that when your target halts, trace data is immediately available for use with TimeMachine. Note that trace data is available for downloading only if trace collection is enabled. By default, this option is cleared.
Retrieve trace when buffer fills	Automatically downloads collected trace data when the target's trace buffer fills. Enabling this option results in trace data periodically being downloaded when the target is running and when trace collection is enabled. By default, this option is cleared.

Collection Option	Description
Target buffer size	With SuperTrace Probe v3 targets: Specifies the number of bytes of trace data to retrieve from the end of the SuperTrace Probe's trace buffer. The SuperTrace Probe v3 always uses all of its trace RAM as a large circular buffer. However, since it takes a long time to retrieve all of the data, MULTI can be configured to initially retrieve only a portion of the data from the end of the buffer. For more information, see “Retrieving Trace Data from a SuperTrace Probe v3” on page 409. Note: Triggering on the SuperTrace Probe v3 uses the configured Target buffer size rather than all available trace RAM. For example, if the Target buffer size is set to 16 MB and the trigger position is set to 50%, trace collection stops when 8 MB of data has been collected after the trigger has occurred. For more information, see “Configuring Trace Collection” on page 489. With other targets: Specifies the number of bytes to use for collecting trace data on the target or probe. The most recent trace data of the quantity specified here is buffered while trace collection is enabled. When you need a large, contiguous capture of trace data, select a large value. Large captures may take longer to download and analyze, and they require more disk space to cache, so choosing the largest possible buffer size is rarely preferable. For all targets: The traced instructions field displays an estimate of the number of instructions that will be traced by a full trace buffer of the size specified. The target memory field displays the value of the target buffer size that is currently specified. Moving the Target buffer size slider left and right decreases and increases this value, respectively.

Collection Option	Description
Assume static OSA	Retrieves/synchronizes OSA data only for the first trace data retrieval after each time the program is loaded. When disabled [default], OSA data is retrieved every time trace data is retrieved. This option should be enabled only if the set of tasks and AddressSpaces in a system is static. Enabling this option is preferable on hardware configurations for which retrieving OSA data from the target takes a prohibitively long time. When new trace data is retrieved and analyzed without new OSA data, the trace data may be decoded using the wrong opcodes. For more information, see “Collecting Operating System Trace Data” on page 406. You can manually retrieve OSA data by executing the trace updateosa command. For more information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Changes you make to this option take effect the next time trace data is retrieved. For information about OSA data, see Chapter 26, “Freeze-Mode Debugging and OS-Awareness” on page 605.
Target Specific Options	Opens a configuration window that contains target-specific options. This button does not appear if the Trace Options window contains a target-specific tab. For more information, see the documentation about target-specific trace options in the <i>Green Hills Debug Probes User's Guide</i> or, if you are using a V850 target, the documentation about V850 trace options in the <i>MULTI: Configuring Connections</i> book.

Collection Option	Description
Host buffer size	Specifies the percentage of free disk space to use for storing collected trace data on your computer. When this space has been filled, older trace data is deleted to make space for newly collected trace data. A heuristic, which considers triggers, bookmarks, and the currently selected instruction, selects which previous downloads are deleted first. The whole buffers from target field displays the estimated number of full buffers (of the size specified by the Target buffer size slider) that can be stored on the host (in the space currently specified by the Host buffer size slider). The value in this field turns red if the host buffer size is estimated to be too small. The size is estimated to be too small when a full buffer from the target would be larger than the specified size. The host disk space field displays the host buffer size that is currently specified. Moving the slider left and right decreases and increases this value, respectively. The host buffer is only temporary storage and is not saved across sessions. However, you may choose to save the trace data that has been collected. For more information, see “Saving and Loading a Trace Session” on page 448.
Directory used for trace data buffers	Displays the location on your computer of the temporary files that store downloaded trace buffers. The environment variable that specifies this path is shown above the text box. To change the directory of the temporary trace files, change the value of the environment variable TMP (Windows) or TMPDIR (Linux/Solaris), and restart MULTI. To collect a large amount of trace data, the specified directory must be on a drive with several gigabytes of free space.

The Analysis Tab

The following table describes each option available from the **Analysis** tab of the **Trace Options** window.

Analysis Option	Description
Attempt to reconstruct gaps in trace data	Enables reconstruction of trace data that was lost due to FIFO overflows. In many cases it is possible to use TimeMachine technology to examine the trace data before and after an overflow and determine which instructions were executed during the overflow. In addition to reconstructing missing instructions, this feature also attempts to determine the addresses and values of data accesses performed by those instructions. Reconstructed trace data is colored blue in the Trace List. Depending on the code being traced, the frequency of overflows, and the number of instructions that are typically lost per overflow on your target, you can expect to see between 50 and 90 percent of FIFO overflows reconstructed. For more information, see “Dealing with Incomplete Trace Data” on page 411. This option is only available with certain targets. By default, this option is selected.

Analysis Option	Description
Aggressiveness	Controls the aggressiveness of the gap reconstruction algorithm. If the gap reconstruction algorithm is allowed to make certain assumptions, it can often reconstruct the missing trace data more successfully. This option allows you to enable specific assumptions in the gap reconstruction algorithm. These assumptions are generally correct, but when an incorrect assumption is made, incorrect reconstruction of the missing trace data can occur. The three aggressiveness levels are: • Level 0 — No Assumptions — No assumptions are made. • Level 1 — Partial Reconstruction — [default] Partial reconstruction of gaps in the trace data is allowed. This means that even if the entire gap cannot be reconstructed, any instructions that can be reconstructed are inserted into the trace. The assumption being made is that an interrupt or exception did not occur during the trace gap. • Level 2 — Don't Invalidate — The gap reconstruction algorithm does not invalidate the register and memory values it has inferred from earlier trace data when it fails to reconstruct a gap. The assumption being made is that the missing instructions did not modify any of those register or memory values. This option is only available if Attempt to reconstruct gaps in trace data is enabled.
Abort processing on opcode failure	Aborts trace decompression when a PC is encountered for which the opcode cannot be identified by using binary image or target memory reads. Reading from target memory is not attempted if the option Read unknown opcodes from target (may halt target) (below) is disabled. By default, this option is selected. It can only be cleared if Read unknown opcodes from target (may halt target) is also cleared.
Read unknown opcodes from target (may halt target)	Allows opcodes necessary for trace decompression to be read from target memory. Opcodes are only read from the target if they cannot be found in the ELF file. The target may be briefly halted to allow target memory to be read. This behavior may interfere with your target process but is necessary in order to reconstruct trace data for code that was not downloaded by MULTI. By default, this option is cleared.
Packets to analyze for register reconstruction	Specifies the maximum number of packets that should be analyzed at any one time to reconstruct the state of registers and memory for TimeMachine. Increasing this number lengthens the delay required for analyzing packets and increases the amount of memory used by the tools, but also results in a more complete reconstruction of the state of registers and memory.

Analysis Option	Description
Stop at discontinuities	Stops the TimeMachine Debugger when it encounters discontinuities in the trace data. When this happens, the message <code>Stopped by discontinuity in trace data</code> is printed in the MULTI command pane. Different types of discontinuities occur in trace data. Some common types are FIFO overflows and gaps between subsequent trace runs. For more information, see “Dealing with Incomplete Trace Data” on page 411. By default, this option is selected.

The Debug Tab

The following table describes each option available from the **Debug** tab of the **Trace Options** window.

Debug Option	Description
Display raw trace packets	Interleaves information about raw compressed trace with instructions in the Trace List. You should not enable this option unless instructed to do so by Green Hills Technical Support. This option is only available for some hardware trace targets. By default, this option is cleared.
Enable debug logging	This option is for debugging only. You should not enable this option unless instructed to do so by Green Hills Technical Support. This option is only available for some trace targets. By default, this option is cleared.

The Target-Specific Tab

Each option available from the target-specific tab (if any) of the **Trace Options** window is described in the documentation pane in the right of the window and also in the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide* or, if you are using a V850 target, in the documentation about V850 trace options in the *MULTI: Configuring Connections* book.

To set an option on this tab, select the option from the list on the left, and modify its value in the upper-right corner of the window.



Tip

You can quickly set toggle options (**On/Off**) by double-clicking them.

Modified options are highlighted until you click **Apply**.

Action Buttons

If the **Save configuration as default** option is selected when you click **OK** or **Apply**, your current settings in the **Trace Options** and **Set Triggers** windows, along with your current target-specific settings, are applied to your current trace session and are saved and automatically loaded each time you use the trace tools. Click the **Save** button to save these settings to a file so that you can load them later by clicking the **Load** button.

Configuring Target-Specific Trace Options

Many targets have target-specific trace configuration options. To configure the options specific to your target (if any):

- Click the button labeled **Target Specific Options** on the **Collection** tab of the **Trace Options** window

or

Click the target-specific tab (the fourth tab) in the **Trace Options** window.

If your target supports target-specific trace configuration options, the **Target Specific Options** button or the target-specific tab will be available, but not both.

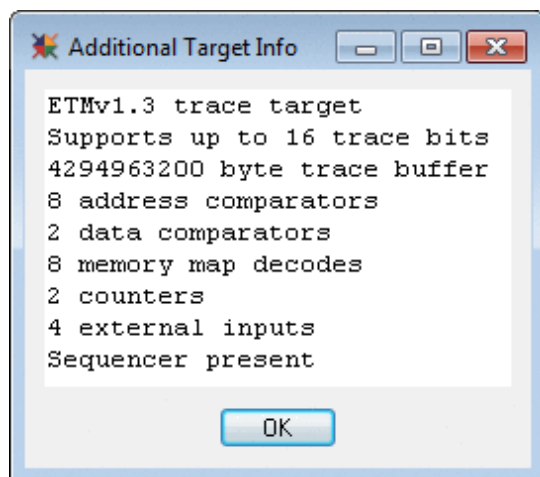
- Select **Config** → **Target Specific Options** in the Trace List or the PathAnalyzer. This menu item opens the same dialog box as the **Target Specific Options** button and is only available when the button is available.

For information about target-specific options, see the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide* or, if you are using a V850 target, the documentation about V850 trace options in the *MULTI: Configuring Connections* book.

Viewing Target-Specific Information

To view the capabilities of the hardware target that you are connected to, select **Config → Show Target Info** from the Trace List or the PathAnalyzer. This opens the **Additional Target Info** window, which displays information about the specific hardware device you are connected to.

For example, when you are connected to an ARM ETM target, this window displays information about triggering resources available, the maximum width of the trace port, and the size of the trace buffer.



Configuring Trace Collection

You can configure trace collection by setting triggers and other target-specific events. Trace collection systems generally have a circular buffer that wraps around when it fills. A trigger event controls when the trace collection system stops collecting data. When a trigger event is encountered, the system continues collecting data until a user-specified percentage of the buffer is filled.

For example, when a trigger is configured to occur in the center of the trace buffer, the trace collection system continues to collect data until 50 percent of the buffer has been filled after the trigger occurs. In this way, the trigger is always present in the trace buffer after it is encountered. When trace collection stops as a result of a trigger event, trace is automatically retrieved. (For information about other ways to control trace collection and retrieval, see “Managing Trace Data” on page 405.)

The trigger location is automatically bookmarked so that it is easy to find. The default name for trigger bookmarks is “Trigger,” and they are colored red by default. The trigger location is also automatically selected when trace is first collected. All trace displays initially center on the trigger.

On most architectures, the bookmarked instruction is not the exact instruction that caused the trigger. The trigger bookmark could be as many as several thousand instructions before or after the instruction that caused the trigger. This is due to the fact that most trace architectures output trace data in such a way that it is impossible to determine the exact instruction that caused the trigger.

You can use Debugger shortcut menus or the **Set Triggers** window to set the trigger for your trace run. For more information, see “The Set Triggers Window” on page 493.



Note

On INTEGRITY, it is not possible to configure trigger and trace events in a specific AddressSpace. The Debugger shortcut menus, commands, and the **Set Triggers** window only allow you to configure triggers and trace events for the kernel AddressSpace. However, most trace hardware and the Green Hills simulators respond to triggers and trace events when the associated virtual addresses are executed or accessed in any AddressSpace. For example, most trace hardware configured to trigger upon execution of address `0x1000` will trigger upon execution of `0x1000` in any AddressSpace. Refer to your processor's documentation for details about how triggers interact with virtual AddressSpaces.



Note

It may be necessary to halt certain targets before changes to trace triggers will take effect.

In addition to trigger configuration, many trace architectures also allow you to configure when different types of trace data will be generated. For example, you may be able to configure your target to only trace certain functions or to only trace data accesses to a range of addresses corresponding to memory-mapped registers. This can be useful if your target has very limited trace bandwidth and you know exactly what you are looking for, or if you are only interested in tracing a subset of the code running on the target. However, the usefulness of MULTI's high-level trace analysis tools such as the TimeMachine Debugger and PathAnalyzer degrades

rapidly as trace data is suppressed. For more information, see “Dealing with Incomplete Trace Data” on page 411.

Configuring Trace Directly from MULTI

When you are connected to a target that supports trace, the following trace options appear in the shortcut menu that opens when you right-click in the Debugger's source pane. Each of these options configures trace hardware and immediately enables tracing.



Note

These options are not available with all targets.

Right-click	Then select	Effect
Any executable line in the source pane	Trace → Trace Around This Line	Sets the trigger event to be any execution of the selected line. You can also use the traceline Debugger command to achieve this effect. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
A function	Trace → Trace This Function	Enables trace only when executing the selected function. Tracing will not occur when executing subfunctions. You can also use the tracefunction Debugger command to achieve this effect. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
A function	Trace → Trace Function Interval	Allows you to create conditions that enable and disable trace based on executions in specific functions. For more information, see “Specifying a Trace Function Interval” on page 492.

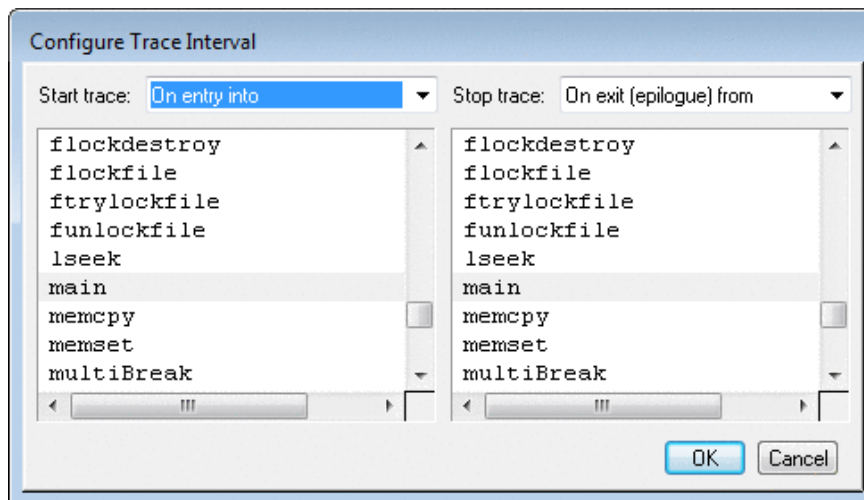
Right-click	Then select	Effect
A global variable	Trace → Trace Around Data Access	Sets the trigger event to be any access of the selected global variable. You can also use the tracedata Debugger command to achieve this effect. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

These options overwrite any events and states in the current trace configuration. To view or modify the current trace configuration, use the **Set Triggers** window. See “The Set Triggers Window” on page 493 for more information.

Specifying a Trace Function Interval

MULTI provides very powerful tools to specify complex scenarios for collecting and displaying trace data. For simple tracing, however, you can use the **Trace Function Interval** option to quickly configure a trace scenario (or to view the configuration of a previously set trace function interval).

To define a trace function interval, right-click a function in the source pane and select **Trace → Trace Function Interval** from the shortcut menu. This opens a **Configure Trace Interval** window.



When tracing using the **Trace Function Interval** option, trace begins in the disabled state. Once the start condition is met, trace begins to be recorded. To set the start

condition, select the option that corresponds to the desired behavior from the **Start trace** drop-down box. To set the stop condition, select the desired behavior from the **Stop trace** drop-down box. The following table describes the options available from these drop-down boxes.

Start/Stop Trace Options	Behavior
Initially (ignores function)	Trace starts in the enabled state. The selected function is ignored.
Never (ignores function)	Trace is never disabled after it is enabled. The selected function is ignored.
On entry into	Trace starts/stops when the entry point of the selected function is executed. (This is the default setting for the left side of the window.)
On exit (epilogue) from	Trace starts/stops when the epilogue of the selected function is executed. (This is the default setting for the right side of the window.)
On any execution in	Trace starts/stops when any instruction of the specified function is executed.

Lists of all the functions in the program are displayed below the **Start trace** and **Stop trace** drop-down boxes. By default, the function that was highlighted when you selected **Trace Function Interval** will be the selected function, but you can click another function to change the selection.

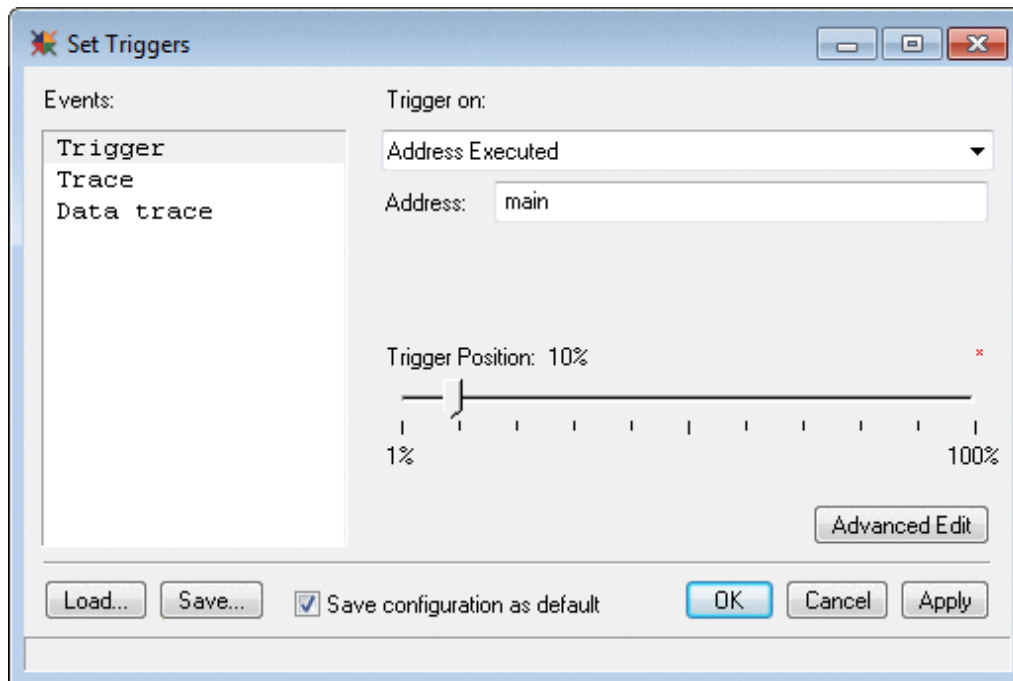
To confirm your settings, click **OK**. This will replace your existing trace configuration with the trace function interval configuration you selected. To view or modify the current trace configuration, use the **Set Triggers** window. See “The Set Triggers Window” on page 493 for more information.

The Set Triggers Window

The **Set Triggers** window provides control over the various trace events defined by each trace architecture. To open the **Set Triggers** window, do one of the following:

- In the Debugger, select **TimeMachine** → **Set Triggers**.
- In the Trace List or the PathAnalyzer, click the  button.
- In the Trace List or the PathAnalyzer, select **Config** → **Set Triggers**.

- In the MULTI Debugger command pane, enter the **trace triggers** command. For information about this command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.



The **Set Triggers** window allows you to modify triggers as well as other trace events.



Note

The **Set Triggers** window settings are saved on a per executable basis because they often reference function and variable names that are specific to the current project.

The events available in this window are target dependent. The most common events are described below.

Event	Description
Trigger	Stops trace collection when the data you are interested in has been collected. For more information, see “Configuring Trace Collection” on page 489.
Trace	Filters out unwanted trace data at the hardware level. Trace data is generated only while this event is active.

Event	Description
Data Trace	Filters out unwanted data trace. Memory accesses will only be traced while both this event and the Trace event are active.
External Output	Asserts an output signal of the trace hardware when active.

Some targets have very limited triggering hardware while others provide extensive triggering and filtering capabilities. Even if your target has extensive triggering hardware, there is always a limit to the number of each type of resource that can be used when defining events in the **Set Triggers** window. If you exceed this limit, the trigger will fail to be set and an error message will be displayed in the status area at the bottom of the window.

The **Set Triggers** window displays each event supported by your current trace architecture in the **Events** list. You can select an event to see the current setting for that event. In addition, you can modify the current value and click **Set** to apply the new setting to the selected event. When you click the **Set** button, the new values are transferred to the trace target in anticipation of the next trace run.

You can set events to occur on the following conditions.

Condition	Description
Always or Trigger Immediately	Always active. This condition is displayed as Trigger Immediately with trigger events. When Trigger Immediately is selected, the first instruction executed after trace is started will cause trace collection to trigger. This results in trace collection stopping automatically as soon as the trace buffer fills.
Never or Don't Trigger	Never active. This condition is displayed as Don't Trigger with trigger events.
Address Executed	Active when the instruction at the specified address is executing.
Function Executing	Active when any instruction in the specified function is executing.
Function Not Executing	Active when any instruction not in the specified function is executing.

Condition	Description
Function and Callees Executing	Active when a function and the functions it calls are executing. This condition is activated when the first instruction of the specified function executes, and it is deactivated when the last instruction of the same function executes. As a result of this behavior, this condition is not activated during a call to the specified function if the trace starts in the middle of the call. Functions with multiple return points are not supported. This condition is not available for all events.
Function and Callees Not Executing	Active when any instruction not in the specified function or in a function called by the specified function is executing. This condition is the inverse of Function and Callees Executing and is affected by the same limitations. This condition is not available for all events.
Function Interval Executing	Activates when the start condition is true and deactivates when the end condition is true. This condition is not available for all events. For more information about function interval tracing, see “Specifying a Trace Function Interval” on page 492.
Data Read/Written	Active when a variable or address is read from or written to.
Data Read	Active when a variable or address is read from.
Data Written	Active when a variable or address is written to.
Exception	Active when the specified exception is taken. This condition is only available on ARM ETM targets.
Custom Event	Active when a complex event is enabled. You can specify the complex event with the Advanced Edit button. For details, see “Editing Complex Events” on page 497.

For the function and address executing conditions, you can enter a function name, a function and line number, a function plus offset or an address in the **Address** field.

For the various data conditions, you can enter a global or static variable or an address in the **Address** field. You can also optionally enter a data value and the event will only occur when that value is read or written. If a data value is specified, the event will only activate if a single data access reads or writes the specified value. For example, these simple data conditions cannot be used to detect a write of a specific 64-bit value to a 64-bit variable if the 64-bit write is broken up into two 32-bit stores by the compiler.

Setting the Trigger Position

For targets that support a trigger, the trigger position slider appears when you edit the **Trigger** event. This slider allows you to set the desired trigger position as a percentage of the trace buffer being collected. The number displayed indicates the desired percentage of the buffer that comes before the trigger event in the final trace buffer. However, in the following circumstances, the selected trigger position has no impact on the trigger position in the final trace buffer:

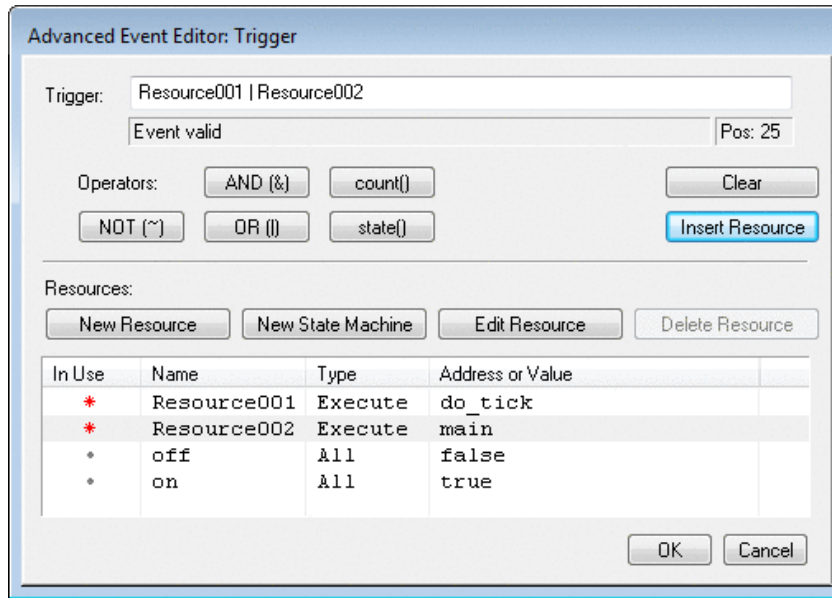
- If a trigger occurs shortly after trace collection is enabled, the actual location of the trigger may be before the desired trigger position.
- If a trigger occurs and then trace is manually or automatically retrieved before enough trace data is collected, the actual location of the trigger will be after the desired trigger position.

In all other cases, the trigger will be at the desired position in the trace buffer. For example, if you set the trigger position to 15 percent, 15 percent of the trace buffer will contain trace data from before the trigger, and 85 percent of the buffer will contain trace data from after the trigger.

Editing Complex Events

When you select the **Advanced Edit** button in the **Set Triggers**, **Trace Filters**, and **Trace Search** dialogs, you are presented with the **Advanced Event Editor** window. This window allows you to specify arbitrarily complex events built from simple events.

The **Advanced Event Editor** window allows you to specify an event by defining **Resources** and combining them in various ways. The top of the window displays the current event and allows you to modify it while the bottom pane displays the currently defined **Resources**.



The first step of defining an event is to create the **Resources** that you will use to build the event. **Resources** can be defined to activate when:

- The target processor executes from a specific address or within a range of addresses.
- The target processor reads or writes at a specific address or within a range of addresses.
- The target processor reads or writes a specific data pattern at a specific address or within a range of addresses.
- An external input to the target trace hardware is asserted (only available if supported by target hardware).
- An event such as a trace overflow, exception, or debug mode entry occurs. This type of **Resource** is not available through the **Set Triggers** window because it can only be used when analyzing trace data after it has already been collected.

In addition, you can create state machines that allow you to specify events that occur in a specific order. Finally, two **Resources**, **on** and **off**, are predefined; these events are always true and always false, respectively.

Creating Resources

To create a new resource, either click the **New Resource** button or right-click in the Resource List and select the **New Resource** menu option. This will open the **Resource Editor** dialog, which allows you to define the attributes for this resource.

The screenshot shows the 'Resource Editor: read_write_array' dialog box. It contains the following fields and options:

- Resource Name:** A text box containing 'read_write_array'.
- Resource Type:** A dropdown menu set to 'Read/Write'.
- Address:** A section with four radio buttons: 'Single', 'Range', 'Array' (which is selected), and 'Symbol'.
- Base:** A text box containing '0x01234567'.
- Size (in bytes):** A text box containing '45'.
- Bitmask:** A text box containing '0xffffffff'.
- Access Size:** A dropdown menu set to '1 byte'.
- Value:** A section with three radio buttons: 'Any Value' (which is selected), 'Equals', and 'Not Equal to'.
- Value:** An empty text box.
- Bitmask:** An empty text box.
- Buttons:** 'OK' and 'Cancel' buttons at the bottom right.

The available attributes are described in the table below.

Attribute	Description
Resource Name	The name of the resource. The name is used to represent the resource when defining events and must be composed of alphanumeric characters and underscores with the first character not a numeral.
Resource Type	Determines what must happen in order for this resource to activate.

Attribute	Description
Address	Defines the address or address range that the target must execute or access in order for this resource to activate. The radio buttons allow you to select between 4 different ways of specifying the Address attribute. • In Single mode, the resource will only activate when one specific address is matched. • In Range mode, both a start and end address are specified. • In Array mode, a start address and an offset are specified. The offset may be specified using the <code>sizeof</code> operator on a symbol name. For example, to specify an event that is true when any element of an array is accessed, you can set the Offset field to <code>sizeof(array_name)</code>. • In Symbol mode, the start and end addresses associated with the given symbol define the address range which is used. For those modes in which a range or offset is used (the Range, Array, and Symbol modes), the end address is exclusive. In both Single and Range modes, the <code>exit</code> operator can be used on a function name to give the address of the last instruction in the function. For example, to create a resource that evaluates to true when a function called <code>increment</code> returns, set the address field to <code>exit(increment)</code>. The Bitmask is ANDed with the accessed address before any comparisons are made. This required attribute applies to Read/Write, Read, Write, and Execute resources. The Access Size specifies a memory access size that must be matched by a read or write in order for this resource to activate. This optional attribute applies to Read/Write, Read, and Write resources and is only available with some targets.
Value	Allows a data pattern to be specified that must be matched by the read or write in order for this resource to activate. The Bitmask is ANDed with the data value transferred by the target before it is compared with the specified data value. The resource only activates if a single data access reads or writes the specified value. For example, a single Write resource cannot be used to detect a write of a specific 64-bit value to a 64-bit variable if the 64-bit write is broken up into two 32-bit stores by the compiler. This optional attribute applies to Read/Write, Read, and Write resources.
Index	Specifies which external input signal to your target's triggering hardware will activate this Resource. This required attribute only applies to External resources.

Attribute	Description
Event Type	Specifies which type of event will activate this Resource . The available types are described in the next section. This required attribute only applies to Event resources.
Exception	Specifies which type of exception will activate this Resource . The available types are described in “Exception Type Descriptions” on page 502. This required attribute only applies to Exception resources, which are only available on ARM ETM targets.
MMD Index	Specifies which memory map decode will activate this Resource . For more information, see the documentation for your processor. This required attribute only applies to Memory Map Decode resources, which are only available on ARM ETM targets.

Event Type Descriptions

Trigger	The trace trigger point.
Trace Disabled	One or more instructions that were not traced as a result of filtering configured with the Set Triggers window.
Overflow	A trace port overflow that caused a gap of one or more instructions in the trace.
Debug Mode	An entry into debug mode caused by the target processor hitting a breakpoint or being halted by the Debugger.
Exception	On most targets: any exception. On ARM targets: one of the exceptions described in the next section.
Error	A trace processing error.
Reconstructed	An instruction that was lost as a result of trace port overflows but was reconstructed by automatic gap reconstruction. For more information, see the option Attempt to reconstruct gaps in trace data in “The Trace Options Window” on page 480.

Exception Type Descriptions

Exception resources are implemented by an Address Execute resource on the exception vector address. Therefore MULTI provides two versions of each of the following exception options. If your application uses high exception vectors (starting at `0xffff0000`), you should choose the **(High)** version.

The following exceptions are only available for triggering on ARM.

Reset	The traced processor was reset.
Undefined Instruction	The traced processor attempted to execute an invalid instruction.
Software Interrupt	The traced processor executed a software interrupt instruction.
Prefetch Abort	The traced processor attempted to execute an instruction for which a memory abort was signaled while the instruction was being fetched.
Data Abort	The traced processor failed to read or write memory due to a memory abort.
IRQ (Interrupt request exception)	The traced processor handled an interrupt request.
FIQ (Fast interrupt request exception)	The traced processor handled a fast interrupt request.

Creating State Machine Resources

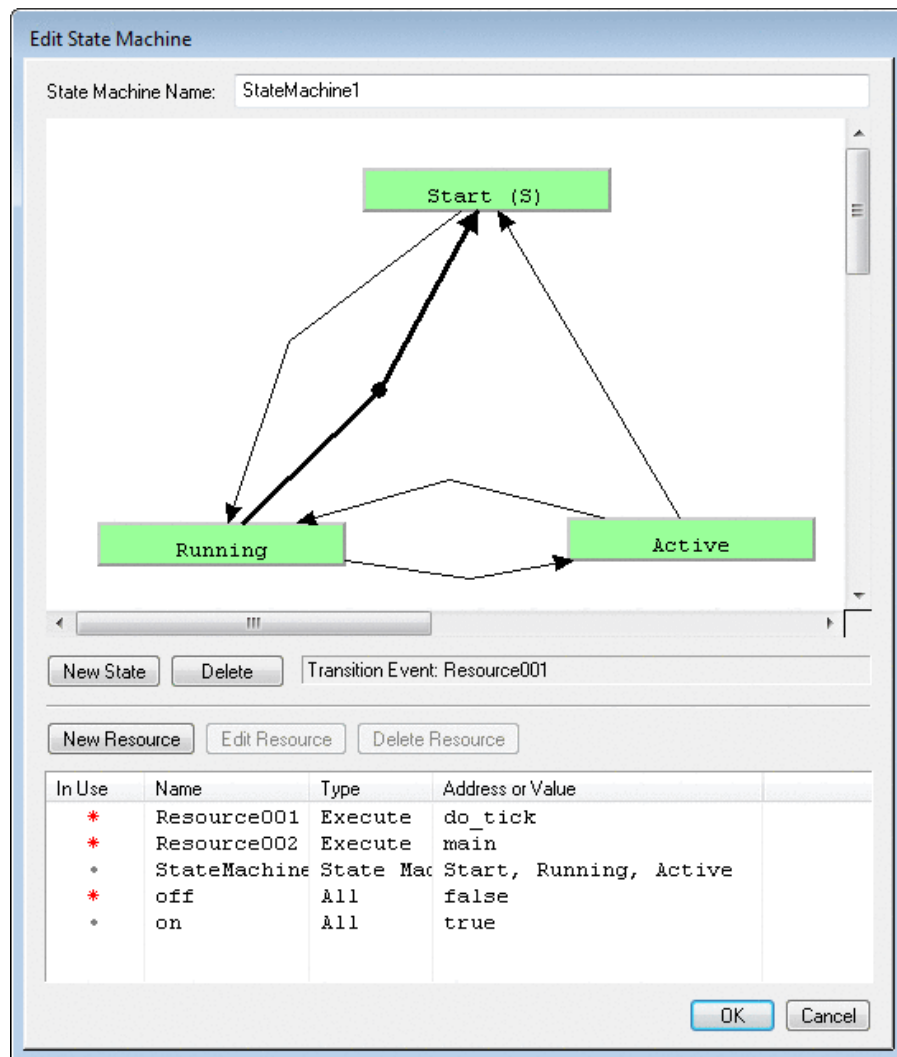
State machine resources allow you to specify events that are active after certain events have occurred and optionally before some others have happened. For example, this allows you to create complex events that are true after a function has been called but before it has returned. This type of event would be true as long as the specified function is present in the call stack.



Note

Support for state machines and the number of supported states depends on your trace architecture. For more information, see the specification for your trace architecture.

To add a state machine, click the **New State Machine** button. This opens the state machine editor, which allows you to create a state machine by adding states and defining state transition events.



There are two ways to add a state to a state machine:

- Click the **New State** button. Then click the state and move it to a location on the canvas.
- Right-click the background canvas and select **Add State**. Then click the state and move it to a location on the canvas.

To rename a state, double-click it and enter a new name in the **Set State Name** dialog.

To delete a state, select it and click the **Delete** button or press the **Delete** key.

A *state transition* defines the events on which the state machine changes state. Once you have added all of the states you want, you can create transitions by right-clicking the source state and selecting **Add Link**. Then click the destination state to create the link. Alternatively, you can right-click and drag to the destination state to define a new state transition.

Once you have defined a transition's start and destination states, the **Advanced Event Editor** window will open so that you can define an event on which this transition will occur. The transition will occur when the event evaluates to true.

To edit the event of an existing link, either double-click it or right-click and select **Edit**.

The state machine starts in the start state, which is denoted by (S) after the state name. To mark a state as the start state, right-click it and select the **Set as Start State** menu option.

Defining Complex Events

After creating the resources and state machines that you need for your event, construct the event by inserting the resources into the event field. This field contains a string that consists of a Boolean combination of resources. In addition, you can specify the count of a resource and the state when a state machine resource is active.

To clear your event, you can click the **Clear** button. When you clear the event field, the **Event Progress Indicator** will display “Event not complete,” which indicates that you must enter more text to define a valid event.

To insert a new resource, select it in the **Resource List** and either click the **Insert Resource** button or right-click and select **Insert Resource**. The selected resource name will appear in the event field.

To create a Boolean combination of two resources, insert a resource and then click one of the Boolean operator buttons (**AND** or **OR**). The corresponding Boolean

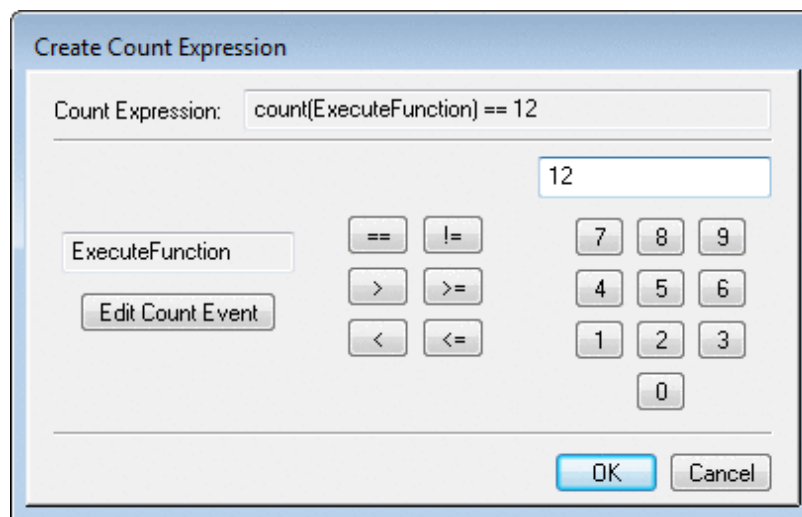
operation will appear in the event field. You can now insert another resource to combine with the first one.

You can specify the precedence of multiple Boolean combinations by using parentheses. Simply type them in the event field around the combinations that you want to have the highest priority and they will be evaluated first. When parentheses are not properly matched, the Event Progress Indicator will display “Unmatched parentheses,” which means that you do not have valid matching parentheses.

Count Expression

Sometimes you may want an event to be active when an event has occurred a certain number of times. For example, if you have a bug that occurs after a function has been executed 10,000 times, then you may want to trigger after the function has been called 10,000 times. Count expressions allow you to define events of this form.

Any time that a resource can appear in the event field, you can also specify the count of a resource. To insert a resource with a count, click the **count()** button. This will open the **Count** dialog, shown below.



This dialog allows you to specify the event to count as well as a comparison operator and a number. A count expression is true whenever the number of times the **Count Event** has occurred during trace collection is such that comparing the count to the specified number with the specified comparison operator evaluates to true.

To set the **Count Event**, click the **Edit Count Event** button. This opens another **Advanced Event Editor** window that allows you to specify the event to count. It is impossible to create nested count statements, so the **count()** button is disabled in this new window. Once you have selected the **Count Event**, use the six operator buttons to select the **Count Operation**. Finally, either type or use the keypad to enter a number in the count field. The expression that will be entered into your event is displayed at the top of this window so that you can tell exactly what will be inserted. To accept this string, click the **OK** button and the string will be inserted into the event field.

To enter a count expression manually, the syntax is `count(event) op value` where *event* is the event to count, *op* is the count operator (one of `==`, `!=`, `>`, `>=`, `<`, or `<=`) and *value* is the number to compare against. You can type an expression of this form directly into the event field in the **Advanced Event Editor** window.



Note

When using hardware counters to control trace collection, the counter may count each cycle, rather than each instruction, when the count expression evaluates to true. This may inflate the count. Please refer to the manual for your trace hardware.

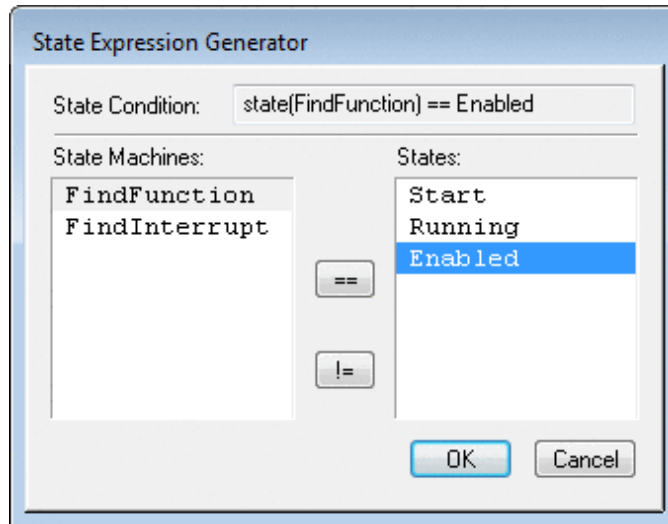


Note

For ETM trace, if you are counting accesses to or execution of a single address with a count, the count is exact. It counts the number of times the address was accessed or executed. If you are counting accesses to or execution within an address range or a symbol, the count is sticky. It counts the number of cycles between access or execution of an address in the range and access or execution of an address outside the range. For practical purposes, this means that using a count with an address range is unlikely to yield the desired results.

State Machine Expressions

You can also insert State Machine resources into the event field any time that another resource can be used. To insert a State Machine resource, click the **state()** button. The **State Expression Generator** dialog will appear. This dialog allows you to select the state that is the active state in a state machine. A State Machine resource evaluates to true when it is in the active state defined by a state expression.



To create a state expression, select the State Machine to insert. Then select whether you want the event to be active when you are in a given state or when you are not in a given state by selecting == or !=. Finally, select the desired state from the **States** list. The **State Expression Generator** displays the string that will be inserted into the event field in the **State Condition** field.

To enter a State Expression manually, the syntax is `state(<machine>) <op> <state>` where <machine> is the name of the State Machine resource, <op> is the operator (either == or !=) and <state> is the name of the active state.

Part V

Advanced Debugging in Specific Environments

Chapter 21

Testing Target Memory

Contents

Quick Memory Testing: Using the Memory Test Wizard	512
Advanced Memory Testing: Using the Perform Memory Test Window	515
Viewing Memory Test Results	523
Continuous Memory Testing	524
Memory Testing with a Target Agent	524
Types of Memory Tests	525
Efficient Testing Methods	535
Running Memory Tests from the Command Line	535
Detecting Coherency Errors	536

MULTI allows you to perform a number of destructive and nondestructive tests on your target's memory. Destructive tests overwrite the contents of memory in the test region. Two graphical tools, the **Memory Test Wizard** and the **Perform Memory Test** window, provide a simple way to configure and perform memory tests.

Additionally, MULTI offers manual and optional automatic methods for checking coherency between target memory and an original executable program file. These tools and available types of memory testing are described in this chapter.



Note

You can perform all memory tests by issuing the **memtest** command in the Debugger command pane. However, the **Perform Memory Test** window provides a simpler way to control all of the same behaviors and parameters that you can configure with the more complicated **memtest** command. For the complete syntax of the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

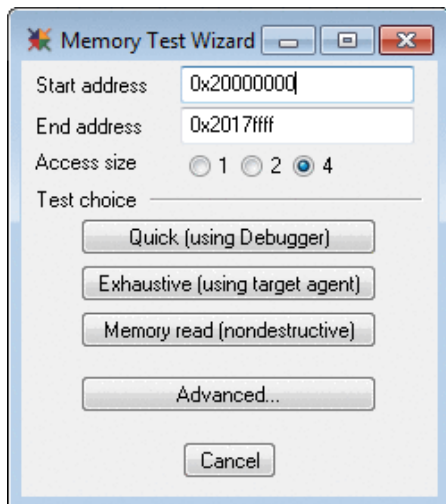
We recommend that you run your target's setup script before testing memory. Memory testing generally requires that your target be in a good state; for example, your target's memory controller should be initialized. If accessing memory via your target connection (for example, Green Hills Probe) does not work, memory testing via the Debugger will not work. In order to test memory using a target agent, you must also be able to download a program (the target agent) onto your target and run it.

Quick Memory Testing: Using the Memory Test Wizard

The **Memory Test Wizard** allows you to configure and launch the most common memory tests quickly. If you want to set options or run tests other than those listed on the **Memory Test Wizard**, click **Advanced** to open the **Perform Memory Test** window. For more information, see “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

To launch the **Memory Test Wizard**, connect to your target, and select **Target** → **Memory Test** from the Debugger.

A sample **Memory Test Wizard** is shown next.



This window allows you to set the basic parameters for your memory test (i.e., the start address and end address of the area of memory to be tested, and the access size to be used while performing the test) and quickly launch one of three common test combinations.

To configure your memory test from the **Memory Test Wizard**, first use the fields described next to define the memory area and access size for the test.

Start address	Defines the lowest address to test. Enter a valid address in this field. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.) This field defaults to 0x00000000.
----------------------	--

End address	Defines the highest address to test. Enter a valid address in this field. This address must be greater than the start address. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.) This field defaults to 0x00000000. The end address should be equal to: $start_address + size - 1$
Access size	Specifies the access size, in bytes, to be used while performing the memory test. Select 1, 2, or 4. The default is 4.

**Note**

Valid values that you enter in the **Start address**, **End address**, and **Access size** fields are carried over to the **Perform Memory Test** window if you select **Advanced**.

The **Test choice** section of the **Memory Test Wizard** allows you to select one of the tests or test combinations listed in the following table.

Quick (using Debugger)	Performs the address bus walking test and data bus walking test, with both walking ones and zeros. When launched with this button, these tests are performed by the Debugger. See “Address Bus Walking Test” on page 525 and “Data Bus Walking Test” on page 527 for a full description of these tests.
-------------------------------	---

Exhaustive (using target agent)	Performs the address bus walking test and data bus walking test, with both walking ones and zeros, as well as the data pattern test. The pattern test uses the pseudorandom pattern and the maximize address bus transition options (see “Data Pattern Test” on page 529 for details). When launched with this button, these tests are performed using a target agent. The target agent is not available for all targets. If no target agent is available for your target environment, this button will be dimmed. For more information about target agents, see “Memory Testing with a Target Agent” on page 524. This test uses the target agent positioned at the start of the test range. See “Address Bus Walking Test” on page 525, “Data Bus Walking Test” on page 527, and “Data Pattern Test” on page 529 for a full description of these tests.
Memory read (nondestructive)	Performs the memory read test. When launched with this button, this test is performed by the Debugger. See “Memory Read Test” on page 532 for a full description of this test.

Clicking one of the preceding buttons launches the corresponding memory test(s) on the target that was selected in the target list when you opened the **Memory Test Wizard**. A **Memory Test Results** window opens (see “Viewing Memory Test Results” on page 523) and remains open after the test completes or is aborted.



Note

To abort memory testing, press the **Esc** key.

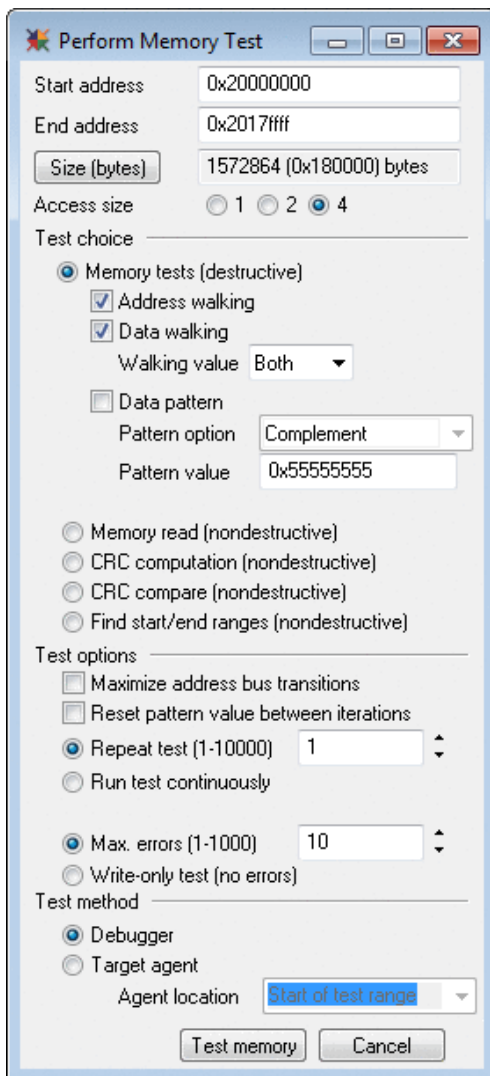
Advanced Memory Testing: Using the Perform Memory Test Window

The **Perform Memory Test** window provides more options and control for configuring memory tests than the **Memory Test Wizard**.

To open the **Perform Memory Test** window:

1. Connect to your target.
2. Select **Target** → **Memory Test** from the Debugger window.
3. Click **Advanced** in the **Memory Test Wizard** that appears.

A sample **Perform Memory Test** window is pictured below. Valid values specified in the **Memory Test Wizard** are carried over to the **Perform Memory Test** window.



This window allows you to specify the area of memory to be tested, the type of memory tests to be performed, and various options about how those tests are performed. The following sections explain how to configure all of these settings.

Defining Memory Areas to Test

The fields at the top section of the **Perform Memory Test** allow you to define the memory area and access size for your memory test.

Start address	0x20000000
End address	0x2017ffff
Size (bytes)	1572864 (0x180000) bytes
Access size	☐ 1 ☐ 2 ☒ 4

These fields are described in the following table.



Note

Three of these fields are identical to fields in the **Memory Test Wizard**. Any valid values you may have input in these fields in the wizard are carried over to the **Perform Memory Test** window.

Start address	Defines the lowest address to test. Enter a valid address in this field. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.) This field defaults to 0x00000000, or to the value entered in the Memory Test Wizard, if any.
End address	Defines the highest address to test. Enter a valid address in this field. This address must be greater than the start address. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.) This field defaults to 0x00000000, or to the value entered in the Memory Test Wizard, if any. The end address should be equal to: $start_address + size - 1$

Size (bytes)	Displays the total size of the area of memory to be tested. This field is provided for your information; the memory test itself uses the specified Start address and End address values. Click the Size (bytes) button if you want to calculate or recalculate the size after changing the Start address or End address.
Access size	Specifies the access size, in bytes, to be used while performing the memory test. Select 1, 2, or 4. This setting defaults to 4, or to the choice entered in the Memory Test Wizard, if any.

Selecting Memory Tests

The second section of the **Perform Memory Test** window allows you to select which memory test(s) to perform.

Test choice

☒ Memory tests (destructive)

☒ Address walking

☒ Data walking

Walking value: Both

☐ Data pattern

Pattern option: Complement

Pattern value: 0x55555555

☐ Memory read (nondestructive)

☐ CRC computation (nondestructive)

☐ CRC compare (nondestructive)

☐ Find start/end ranges (nondestructive)

The test choices include both *destructive* and *nondestructive* tests. Destructive tests overwrite the contents of memory in the test region. You can run more than one type of destructive test simultaneously, but nondestructive tests must be run individually. The following table briefly describes the test choices. Each test is described in more detail in “Types of Memory Tests” on page 525.

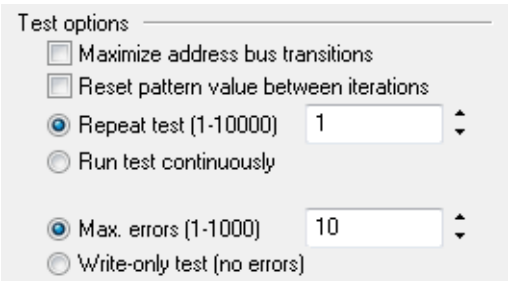
Address walking	Selects the address bus walking test. This destructive memory test verifies that address lines are not stuck at one or zero and are not tied together. For more details about this test, see “Address Bus Walking Test” on page 525. This test can be run at the same time as other destructive memory tests.
------------------------	--

Data walking	Selects the data bus walking test. This destructive memory test verifies that data bus lines are not stuck at one or zero and are not tied together. For more details about this test, see “Data Bus Walking Test” on page 527. This test can be run at the same time as other destructive memory tests.
Walking value	Determines the walking value for address walking and/or data walking tests. Make a selection to specify whether these tests should be performed with a walking value that is a single one bit surrounded by zero bits (One) or a single zero bit surrounded by one bits (Zero), or if walking tests should be performed twice, once with the walking value <i>one</i> and once with the walking value <i>zero</i> (Both). The default setting is Both. This option is only available if you have selected the address walking and/or data walking test(s).
Data pattern	Selects the data pattern test. This destructive memory test verifies that all memory addresses in the range can successfully store values. For more details about this test, see “Data Pattern Test” on page 529. This test can be run at the same time as other destructive memory tests.
Pattern option	Determines what type of modifications will be performed on the data pattern between iterations when performing a data pattern test. Select Static, Complement, Rotate, Rotate and Complement, or Pseudorandom. The default is Complement. For more detail about these options, see “Data Pattern Test” on page 529. This option is only available if you have selected the data pattern test.
Pattern value	Specifies the pattern to use when performing a data pattern test. The default value is 0x55555555, which is a sequence of alternating binary zeros and ones. This option is only available if you have selected the data pattern test.
Memory read (nondestructive)	Selects the memory read test, which checks that each memory address obtains the same value when the address is read twice. For more details about this test, see “Memory Read Test” on page 532. This test cannot be run at the same time as any other memory tests.
CRC computation (nondestructive)	Selects the CRC computation test, which computes a CRC checksum on a range of memory. For more details about this test, see “CRC Compute” on page 532. This test cannot be run at the same time as any other memory tests.

CRC compare (nondestructive)	Selects the CRC compare test, which repeatedly computes CRC checksums for a range of memory, in order to verify that memory can be read reliably. For more details about this test, see “CRC Compare” on page 533. This test cannot be run at the same time as any other memory tests.
Find start/end ranges (nondestructive)	Selects the find start/end ranges test, which locates possibly unused memory at the start and end of the range being tested. For more details about this test, see “Find Start/End Ranges” on page 533. This test cannot be run at the same time as any other memory tests.

Setting Test Options

The third section of the **Perform Memory Test** window allows you to choose options that will apply to the tests you have selected. (Options that do not apply to the tests you have selected are dimmed.)



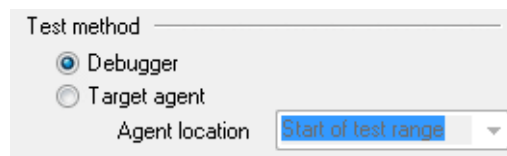
The available options and their effects are listed in the following table. Not all options are available with all tests.

Maximize address bus transitions	Uses a sequence of addresses in the data pattern test or memory read test that maximizes the address line transitions between accesses. If this option is not selected, the default behavior accesses memory sequentially from low addresses to high addresses. When this option is selected, the sequence of memory accesses would begin <i>start</i>, <i>end</i>, <i>start</i>+1, <i>end</i>-1, and so on; assuming that <i>start</i> and <i>end</i> define a power of two sized and aligned region and the access size is one byte. If <i>start</i> and <i>end</i> do not define a power of two sized and aligned range, the range is split into sequential power of two sized and aligned ranges for the purposes of this test.
---	--

Reset pattern value between iterations	Executes the selected tests on every iteration rather than attempting to use different starting pattern values on successive test iterations. This option is only valid if the Repeat test value is greater than one or if the Run test continuously option is used. For pattern tests, this option has no effect if the Pattern option is Static .
Repeat test	Repeats memory test(s) the indicated number of times. If unspecified, the test(s) will be performed only once.
Run test continuously	Repeats memory test(s) continuously.
Max. errors	Aborts memory test(s) after detecting the specified number of errors. This value must be between 1 and 1000. If unspecified, MULTI will abort the test after 10 errors.
Write-only test (no errors)	Skips the reading phase of the address bus walking, data bus walking, and/or data pattern tests. Since no reading is performed, no errors can be reported.

Specifying Test Methods

This last section of the **Perform Memory Test** window allows you to choose whether tests will be performed by the Debugger reading and writing memory directly, or by MULTI downloading a small target agent program that will perform the memory accesses. (See “Memory Testing with a Target Agent” on page 524 for more details about using a target agent.)



The options available in this section and their effects are described in the table below.

Debugger	Specifies that the Debugger should perform the test or tests directly.
Target agent	Specifies that a target agent should be used to perform the test or tests. For more information, see “Memory Testing with a Target Agent” on page 524.

Agent location	Specifies where the target agent is downloaded. Select Start of test range or End of test range , or specify another location by entering it in the field, replacing the <specify location> choice. The default is to download the target agent at the start of the address range being tested. For more details, see “Memory Testing with a Target Agent” on page 524.
-----------------------	---

Running Memory Tests

After you have specified the memory range to be tested, the access size, the test(s) to perform, your test options, and your desired test method in the **Perform Memory Test** window, as described in the previous sections, click the **Test memory** button to run the selected test(s). The test(s) run on the target that was selected in the target list when you opened the **Memory Test Wizard**.

A **Memory Test Results** window is displayed and shows the test progress and results. For more information, see “Viewing Memory Test Results” on page 523. This window remains open after the test completes or is aborted.

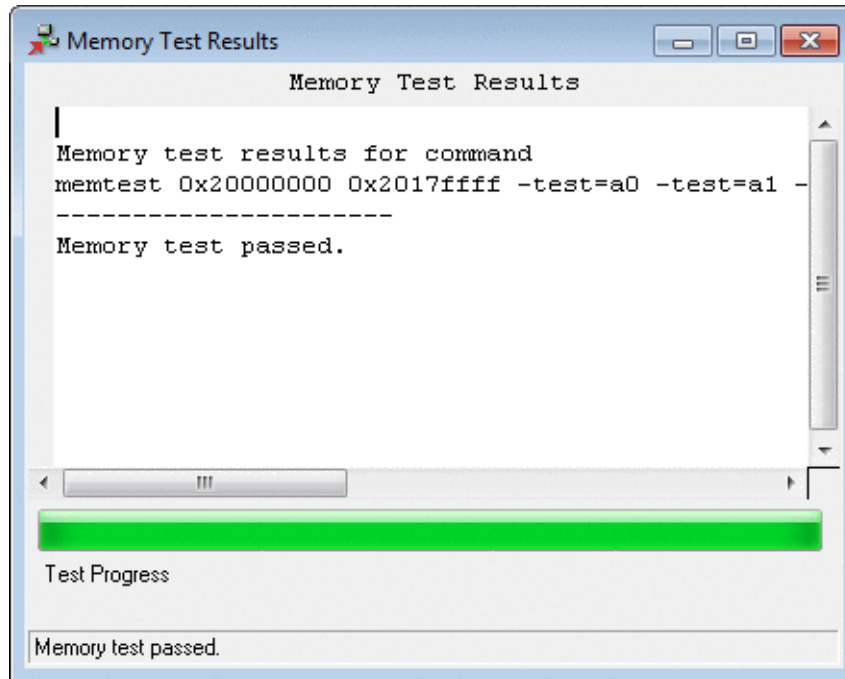


Note

To abort memory testing, press the **Esc** key.

Viewing Memory Test Results

After you launch one or more memory tests using the **Memory Test Wizard** or the **Perform Memory Test** window, a **Memory Test Results** window will appear.



The **Memory Test Results** window will contain the following information:

- A **memtest** command that is equivalent to the various tests and options selected in the **Memory Test Wizard** or **Perform Memory Test** window. (This is provided to make it easier to write MULTI Debugger scripts that can automatically perform memory tests.) For information about the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.
- The printed output of the command, including any memory test errors. (This appears below the line of dashes.)
- A test progress indicator.
- A status bar, which displays messages about the success or failure of the test(s).

Continuous Memory Testing

Continuous memory testing can be helpful when you are debugging hardware problems with a logic analyzer or other diagnostic equipment. To perform continuous testing, select the **Run test continuously** option in the **Perform Memory Test** window, or pass the **-continuous** option to the **memtest** command. To terminate the test after a specific number of errors, enter the number of errors in the **Max. errors** field, or pass the **-maxerr=number_of_errors** option to the **memtest** command. For information about the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

To terminate a continuous test manually, press **Esc**.

Memory Testing with a Target Agent

You can perform memory tests using either the Debugger or a small target agent program. Running memory tests from the Debugger is slower than running the tests using a target agent, but a target agent requires that the memory be initialized correctly. To specify which method you want MULTI to use, select either **Debugger** or **Target agent** in the bottom section of the **Perform Memory Test** window (see “Specifying Test Methods” on page 521).

When using a target agent, you must specify where MULTI should load the target agent code. To do this, use the **Agent location** field of the **Perform Memory Test** window to select one of the following locations for target agent placement:

- Start of test range (This is the default location)
- End of test range
- Specified location outside the test range



Note

Keep the following restrictions and conditions in mind when specifying the location of your target agent:

- If you select the start or end of the test range, the test range must be at least twice as large as the target agent. This allows the complete range to be tested, by first placing the target agent at the start of the

range, and then placing the target agent at the end of the range. (The target agent requires an estimated 10–20 KB of memory for its code and data. However this varies by target CPU architecture and is subject to change.)

- The target agent should not be located at the start or end of the test range for nondestructive tests, or else the target agent will overwrite that memory.
- If you specify a target agent location other than the start or end of the test range, it must not overlap the test range.

If you are running memory tests using the **memtest** command, use the **-tgtagent** option to specify that memory testing be performed with a target agent. To specify the location of the target agent code, use one of the following options:

- **-tgtagentstart** — Place target agent at the start of the test range.
- **-tgtagentend** — Place target agent at the end of the test range.
- **-tgtagentloc=expr** — Place target agent at the location specified by the expression *expr*.

For more information about the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Types of Memory Tests

The sections below describe the different types of memory testing in detail. For information about launching these tests, see “Quick Memory Testing: Using the Memory Test Wizard” on page 512, “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515, or “Running Memory Tests from the Command Line” on page 535.

Address Bus Walking Test

The address bus walking test is a destructive memory test that provides an efficient way to verify that address lines are not stuck at one or zero and are not tied together. This test is usually fast enough to be run from the Debugger, so it generally is not

necessary to use a target agent (see “Specifying Test Methods” on page 521 for information about these two test methods).

In this test, the range of memory is divided into sequential regions that are sized and aligned to a power of two. For example, if memory from 0xa00 to 0xffffffff is to be tested, the actual ranges that will be tested are as follows:

0xa00 to 0xbff (512 B)	0b1010 0000 0000 0b1011 1111 1111
0xc00 to 0xfff (1KB)	0b1100 0000 0000 0b1111 1111 1111
0x1000 to 0x1fff (2KB)	0b0001 0000 0000 0000 0b0001 1111 1111 1111

and so on, up to:

0x800000 to 0xffffffff (8MB)	0b1000 0000 0000 0000 0000 0000 0b1111 1111 1111 1111 1111 1111
---------------------------------	--

This test can be performed as a *walking one test* or a *walking zero test*, or you can set it to run twice, testing once for ones and once for zeros. Depending on the walking value selected (one or zero), a pattern value is written to an address in the range with a single address bit set to 1 (walking one test) or to 0 (walking zero test). All other address bits that reference that region are set to the opposite value. The low bits are cleared, as appropriate for the access size used. The pattern's complement is written to locations in the test range with a different address bit set (walking one test) or cleared (walking zero test). The pattern's complement is also written to the location at the start of the range (walking one test) or the end of the range (walking zero test). After the complement values are written, the original pattern value that was written is verified to make sure it has not changed. This is repeated for every address bit that references the range of memory to be tested.

The default pattern is 0x55555555, 0x5555, or 0x55, depending on the access size.

For example, using the range 0x0000 to 0xffff, a walking one test with an access size of 4 produces a sequence that begins as follows.

```
Write 0x55555555 to address 0x0004
Write 0xaaaaaaaa to address 0x0008
Write 0xaaaaaaaa to address 0x0010
...
Write 0xaaaaaaaa to address 0x8000
```

```
Write 0xaaaaaaaa to address 0x0000
Read from address 0x0004 and verify that the value is 0x55555555.

Write 0x55555555 to address 0x0008
Write 0xaaaaaaaa to address 0x0004
Write 0xaaaaaaaa to address 0x0010
...
Read from address 0x0008 and verify that the value is 0x55555555.
...
Write 0x55555555 to address 0x0000
Write 0xaaaaaaaa to address 0x0004
Write 0xaaaaaaaa to address 0x0008
...
Read from address 0x0000 and verify that the value is 0x55555555.
```

To run this test from the **Perform Memory Test** window, select the **Address walking** radio button, then select a **Walking value (One, Zero, or Both)**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

This test is also run twice, once with walking ones and once with walking zeros, if you select the **Quick** or **Exhaustive** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 512.

To run this test using the **memtest** command, use the **-test=a0** option (for a walking zero test) and/or the **-test=a1** options (for a walking one test). See the description of the **memtest** command in “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Data Bus Walking Test

The data bus walking test is a destructive memory test that provides a way to verify that data bus lines are not stuck at one or zero and are not tied together. This test is usually fast enough to be run from the Debugger, so it generally is not necessary to use a target agent (see “Specifying Test Methods” on page 521 for more information about these different approaches).

This test can be performed as a *walking one test* or a *walking zero test*, or you can set it to run twice, testing once for ones and once for zeros. Depending on the walking value selected (one or zero), successive values containing either a single one bit (walking one test) or a single zero bit (walking zero test) are written to successive memory locations and then read back. Only the first 8, 32, or 128 bytes of memory in the range are modified by this test, depending on the access size specified.

For example, using the range 0x0000 to 0xffff, a data walking one test with an access size of 4 produces a sequence that begins as follows:

```
Write 0x00000001 to address 0x0000
Write 0x00000002 to address 0x0004
Write 0x00000004 to address 0x0008
...
Write 0x80000000 to address 0x007c
Read from address 0x0000 and verify that the data value
    is 0x00000001
Read from all other addresses written and verify that the
    value is as expected.
```

You can specify a repeat count, which will rotate the initial data pattern at the start of each test. Using the same options utilized in the previous example, a repeat count will produce a sequence that begins with the following:

```
Iteration 1:
Write 0x00000001 to address 0x0000
...
Iteration 2:
Write 0x00000002 to address 0x0000
...
Iteration 3:
Write 0x00000004 to address 0x0000
...
```

If the memory range is not large enough to write all distinct values, the test will write some of the walking data values, read them back to verify that the value is what is expected, then start from the beginning of the memory range and write more of the walking data values.

To run this test from the **Perform Memory Test** window, select the **Data walking** radio button, then select a **Walking value (One, Zero, or Both)**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

This test is also run twice, once with walking ones and once with walking zeros, if you select the **Quick** or **Exhaustive** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 512.

To run this test using the **memtest** command, use the **-test=d0** option (for a walking zero test) and/or the **-test=d1** options (for a walking one test). See the description of the **memtest** command in “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Data Pattern Test

The data pattern test is a destructive memory test that provides a way to comprehensively test that all memory addresses in the range can successfully store values. This test can help diagnose ground bounce and voltage problems in the memory range.

The data pattern test writes a specified pattern to successive memory locations. The same pattern value can be written to all locations, or you can specify that certain modifications be made between iterations. The options for how the pattern is handled are listed in the table below.

Pattern option	Effect
Static	Writes the same value to all locations.
Complement	The specified pattern is complemented after each write. You can use this option to generate a large number of address and data bus transitions on successive memory accesses. For example, if the range 0x0000 to 0xffff is tested with the pattern 0x01234567, an access size of 4, and the pattern is complemented, the test sequence for the pattern test would be: <pre>Write 0x01234567 to address 0x0000 Write 0xfedcba98 to address 0x0004 Write 0x01234567 to address 0x0008 ... Read from all addresses written and verify that the value is as written.</pre>
Rotate	The specified pattern is rotated each time it is written. The previous pattern value is shifted one bit position to the left and the most significant bit of the previous pattern value is copied to the least significant bit position of the next pattern value.

Pattern option	Effect
Rotate and Complement	The specified pattern is rotated after its value and its complement are written. For example, if the range 0x0000 to 0xffff is tested with the pattern 0x01234567, an access size of 4, and the pattern is rotated and complemented, the test sequence for the pattern test would be: <pre>Write 0x01234567 to address 0x0000 Write 0xfedcba98 to address 0x0004 Write 0x02468ace to address 0x0008 Write 0xfdb97531 to address 0x000c ... Read from all addresses written and verify that the value is the same as was written.</pre>
Pseudorandom	The specified value is fed through a linear feedback shift register (LFSR) that generates a sequence of $2^n - 1$ values. If this method is selected, the next value in a sequence is generated as follows (bit positions are indicated with bit 0 as the least significant bit): 1. Compute a new bit 0 value by performing the exclusive-or of four bits in the current value. Bit positions in this description are numbered starting with the least significant bit. 2. Shift the current value left by one. 3. Insert the computed bit 0 value in the new value. The four bit positions vary according to access size, as follows: • If the access size is 8, the bit positions are 3, 4, 5, and 7. • If the access size is 16, the bit positions are 3, 12, 14, and 15. • If the access size is 32, the bit positions are 0, 1, 21, and 31. The data pattern consisting of all zeros is not permitted, since the LFSR will always generate a zero in that case.



Note

You can use the **Maximize address bus transitions** option in the **Perform Memory Test** window or the **-maxtransitions** option with the **memtest** command to cause MULTI to use a sequence of addresses for this test that maximizes the address line transitions between accesses. If

this option is not specified, the default behavior is to access memory sequentially from low to high addresses. When this option is selected, the sequence of memory accesses would begin `start`, `end`, `start+1`, `end-1`, and so on; assuming that `start` and `end` define a power of two sized and aligned region and the access size is one byte. If `start` and `end` do not define a power of two sized and aligned range, the range is split into sequential power of two sized and aligned ranges for the purposes of this test. For information about the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

To run this test from the **Perform Memory Test** window, select the **Data pattern** radio button, then select a **Pattern option** (**Static**, **Complement**, **Rotate**, **Rotate and Complement**, or **Pseudorandom**) and specify a **Pattern value**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

This test is also run, using a target agent, the pseudorandom pattern option, and the **Maximize address bus transitions** option, if you select the **Exhaustive** button on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 512.

To run this test using the **memtest** command, use the **-test=p** option and specify the pattern with the **-pattern=value** option. To specify the pattern behavior, use the following options:

- **-complement** — To complement the data pattern value between memory writes.
- **-rotate** — To rotate the data pattern for the pattern test between writes.
- **-random** — To use a pseudorandom sequence of values for the pattern test.

You can pass the **-complement** and **-rotate** options together. See “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Memory Read Test

The memory read test is a nondestructive memory test that verifies that each memory address retains the same value when the address is read twice.

This test reads memory from successive memory locations. After reading each address for the first time, the test reads back the value from the prior address and verifies that the values are identical.

This test does not destroy the memory in the range. For example, to test the range 0x0000 to 0xffff with an access size of 4, the test sequence begins:

```
Read from address 0x0000
Read from address 0x0004
Check: Read from address 0x0000 and verify that the value
       matches the earlier read.
Read from address 0x0008
Check: Read from address 0x0004 and verify that the value
       matches the earlier read.
...
```

To run this test from the **Perform Memory Test** window, select the **Memory read (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

This test is also run if you select the **Memory read (nondestructive)** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 512.

To run this test using the **memtest** command, use the **-test=r** option. See “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

CRC Compute

CRC compute provides a way to compute a CRC checksum on a range of memory. This nondestructive test reads bytes from successive memory locations and computes a CRC checksum from the result. The algorithm used is a standard 32-bit CRC and matches the algorithm used by default in the Green Hills Software linker's **-checksum** option.

To run this test from the **Perform Memory Test** window, select the **CRC computation (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

To run this test using the **memtest** command, use the **-test=cr** option. See “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

CRC Compare

CRC compare provides a way to repeatedly compute a CRC checksum for a range of memory, in order to verify that memory can be read reliably. This nondestructive test reads bytes from successive memory locations and computes a CRC checksum from the result. It then recomputes the CRC checksum and verifies that the recomputed checksum matches the original computed value. You can specify how many times the test should be repeated. If the checksum does not match, an error is printed and the most recent checksum value is used from that point on for any further verify iterations.

The algorithm used is a standard 32-bit CRC and matches the algorithm used by default in the Green Hills Software linker's **-checksum** option.

To run this test from the **Perform Memory Test** window, select the **CRC compare (nondestructive)** radio button. To run the test repeatedly, also select the **Repeat test** radio button and specify the number of times the test should run. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

To run this test using the **memtest** command, use the **-test=cc** option. To cause the CRC compare to run more than once, use **-repeat=number_of_tests** to specify the number of times the operation will be performed. See “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Find Start/End Ranges

The find start/end ranges test is a nondestructive test used to discover possibly unused memory at the start and end of the specified range. Sequences of identical values of the specified access size at the start or end of the specified range, if present, are reported as possibly unused memory.

**Note**

The term *range* is used in different ways in this test. The *specified range* represents the entire area of memory to be examined and is an input to this test. The *start range* and *end range*, if present, are subranges of the specified range and are the outputs of this test.

This test consists of two parts. First, the test reads memory locations sequentially from the start of the specified range. If multiple consecutive memory locations contain the same value, those consecutive, identical memory locations are reported as a potentially unused memory range. When the test reads the first memory location that contains a value different from all the earlier values, the first part of the test stops. The second part of the test is similar to the first part of the test, except that memory locations are read sequentially from the end of the specified range toward the start of the specified range. The output indicates the extent of identical memory values at the start and end of the provided range. (If the entire specified range contains the same value, the test will only report a single range equal to the specified range.)

For example, if the range 0x0000 to 0xffff is tested with an access size of 4, the test output might be something like:

```
Start range: 0x00000000
            to: 0x000000ff
            value: 0x00000000

End range: 0x0000ba4c
           to: 0x0000ffff
           value: 0xffffffff
```

This output indicates that the first 0x100 bytes of the range contain the 4-byte value 0x00000000. This output also implies that the next memory location contains a value other than 0x00000000. The final 0x45b4 bytes of the specified range (0xba4c-0xffff) contain the 4-byte value 0xffffffff. This output also implies that the location 0xba48 contains a value other than 0xffffffff.

To run this test from the **Perform Memory Test** window, select the **Find start/end ranges (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 515.

To run this test using the **memtest** command, use the **-test=fr** option. See “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Efficient Testing Methods

MULTI provides a wide variety of memory testing options ranging from simple to complex. You may want to devise a testing scheme to reap the most benefit from the various testing options. For instance, you might first use Debugger-based tests to verify that memory is initialized correctly and that there are no problems with data or address lines, and then move on to running tests using a target agent, in order to obtain broader testing coverage. A sample testing sequence using this model might be:

1. Perform one or more address and/or data walking tests from the Debugger.
2. Perform one or more pattern tests over a small area of memory from the Debugger.
3. Perform one or more of the other tests using the target agent across the entire range of memory.

Running Memory Tests from the Command Line

Memory testing can be performed using the **memtest** command. However, due to the vast array of test types and options, the syntax for this command can be quite complex. For this reason, we recommend that you use the **Memory Test Wizard** or the **Perform Memory Test** window to configure and run memory tests.



Note

If you want to write scripts that contain memory testing commands, you can still use the **Memory Test Wizard** or the **Perform Memory Test** window to help you determine the exact command syntax for the specific test and testing options you want to use. To do this, first use one of these graphical tools to configure the test you want to run, and then run the test. When the test completes, the **Memory Test Results** window will display the exact command syntax that corresponds to the testing options you specified in the GUI. You can use the command syntax given there in the Debugger command pane or in customized scripts.

For information about the **memtest** command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Detecting Coherency Errors

If the contents of memory differ from what you loaded onto your target, a coherency error may exist. In this context, *coherency* refers to the byte-value agreement between memory and the file that you originally loaded. For example, most programs contain a `.text` section that is not modified. As a result, the byte values of the `.text` section that appears in the executable program file should match the byte values of the `.text` section that is loaded into memory while the process is running.

Coherency errors may occur as the result of self-modifying code, buffer or stack overruns, bad memory hardware, or simply the wrong version of code being debugged. In the worst case scenario, code in memory is only slightly different from code in the executable program file, making it difficult to detect the discrepancy. MULTI offers two complementary ways of detecting this problem: manual coherency checking via the **verify** command and automatic coherency checking via a debugging setting. The next two sections describe each of these coherency checking methods.



Note

MULTI supports coherency checking with most debug servers. However, if the **Debug** → **Debug Settings** → **Auto Check Coherency** menu item is dimmed, most automatic coherency checking is not available in your current environment.

Checking Coherency Manually

To manually check the coherency of an address range, enter the following command in the command pane:

```
verify address_expression [num_addresses]
```

where:

- *address_expression* specifies the address expression at which to begin coherency checking.
- *num_addresses* specifies the number of addresses to verify past *address_expression*. If you omit *num_addresses*, this command verifies until the end of the function that encloses *address_expression*.

The Debugger compares the bytes in the specified range of target memory against the bytes in the content of the executable program file, and prints a list of addresses where they differ. The Debugger also highlights the lines in the source pane corresponding to those addresses to warn you that their contents differ.

To manually check the coherency of all downloaded non-data sections that cannot be written to, enter the **verify -all** command in the command pane. The `.text` section is one example of a section that you cannot write to. Because certain sections of memory, such as `.bss`, `.data`, and `.heap`, may be written to during program execution, you can expect them to differ from the executable program file. When you specify the **-all** option, the **verify** command does not check these sections. However, you can verify them manually by entering **verify -section *section_name*** in the command pane.

For comprehensive information about the **verify** command, see Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Checking Coherency Automatically

When automatic coherency checking is enabled, MULTI checks the coherency of the specified number of addresses at every stop. If it finds discrepancies, it highlights the differing lines in the source pane.

To enable automatic coherency checking, do one of the following:

- Select **Debug** → **Debug Settings** → **Auto Check Coherency**.
- Set the `_AUTO_CHECK_COHERENCY` system variable. Set this variable to nonzero to enable automatic checking; set it to zero to disable automatic checking. See also the `_AUTO_CHECK_COHERENCY` variable in “System Variables” on page 310.

MULTI attempts to check an equal number of addresses before and after the current program counter; however, it does not cross procedure boundaries. For example, if MULTI is stopped at the first instruction of a procedure, automatic checking inspects addresses only after the program counter. It continues for the specified number of addresses.

By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer). To change the number of addresses checked, do one of the following:

- Select **Debug** → **Debug Settings** → **Number Of Addresses To Check**. In the dialog box that appears, enter the number of addresses you want checked on each stop.
- Set the `_AUTO_CHECK_NUM_ADDRS` system variable to an appropriate value. See also the `_AUTO_CHECK_NUM_ADDRS` variable in “System Variables” on page 310.



Note

Because extra memory is read on every stop, use care when setting the number of addresses to be read. Setting a large number of addresses may slow normal run control.

Chapter 22

Programming Flash Memory

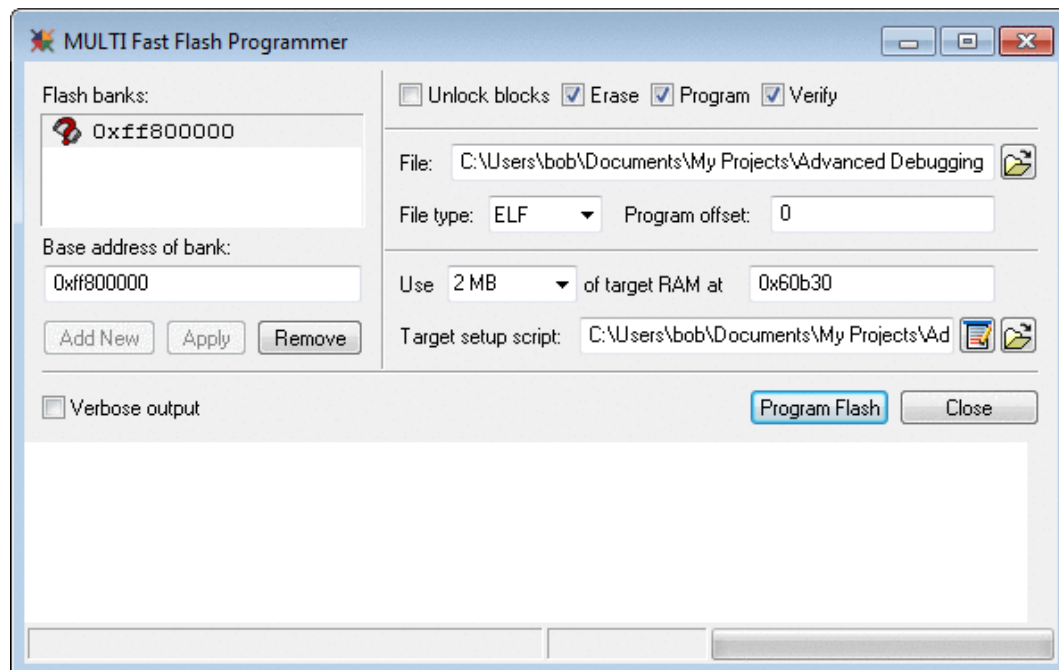
Contents

The MULTI Fast Flash Programmer Window	540
Prerequisites to Working with Flash	541
Using the MULTI Fast Flash Programmer	541
The MULTI Fast Flash Programmer GUI Reference	545
Flash Configuration File	546
Troubleshooting Flash Memory Operations	547

MULTI provides a graphical interface that makes it easy to write a memory image (either unformatted, in ELF format, or in S-Record format) from the host to flash memory on the target. Because flash memory is non-volatile, writing your program to flash memory allows it to run when the target is reset. This chapter describes how to erase, program, and verify flash memory using the **MULTI Fast Flash Programmer**.

The MULTI Fast Flash Programmer Window

The simplest way to program, verify, or erase flash memory is to use the **MULTI Fast Flash Programmer**.



To open this window, do one of the following:

- In the Debugger, select **Debug** → **Prepare Target**. In the **Prepare Target** dialog box that appears, select **Program Flash ROM**.
- In the Project Manager, select **Tools** → **Flash Selected Program**. (This option is only available if the selected file is a compiled program and you are connected to the target.)
- In the Debugger, select **Target** → **Flash**.

- In the Debugger command pane, enter the **flash gui** command. For information about this command, see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Prerequisites to Working with Flash

Before you can use MULTI to work with flash memory, you must ensure that you have met the following prerequisites:

- The setup script initializes the board so that flash memory is accessible and not cached.
- The setup script disables any external interrupts or watchdog timers that may interfere with flash programming.
- Your project's link map is linked for ROM and not RAM.

Flash programming speed is greatly enhanced by allocating more RAM for use by target agents. Choose the largest segment of RAM that can be accessed over the debug connection.

If the flash device has protected sectors that you need to overwrite, select the **Unlock sectors** option in the **MULTI Fast Flash Programmer**. Protected sectors may contain vital initialization code, so verify that the program image will not overwrite important data before setting this option. (This option is only available if you have selected **Erase** or **Program** in the top-right corner of the **MULTI Fast Flash Programmer**.)

Using the MULTI Fast Flash Programmer

The following sections describe how to perform important setup tasks and how to write to, verify, or erase flash memory.

Specifying Flash Banks

To add a flash memory bank, enter the base address in the **Base address of bank** field. (Base addresses can be specified in either decimal or hexadecimal format.)

Then click the **Add New** button. To change the base address of a bank already in the list, select it, enter the new base address, and click the **Apply** button.

Suppose your board has a CPU internal flash memory device and an external CFI-compatible flash memory device. You would need to specify two flash banks. To do so, you could:

1. Select the base address in the **Flash banks** pane.
2. Change the value in the **Base address of bank** field to be the base address of the internal device.
3. Click the **Apply** button.
4. Enter the address of the external flash device in the **Base address of bank** field.
5. Click the **Add New** button.

Performing these steps results in a list of two flash banks. The flash utility programs both of them as needed.

Choosing a File for Flash Operations

To select the file you want to write to flash memory, verify in flash memory, or erase from flash memory, enter the pathname of the file in the **File** field. You can also click  to open a file chooser and browse to the appropriate file. After selecting the file, use the **File Type** drop-down list to specify whether the selected file is in ELF format, S-Record format, or is unformatted (**Raw Binary**). If the file is unformatted, be sure to set the offset from the base of flash memory.

Setting a Write Offset

You can set a write offset by entering an offset value in the **Program offset** field. The meaning of the offset value differs for each file type, as described in the following list. (The offset value is optional and defaults to 0.)

- *ELF or S-Record format* — If the file you are writing to flash memory is in ELF or S-Record format, the offset is added to each address location specified in the file memory map. If the map correctly specifies where the program should be written, enter an offset of 0.

Offsetting an ELF or S-Record format file is helpful if you are downloading a program to flash memory and the flash bank has been mapped to a new location in memory. For example, the flash memory could be mapped to the same location as memory-mapped registers. If this is the case, you should re-map the flash memory. Because re-mapped flash memory is in a different location during programming than it is at reset, link the ELF program for the location where flash memory will be at reset, and enter the difference in addresses as the offset value.



Note

To write a program linked for RAM to flash memory, you must edit the linker directives file (see the documentation about linker directives files in the *MULTI: Building Applications* book). A program linked for RAM will not run if the offset is used to move the program to the flash area of the memory map.

- *Raw image* — If the file you are writing to flash memory is an unformatted memory image, the offset value specifies the location where the image will be written, relative to the first base address in the list. If you erase flash memory, the erase begins at the base address plus the offset.

Writing to Flash Memory

To write a file to flash memory on the target, use the **MULTI Fast Flash Programmer** to set the programming options. Select the **Erase** option in addition to the **Program** option if the flash sectors have not previously been erased. For a description of each option, see “The MULTI Fast Flash Programmer GUI Reference” on page 545. When you have set the applicable options, click the **Program Flash** button. MULTI downloads and runs target agents to modify the flash memory, unless no RAM was allocated.

The status of the flash operation is listed in the output pane of the window. You can cancel the operation at any time by clicking the **Cancel** button, which is enabled while the operation is in progress. Some flash operations may not respond immediately to the cancel request, and the **MULTI Fast Flash Programmer** may also wait for certain operations to complete before returning control to you.



Tip

If you experience problems, see “Troubleshooting Flash Memory Operations” on page 547.

For information about programming flash from a script, see the **flash burn** command in “General Memory Commands” in Chapter 10, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Erasing Flash Memory

You can also use the **MULTI Fast Flash Programmer** to initialize the flash device without writing an image to it. To do this, select **Erase**, clear **Program**, and set the other options with the same values that you would use for writing an image. For a description of each field, see “The MULTI Fast Flash Programmer GUI Reference” on page 545.

When you click **Erase Flash**, the target agent erases a region of flash memory the same size as the chosen image. For example, if you select an ELF file with 1 MB of data at the start of flash memory, this feature erases only the first 1 MB of flash. You can cancel the operation at any time by clicking the **Cancel** button. The window indicates the status of the erase in the output pane.



Tip

If you experience problems, see “Troubleshooting Flash Memory Operations” on page 547.

Verifying Flash Memory

To verify that the data written to the flash device contains a complete and up-to-date image of your program, open the **MULTI Fast Flash Programmer**, select **Verify**, and clear **Erase** and **Program**. Fill in the remaining settings as you would for a download. For a description of each field, see “The MULTI Fast Flash Programmer GUI Reference” on page 545.

When you click **Verify**, the **MULTI Fast Flash Programmer** compares the executable with the data on the target and stops when a difference is found.

The MULTI Fast Flash Programmer GUI Reference

The following table provides a brief description of the fields and buttons of the **MULTI Fast Flash Programmer**.

Flash banks	Specifies the base addresses of the flash memory in the target memory map.
Base address of bank	Allows you to modify the address of the flash bank selected in the Flash banks pane. Base addresses can be specified in either decimal or hexadecimal format. See also “Specifying Flash Banks” on page 541.
Add New	Adds the new flash bank (whose base address is specified in the Base address of bank field) to the Flash banks pane.
Apply	Applies changes made in the Base address of bank field.
Remove	Removes the flash bank selected in the Flash banks pane.
Unlock sectors	Allows protected sectors of your flash device to be overwritten. Protected sectors may contain vital initialization code, so verify that the program image will not overwrite important data before selecting this option. This option is only available if you have selected Erase or Program .
Erase Program Verify	Specifies whether MULTI attempts to erase flash memory, program flash memory, or verify flash memory when you click the Erase Flash, Program Flash, or Verify Flash button at the bottom of the window. The Erase option erases all the flash sectors in the programming range. Select this option to erase the flash chip, or select it in addition to the Program option to reprogram a chip whose flash range has previously been programmed. See also “Using the MULTI Fast Flash Programmer” on page 541.
File	• If the Erase operation is selected — Specifies the file to use to determine the amount of memory to erase. • If the Program operation is selected — Specifies the name of the file to be written. • If the Verify operation is selected — Specifies the source file for verification. Enter the name of the appropriate file or click  to open a file chooser and browse to the appropriate file. See also “Choosing a File for Flash Operations” on page 542.
File type	Specifies the format of the selected file. Select either ELF , Raw Binary , or S-Record .

Program offset	Specifies a program offset value. For ELF and S-Record format executables, this value is added to the addresses encoded in the file. For unformatted memory images, the first base address is added to this value to determine where in memory the file should be programmed. For more information, see “Setting a Write Offset” on page 542.
Use size of target RAM at location	Allows you to specify the size and location of RAM to be used by the flash utility. If the memory at the beginning of RAM is unusable, you should enter the first usable address, and leave the RAM size control set to the size of the full memory block. The RAM location field is only available if the RAM size selection is more than 0 KB .
Target setup script	Allows you to specify the .mbs format script that is run before accessing flash memory. If this field is empty, the MULTI Fast Flash Programmer will run the default setup script for the debug connection.
Verbose output	Prints warning and status messages that may be helpful for troubleshooting purposes.
Erase Flash Program Flash Verify Flash	Begins the flash operation according to the specifications you have entered in the window. Note: Because programming flash memory involves downloading target agents and running them, you should kill any processes running on the target before clicking the Erase Flash , Program Flash , or Verify Flash button. If the flash operation begins while other processes are running, the running processes terminate.
Close	Closes the MULTI Fast Flash Programmer window.

Flash Configuration File

The settings in the **MULTI Fast Flash Programmer** are persistently stored in the file **flash.cfg**, which is located in your personal configuration directory:

- Windows 7/Vista — *user_dir*\AppData\Roaming\GHS\
- Windows XP — *user_dir*\Application Data\GHS\
- Linux/Solaris — *user_dir*/.ghs/

The **flash.cfg** file contains an entry for each target type. Entries are updated after successful flash programming sessions. When the **MULTI Fast Flash Programmer** opens, these saved configuration settings replace any defaults read from the ELF file or debug information. Because the settings are indexed by target and not by

project, a new project for a target that has been successfully programmed will use the target's most recent settings rather than the default settings for a new project.

To restore all **MULTI Fast Flash Programmer** settings to the default values, remove or rename the **flash.cfg** file.

Troubleshooting Flash Memory Operations

The following text lists some problems that are commonly encountered when working with flash memory and gives some troubleshooting steps for each.

If the **MULTI Fast Flash Programmer** does not detect any flash memory after you click the **Program Flash** or **Erase Flash** button:

1. Check that the setup script initializes the board so that flash memory is accessible for both reads and writes.
2. Verify that the base addresses entered in the **MULTI Fast Flash Programmer** are the starting addresses of the flash chips.
3. Your flash chip may not be supported. For information about adding support for new flash memory chips, see the **flash_chips.odt** file located at *compiler_install_dir/defaults*.

If the **MULTI Fast Flash Programmer** prints messages indicating that writing the program sections was successful, but then fails to verify the image, the flash memory may have protected sectors that the flash driver cannot unlock. To fix this:

1. Refer to your flash chip documentation about procedures to unlock the protected sectors.
2. Following the procedure given in your flash chip datasheet, unlock the flash sectors in the setup script.

If the **MULTI Fast Flash Programmer** prints an error about a protected section of memory, set the **Unlock sectors** option. For more information about this option, see “The MULTI Fast Flash Programmer GUI Reference” on page 545.

Chapter 23

Working with ROM

Contents

Building an Executable for ROM	550
Executing a ROM Program	552
Debugging a ROM Program	554
Building an Executable for ROM-to-RAM	554
Executing a ROM-to-RAM Program	556
Debugging a ROM-to-RAM Program	557

The MULTI Integrated Development Environment enables embedded system programmers to:

- Build a project that can be transferred to the target's nonvolatile memory (ROM)
- Transfer a project's executable image to a target's flash memory
- Run and debug an executable that is located in ROM
- Run and debug an executable that is copied from ROM to RAM at startup

Building an Executable for ROM

An executable that is executed out of ROM has two primary differences from a RAM executable:

1. The linker directives file used to create the executable specifies that all of the compiled sections are in ROM.
2. Any necessary target board initialization may need to be done as part of the program's initialization instead of in a MULTI target setup script.

Creating a New Program

To create a new ROM executable image, perform the following steps:

1. If you do not already have one, create a MULTI project for your target configuration. For information about how to do this, see Chapter 1, “Creating a Project” in the *MULTI: Managing Projects and Configuring the IDE* book.
2. If you just created a MULTI project, you are automatically prompted to add items to it. Otherwise, select the Top Project and click the **Add Items** button () in the Project Manager.
3. On the **Project Manager: Select Item to Add** screen that appears, select a demo or example. If you want a framework to which you can add your own source code, select **Program**. Click **Next**.
4. Change the next screen's information as desired. Click **Next**.
5. Select **Link to and Execute out of ROM** from the **Program Layout** drop-down list, and make other selections as appropriate. Click **Finish**.

Your new program uses a Green-Hills-provided linker directives file that links all of the sections into ROM.

For more information about adding to projects, see “Managing Your Project” in Chapter 2, “Managing and Building Projects with the Project Manager” in the *MULTI: Managing Projects and Configuring the IDE* book.

Configuring an Existing Program

Your linker directives file must place all sections into ROM instead of RAM. If your existing program is not already using a Green-Hills-provided linker directives file configured for ROM, you can switch to using one by performing the following steps:

1. In the Project Manager, select the program.
2. Select **Edit** → **Configure**.
3. In the dialog box that appears, select **Link to and Execute out of ROM** from the **Program Layout** drop-down list.

For more information, see the documentation about linker directives files in the *MULTI: Building Applications* book.

If your program does not use a Green Hills linker directives file, modify it so that all sections will be placed into ROM instead of RAM. You must also define several special linker symbols in your linker directives file. This ensures that all MULTI ROM debugging features work properly. The special linker symbols you should define are:

Symbol	Value
__ghs_ramstart	The beginning of RAM.
__ghs_ramend	The end of RAM.
__ghs_romstart	The beginning of ROM.
__ghs_romend	The end of ROM.
__ghs_rambootcodestart	The beginning of the sections located in RAM. Should be 0 for ROM programs.
__ghs_rambootcodeend	The end of the sections located in RAM. Should be 0 for ROM programs.

Symbol	Value
<code>__ghs_rombootcodestart</code>	The beginning of the sections located in ROM.
<code>__ghs_rombootcodeend</code>	The end of the sections located in ROM.

Ranges of memory defined by `__ghs_*start` and `__ghs_*end` must follow these rules:

- The `__ghs_*end` symbol address should be after the corresponding `__ghs_*start` symbol address. Ranges are allowed to span address 0 as long as they do not wrap around the entire address space.
- The `__ghs_*start` symbol address should be the address of the first byte of the region you are describing. For example, if RAM is 8 megabytes starting from address `0x10000000`, `__ghs_ramstart` would be `0x10000000`.
- The `__ghs_*end` symbol address should be the address of the byte after the region you are describing. For example, if RAM is 8 megabytes starting from address `0x10000000`, `__ghs_ramend` would be `0x10800000`, not `0x107fffff`.
- If the `__ghs_*start` symbol is defined, the corresponding `__ghs_*end` symbol must be defined, and vice versa.
- Neither `__ghs_*start` nor `__ghs_*end` may have the value `0xffffffff` (on 32-bit systems) or `0xffffffffffffffff` (on 64-bit systems). If need be, you may be able to work around this restriction by subtracting 1 from the actual address.

Executing a ROM Program

Running a program that loads from ROM is different than running an executable that has been downloaded from the host to the target. First, you must transfer the image into ROM. For more information about transferring an image to ROM, see “Flashing Your Executable” on page 114.

Attaching to a Running ROM Process

If your program runs automatically when the target board is reset, you can attach to it after it has started running and begin debugging.

To attach to the process:

1. Reset your target to start your program.



Note

All target initialization must be done as part of the program's initialization instead of in a target setup script.

2. Open your program in the Debugger.
3. Select **Debug** → **Prepare Target**. In the **Prepare Target** dialog box that appears, select **Program already present on target**. MULTI matches the information on the target with the executable image you are debugging.
4. You may now begin debugging.

Advanced: Starting a ROM Program from the Debugger

You may be able to start your ROM program from the Debugger. To do so, perform the following steps:

1. Open your program in the Debugger.
2. Initialize your target board to the point where it can successfully start the program. For example, you can use the **setup** command to run your target setup script. If your program contains its own target initialization code, running the setup script is unnecessary, but you may still need to reset your target. To do so from the Debugger, you can click the **Reset** button (🔄) or issue the **reset** command. For information about the **setup** and **reset** commands, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
3. Select **Debug** → **Prepare Target**. In the **Prepare Target** dialog box that appears, select **Program already present on target**.
4. MULTI presents your target in its current state. If your target is not already at the beginning of your program, set the program counter to the beginning of the code you want to execute by entering `$pc = starting_address` in the Debugger command pane. For example, if your ROM boot code begins at the program's entry point address, you may want to enter `$pc = _ENTRYPOINT`.

5. Enter **rominithbp -setup** in the Debugger command pane to set the post-initialization hardware breakpoint. For information about the **rominithbp** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.
6. You may now begin debugging.

Debugging a ROM Program

Debugging a process running in ROM is very similar to debugging in RAM. The primary differences are:

- Software breakpoints cannot be used in ROM sections. Since software breakpoints are unavailable, clicking a breakdot will attempt to set a hardware breakpoint. The number of hardware breakpoints available varies from target to target, but it is usually fewer than four.



Note

MULTI uses hardware breakpoints to implement source-line (**s**) and function (**n**) stepping in ROM. If a hardware breakpoint is unavailable, the target may start running instead of stopping after executing the source line or function. Instruction stepping (**si**) is unaffected.

- ROM programs built with run-time error checking enabled handle run-time errors as if the Debugger is not connected. For more information, see the documentation about run-time error checks in the *MULTI: Building Applications* book.

Building an Executable for ROM-to-RAM

An executable which is loaded from ROM and copied into RAM has two primary differences from a RAM executable:

1. The linker directives file used to create the executable specifies the locations of uncompressed boot code sections in ROM, compressed code sections in ROM, and uncompressed code sections in RAM. The RAM code sections are

not actually part of the executable image, instead they are copied from ROM at run time by the boot code.

2. Any necessary target board initialization may need to be done as part of the program's initialization instead of in a MULTI target setup script.

Creating a New Program

To create a new ROM-to-RAM executable image, perform the steps listed in “Creating a New Program” on page 550, but instead of choosing **Link to and Execute out of ROM** from the **Program Layout** drop-down list, choose **Link to ROM and Execute out of RAM**.

Your new program contains a Green-Hills-provided linker directives file that links all of the sections into ROM and specifies which sections will be copied into RAM.

Configuring an Existing Program

Your linker directives file must place all boot code into ROM, and compressed versions of the remaining code into ROM. Additionally, the linker directives file must specify the destination in RAM of each code section.

If your existing program is not already using a Green-Hills-provided linker directives file configured for ROM-to-RAM, you can switch to using one by following the steps listed in “Configuring an Existing Program” on page 551, but instead of choosing **Link to and Execute out of ROM** from the **Program Layout** drop-down list, choose **Link to ROM and Execute out of RAM**. For more information, see the documentation about linker directives files in the *MULTI: Building Applications* book.

If your program does not use a Green Hills linker directives file, modify it so that it meets the requirements specified at the beginning of this section. Additionally, you must define several special linker symbols in your linker directives file. This ensures that all MULTI ROM-to-RAM debugging features work properly. The special linker symbols you should define are:

Symbol	Value
<code>__ghs_ramstart</code>	The beginning of RAM.
<code>__ghs_ramend</code>	The end of RAM.

Symbol	Value
<code>__ghs_romstart</code>	The beginning of ROM.
<code>__ghs_romend</code>	The end of ROM.
<code>__ghs_rambootcodestart</code>	The beginning of the sections located in RAM. All of the destination sections should be included.
<code>__ghs_rambootcodeend</code>	The end of the sections located in RAM. All of the destination sections should be included.
<code>__ghs_rombootcodestart</code>	The beginning of boot code sections located in ROM. Do not include any of the compressed sections.
<code>__ghs_rombootcodeend</code>	The end of boot code sections located in ROM. Do not include any of the compressed sections.
<code>__ghs_after_romcopy</code>	The first address executed after the ROM-to-RAM copy is complete.

For rules governing the use of `__ghs_*start` and `__ghs_*end`, see “Configuring an Existing Program” on page 551.

Executing a ROM-to-RAM Program

Running a ROM-to-RAM program is the same as running a ROM program. For instructions, see “Executing a ROM Program” on page 552.



Note

Your ROM-to-RAM program must be copied into RAM before software breakpoints can be set on the target and before system calls can be routed through the MULTI Debugger.

By default, MULTI adheres to the rules explained next when determining whether ROM-to-RAM copy initialization has occurred. You can use the **rominithbp** command to manually signal to MULTI when the ROM-to-RAM copy initialization is complete. For information about the **rominithbp** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

After you prepare your target with the **Program already present on target** option, MULTI determines the location of the program counter (PC) the first time the target halts. If the PC indicates that the next instruction is in RAM, MULTI assumes that

the program has been copied to RAM and lifts ROM debugging restrictions. If, however, the PC indicates that the next instruction is in ROM, MULTI assumes that the program has not yet been copied to RAM. In this case, MULTI automatically sets a hardware breakpoint, called the *post-initialization hardware breakpoint*, at `__ghs_after_romcopy`, which defaults to one of the first instructions in RAM. When this hardware breakpoint is hit, MULTI is signalled that ROM initialization is complete and that the program has been copied into RAM.

After MULTI has determined that the program has been copied to RAM, it performs all post-initialization actions (such as setting software breakpoints and triggering Debugger hooks). If the target was halted by the post-initialization hardware breakpoint, MULTI resumes it.

Debugging a ROM-to-RAM Program

Because the process is running out of RAM, there are no special limitations on debugging. However, if you need to debug any of the boot code that is executed before the copy into RAM occurs, you will be debugging in ROM. For more information, see “Debugging a ROM Program” on page 554.

Chapter 24

Non-Intrusive Debugging with Tracepoints

Contents

About Tracepoints	560
Working with Tracepoints	561
Collecting Debugging Information Non-Intrusively: Example	567
The Tracepoints Tab of the Breakpoints Window	571
Debugging in Passive Mode	574

This chapter describes how to perform non-intrusive debugging (also known as field debugging) using tracepoints, which allow you to collect useful debugging data without halting a process.

About Tracepoints

The MULTI Debugger allows you to perform non-intrusive debugging using tracepoints. A tracepoint is set at a particular instruction and collects the values of one or more variables each time the tracepoint is hit. Unlike the processing for standard breakpoints, all tracepoint processing is performed by the target. Thus, a tracepoint can be hit and collect data even when the Debugger is not connected to the target. Additionally, since MULTI does not have to stop the target to perform processing, using tracepoints should not significantly impede the normal functioning of the target.

Data collected by tracepoints accumulates in the tracepoint buffer for later retrieval. If the tracepoint buffer becomes full, then tracepoints will stop collecting data until more buffer space is made available by a purge operation (see “Purging the Tracepoint Buffer” on page 567). If a tracepoint is being hit more frequently than the threshold specified by the user when the tracepoint was created, then that particular tracepoint will be disabled and will no longer collect data.

Tracepoints are invaluable in situations where normal invasive debugging techniques, such as halting the process, may have negative consequences. If tracepoints are supported, you can use the MULTI Debugger's passive mode, which allows tracepoints but rejects more invasive debugging actions, such as setting breakpoints or halting the target. For more information, see “Debugging in Passive Mode” on page 574.



Note

Not all targets and operating systems support tracepoints. You must be connected to a target and operating system that support tracepoints in order to perform the actions described in this chapter.

Working with Tracepoints

Like breakpoints, tracepoints are displayed with an icon (🔍) in the Debugger source pane and are also listed in the **Breakpoints** window. To open the **Breakpoints** window, do one of the following:

- In the Debugger, click the **Breakpoints** button (🔍).
- In the Debugger, select **View** → **Breakpoints**.
- In the Debugger command pane, issue the **breakpoints** command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Choose the **Tracepoints** tab of the **Breakpoints** window to view tracepoint information. For a detailed description of all the tracepoint options available in this window, see “The Tracepoints Tab of the Breakpoints Window” on page 571.

The following sections describe how to perform the most common tracepoint tasks.



Note

Tracepoints, which are stored on the target, can collect information about the target even when MULTI is not connected to it.

Setting a Tracepoint

A tracepoint can be set in any of the following ways:

- From the Debugger source pane:
 1. Right-click the breakpoint to the left of the source line on which you want to set a new tracepoint.
 2. Choose **Set Tracepoint** from the shortcut menu that appears. This will open the **Tracepoint Editor** dialog.
 3. Enter appropriate values in the fields of the **Tracepoint Editor** dialog (see “Tracepoint Editor Dialog” on page 562) and click **OK**.
- From the **Breakpoints** window:
 1. Choose the **Tracepoints** tab.
 2. Click the **New Tracepoint** button to open the **Tracepoint Editor** dialog.

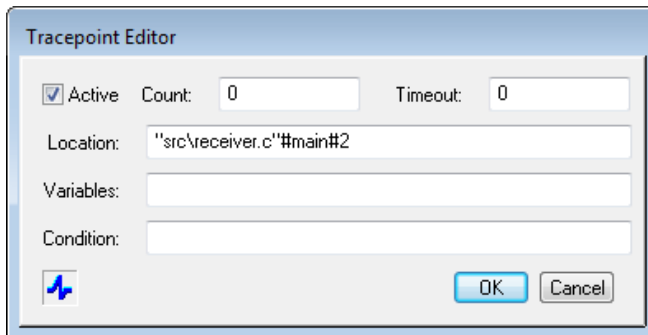
3. Enter appropriate values in the fields of the **Tracepoint Editor** dialog (see “Tracepoint Editor Dialog” on page 562) and click **OK**.
- From the command pane, enter the **tpset** command with appropriate arguments. For information about this command, see “Tracepoint Commands” in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

**Note**

It may not be possible to set tracepoints on some variables. For more information, see “Limitations: Information You Cannot Collect with Tracepoints” on page 570.

Tracepoint Editor Dialog

Tracepoints have several properties that you can modify. When you create or edit a tracepoint, a **Tracepoint Editor** dialog box opens that allows you to set these properties. None of your changes to the tracepoint will take effect until you click **OK**. You can click **Cancel** at any time to discard the changes you have made.



The table below lists the properties of a tracepoint that can be edited from this dialog.

Property	Meaning
Active	Indicates whether the tracepoint is active. When the tracepoint is active, a check appears in this box and the tracepoint records the values of its variables (see below) whenever the process reaches the tracepoint's location. When the tracepoint is inactive, it will not record any data when it is hit.

Property	Meaning
Count	Specifies a maximum number of times the tracepoint can be hit in a timeout period before being disabled. If the tracepoint is hit more than Count times in a period of Timeout time units, the tracepoint will automatically be disabled to prevent excessive performance degradation. If this field is set to 0, the tracepoint will never be automatically disabled.
Timeout	Specifies the length of a timeout period as a multiple of time units. The exact length and definition of the time units used by tracepoints is implementation-specific. If the tracepoint is hit more than Count times in a period of Timeout units, the tracepoint will automatically be disabled to prevent excessive performance degradation. If this field is set to 0, the tracepoint will never be automatically disabled.
Location	Specifies an address expression representing where this tracepoint will be set.
Variables	Specifies a comma-delimited list of symbols whose values should be stored when the tracepoint is hit. The symbols will be evaluated in the context of the tracepoint's source line.
Condition	Specifies an optional condition. For many targets, this condition is ignored, but in some cases it can be used by the operating system integration to determine if the tracepoint should collect data when it is hit. The format and interpretation of this field is implementation-specific. (For more information, consult the documentation for your specific operating system integration.)

Editing a Tracepoint

Once a tracepoint is set, you can use the **Tracepoint Editor** dialog to modify its settings. To open this dialog, do one of the following:

- In the source pane, right-click the tracepoint icon () and choose **Edit Tracepoint** from the shortcut menu that appears.
- In the **Tracepoints** tab of the **Breakpoints** window, double-click the tracepoint.
- In the **Tracepoints** tab of the **Breakpoints** window, select a tracepoint and click the **Edit** button.
- In the **Tracepoints** tab of the **Breakpoints** window, right-click a tracepoint and choose **Edit Selected Tracepoint** from the shortcut menu that appears.

- In the Debugger command pane, enter the **edittp** command. (For information about this command, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.)

See “Tracepoint Editor Dialog” on page 562 for a description of tracepoint properties you can edit from this dialog.

Listing Tracepoints

To view the list of active tracepoints, do one of the following:

- From the **Breakpoints** window, choose the **Tracepoints** tab, which displays all of the currently set tracepoints, both active and inactive (see “The Tracepoints Tab of the Breakpoints Window” on page 571 for more information).
- From the Debugger command pane, issue the **tplist** command. For information about this command, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

The Debugger caches the list of tracepoints to improve performance. To update the cache to reflect any tracepoints that have been automatically disabled, click the **Refresh** button on the **Tracepoints** tab in the **Breakpoints** window, or use the command **tplist refresh**.

Deleting a Tracepoint

To delete a tracepoint, do one of the following:

- From the Debugger source pane, click a tracepoint icon (🔍).
- From the **Breakpoints** window:
 1. Choose the **Tracepoints** tab.
 2. Select the tracepoint to be deleted.
 3. Click the **Delete** button.
- From the command pane, enter the **tpdel** command, using an address expression or an ID to indicate which tracepoint to delete.

For example, the tracepoint set in previous examples could be deleted by entering either `tpdel main#2` or `tpdel %0`. For more information, see the **tpdel** command in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Enabling or Disabling a Tracepoint

Tracepoints can be enabled and disabled. Disabled, or inactive, tracepoints do not collect data. To enable or disable a tracepoint, do one of the following:

- From the Debugger source pane, right-click the tracepoint icon () , and select **Enable Tracepoint** or **Disable Tracepoint**.
- From the **Breakpoints** window:
 1. Choose the **Tracepoints** tab.
 2. In the list of tracepoints, click in the **Active** column next to the line for the tracepoint to be enabled or disabled.
- From the Debugger command pane, use the **tpenable true** or **tpenable false** command, using an address expression or an ID to indicate which tracepoint to enable or disable, respectively.

For example:

```
> tpenable false %0
0 main#2:    0x101f4 200/400 <disabled> (argc,argv)
> tpenable true main#2
0 main#2:    0x101f4 200/400 (argc,argv)
```

For more information, see the **tpenable** command in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Resetting a Tracepoint

The **tpreset** command resets the hit count for a tracepoint to 0 (zero). You can use an address expression or an ID to specify which tracepoint to reset. For example, `tpreset %0` or `tpreset main#2` would reset the tracepoint. For more information, see the **tpreset** command in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing the Tracepoint Buffer

The data collected by tracepoints accumulates in the tracepoint buffer. To view the tracepoint buffer, do one of the following:

- From the **Breakpoints** window:
 1. Choose the **Tracepoints** tab.
 2. Click the **Dump Recorded Data** button.
- From the Debugger command pane, issue the **tpprint** command.

For example, the tracepoint used in the previous examples might yield the following data:

```
> tpprint
-----TRACEPOINT BUFFER CONTENTS-----
Tracepoint buffer contains 78 bytes.

Tracepoint Set  TID = 0x00005a5d (current name: <unknown>)
                 hello.c:main#2: 0x101f4
                 Timestamp = 0x00000001

Tracepoint Hit  TID = 0x00005a5d (current name: <unknown>)
                 hello.c:main#2: 0x101f4
                 Timestamp = 0x0000058c
                 (argc)  Memory =
                        0x00000001
                 (argv)  Memory =
                        0x00195010

Task Resumed    TID = 0x00005a5d (current name: <unknown>)
                 hello.c:main#2: 0x101f4
                 Timestamp = 0x0000058d
-----END TRACEPOINT BUFFER CONTENTS-----
```

In this example, three events are logged in the tracepoint buffer. First, a tracepoint is set at `main#2`. Then the tracepoint is hit at timestamp `0x0000058c` and logs the value `0x00000001` for `argc` and the value `0x00195010` for `argv`. Finally, at timestamp `0x0000058d`, the task is resumed after hitting the tracepoint. The amount of space used in the buffer is 78 bytes.

For more information, see the **tpprint** command in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Purging the Tracepoint Buffer

The size of the tracepoint buffer is finite, so it is useful to remove old data from the tracepoint buffer to make room for new data to be collected. This operation is performed with the **tppurge** command. The command **tppurge all** removes all data from the tracepoint buffer, while the command **tppurge size** removes *size* bytes from the beginning of the tracepoint buffer.



Caution

If the entire buffer is not purged, the user is responsible for selecting a size that specifies an integral number of events. It is recommended that only sizes displayed by the **tpprint** command be used when manually specifying *size*.

For more information, see the **tppurge** command in Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Collecting Debugging Information Non-Intrusively: Example

This section contains an extended example that is designed to demonstrate what information can and cannot be collected using tracepoints. The example is divided into the following parts:

- “Source Code for the Sample Program” on page 567
- “Examples of Valid tpset Commands” on page 568
- “Limitations: Information You Cannot Collect with Tracepoints” on page 570

Source Code for the Sample Program

This example uses a program compiled from the source code that follows. For simplicity, this program uses a naming convention in which **G** indicates *global*, **L** indicates *local*, and **M** indicates *member*. For example, **Gstruct** refers to a global structure.

```
struct my_struct
{
    int Mint;
};
```

```
class UT2
{
private:
    static int Mstaticint;
    int Mint;
public:
    UT2(int);
};

int Gint = 5;
int Garray[5] = {10, 11, 5, 7, 3};
my_struct    Gstruct;
my_struct    Garraystruct[1] = { Gstruct };
my_struct * Gstructptr = &Gstruct;
UT2* Gut2ptr;
int UT2::Mstaticint = 1;

int foo(int val)
{
    return val + 1;
}

int main(int Argint, char **argv)
{
    int Lint = 2;
    char Lchar = 'c';
    char* Lcharptr = "tpexample.cc";
    int Larray[4] = {1, 2, 3, 4};
    my_struct Lstruct;

    // variables and expressions that can be collected directly
    Lstruct.Mint = Argint + Lint;
    Garraystruct[0].Mint = (int) Lcharptr;

    Gint = Argint + Lchar + Lint;
    Gstruct.Mint = Lstruct.Mint + Larray[2] + Garray[1];

    Gut2ptr = new UT2(Lint);
LABEL:
    return 0;
}

UT2::UT2(int param)
{
    this->Mint = param;
}
```

Examples of Valid tpset Commands

The following examples demonstrate how to collect information, change the type of information being collected, and specify the tracepoint address using the **tpset** command. Most of these actions can also be performed using the **Tracepoint Editor**

dialog (see “Tracepoint Editor Dialog” on page 562). For a full description of the **tpset** command and other tracepoint commands, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

- To collect structure members, enter:

```
tpset 0/0 (Gstruct.Mint, Lstruct.Mint) main##LABEL
```

- To collect an array element, enter:

```
tpset 0/0 (Larray[1], Garray[2]) main##LABEL
```

- To collect an entire array, enter:

```
tpset 0/0 (Garray) main##LABEL
```

- To collect the value of an expression where the resulting value's location can be statically calculated, enter:

```
tpset 0/0 (Garraystruct[1].Mint) main##LABEL
```

- To collect member values when in a method, enter:

```
tpset 0/0 (Mint) UT2::UT2(int)#2
```

- To collect a string (note that the `char*` type is treated specially when gathering data), enter:

```
tpset 0/0 (Lcharptr) main##LABEL
```

- To collect a char pointer only, not the string, enter:

```
tpset 0/0 ((void*)Lcharptr) main##LABEL
```

- To collect a block of memory (for example, to collect 80 bytes at address `0x12345`), enter:

```
tpset 0/0 ((unsigned char[80])0x12345) main##LABEL
```

- To use a cast operation to change the kind of information collected (for example, to gather the integer value at the beginning of the string pointed to by `Lcharptr`), enter:

```
tpset 0/0 (*(int*)Lcharptr) main##LABEL
```

- To specify a tracepoint by a procedure-relative line number, enter:

```
tpset 0/0 (Gint) main#16
```

- To specify a tracepoint by a file-relative line number (`main##LABEL` is on line 46 of the sample source code), enter:

```
tpset 0/0 (Gint) "sample.cxx"#46
```

- To specify a tracepoint by address (for example, if `0x101d4` is the address of an instruction in the sample program), enter:

```
tpset 0/0 (Gint) 0x101d4
```

- To specify a tracepoint by a C++ method-relative line number, enter:

```
tpset 0/0 (Mint) UT2::UT2(int)#2
```

Limitations: Information You Cannot Collect with Tracepoints

Listed below are some examples of information that cannot be collected with tracepoints. Where possible, an example of an invalid **tpset** command that attempts to collect this information is given. For information about the **tpset** command, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

- You cannot collect a variable that requires multiple dereferences. For example, the following command is invalid:

```
tpset 0/0 (**argv) main##LABEL
```

- You cannot collect an array element indexed dynamically. For example, the following command is invalid:

```
tpset 0/0 (Garray[Lint]) main##LABEL
```

- You cannot collect a struct member from a struct pointer. For example, the following command is invalid:

```
tpset 0/0 (Gstructptr->Mint) main##LABEL
```

- You cannot collect the value of an expression where one or more portions cannot be determined statically (for example, the value of `Lstruct.Mintptr` is dynamic). For example, the following command is invalid:

```
tpset 0/0 (*Lstruct.Mintptr) main##LABEL
```

- You cannot collect the value of an expression that includes a function call or that would require execution of the application's target code. For example, the following command is invalid:

```
tpset 0/0 (foo(5)) main##LABEL
```

- You cannot collect a C++ class member when the `this` pointer does not exist. In some cases, when a member function makes no reference to the class members, the `this` pointer will be optimized away by the compiler. Consequently, the tracepoint processing has no way of statically determining where the class's members are at run time.

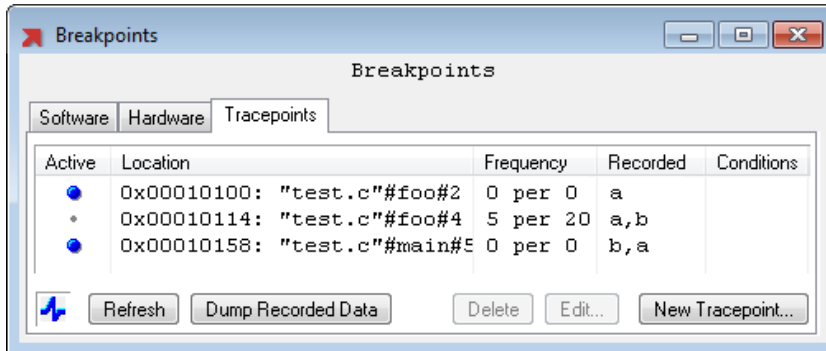
The Tracepoints Tab of the Breakpoints Window

The **Tracepoints** tab of the **Breakpoints** window contains a list of all the tracepoints in your program, along with some buttons to help you work with them.

To access the currently set tracepoints, open the **Breakpoints** window and choose the **Tracepoints** tab. To open the **Breakpoints** window, do one of the following:

- In the Debugger, click the **Breakpoints** button (.
- In the Debugger, select **View** → **Breakpoints**.

- In the Debugger command pane, enter the **breakpoints** command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.



The table below describes the buttons on the **Tracepoints** tab.

Button	Effect
Refresh	Reloads the list of tracepoints from the target. The target may deactivate tracepoints during your program's execution if their Count per Timeout limit is reached, so you may want to refresh the list of tracepoints periodically to see if any of them have been deactivated. In order to conserve target connection bandwidth, this is not done automatically.
Dump Recorded Data	Downloads any data that the tracepoints have recorded from the target and prints it to the command pane. In order to conserve target connection bandwidth, this is not done automatically.
Delete	Deletes the selected tracepoints. Once a tracepoint is deleted from its source line, your program will not record data there anymore until you set another tracepoint on the same line. If you only want to remove a tracepoint temporarily, you can disable it by clicking in the Active column.
Edit	Opens the Tracepoint Editor dialog (see “Tracepoint Editor Dialog” on page 562).
New Tracepoint	Opens a dialog box that allows you to create a new tracepoint.

For information about the MULTI commands corresponding to these buttons, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.

The mouse and keyboard actions listed next are available from the **Tracepoints** tab.

Action	Effect
Click a tracepoint	Displays the tracepoint's location in the Debugger.
Double-click a tracepoint or Press Enter	Opens the Tracepoint Editor dialog (see “Tracepoint Editor Dialog” on page 562).
Click the Active cell of a tracepoint	Toggles the tracepoint to active or inactive.
Press Delete	Deletes the selected tracepoint(s).
Right-click a tracepoint	Opens a shortcut menu of common actions. See the following table for a description of the menu items.

In the **Tracepoints** tab, you can use a context-sensitive menu to perform actions on selected tracepoints. To open this menu, select one or more tracepoints from the list and right-click. The shortcuts available from this menu are listed in the following table.

Item	Action
Show In Debugger	Displays the tracepoint's location in the Debugger.
Enable Selected Tracepoints	Makes all of the selected tracepoints active, so that the target will record their associated data when they are hit.
Disable Selected Tracepoints	Makes all of the selected tracepoints inactive, so that the target will not record their associated data when they are hit.
Edit Selected Tracepoint	Opens the Tracepoint Editor dialog on the selected tracepoint (see “Tracepoint Editor Dialog” on page 562).
Delete Selected Tracepoints	Deletes the selected tracepoints.
New Tracepoint	Opens the Tracepoint Editor dialog, which allows you to create a new tracepoint (see “Tracepoint Editor Dialog” on page 562).

Debugging in Passive Mode

MULTI can operate in passive mode, which causes the Debugger to reject invasive debugging actions that would significantly impact the functioning of the process being debugged. For example, while in passive mode, MULTI rejects attempts to halt the process, set breakpoints, write to memory, or write to registers. Setting tracepoints, however, is permitted in passive mode. MULTI determines the current mode (passive or non-passive) by querying the target when a connection is established.



Note

Passive mode is not available with all targets. Additionally, passive mode is not supported for programs compiled with run-time error checking information. (The MULTI Debugger sets a breakpoint in these programs to help it determine when a run-time error has occurred. This can result in your target halting while the Debugger is still in passive mode.)

For those operating system integrations that support passive mode, it is possible to enable or disable passive mode from the MULTI command pane. However, some operating system integrations require a password to enter or exit passive mode. Please refer to the documentation for your specific operating system integration to determine whether a password is necessary, and if so, what the default value of the password is.

Entering and Exiting Passive Mode

The **passive** command is used to enter and exit passive mode. For example, suppose the current operating system integration supports both passive mode and passive mode passwords, and that the current password for passive mode is `opensesame`. Below is an example of a series of commands that demonstrate the function of passive mode:

```
> passive on opensesame
Passive mode state successfully changed.
> halt
halting process...Halting is not allowed in passive mode.
> passive off opensesame
Passive mode state successfully changed.
```

```
> halt
halting process...done.
```

The `passive on opensesame` line causes MULTI to enter passive mode (if the current operating system integration did not require a password, then the first line would have been `passive on`). In this mode, intrusive debugging commands (such as the `halt` command issued in line three of the example) are rejected. The `passive off opensesame` line causes MULTI to exit passive mode. Thus, the `halt` command issued in the next line succeeds.

For more information about the **passive** command, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

Entering passive mode disables all active software and hardware breakpoints in the program you are debugging. These breakpoints must be re-enabled manually after exiting passive mode (that is, they are not automatically re-enabled).

The current state of passive mode is saved on the target for as long as the target is running. MULTI queries the target upon connection to determine whether the target is currently in passive mode. For example, if you connect to a target, enter passive mode, disconnect from the target (but leave the target running), and then reconnect to it, MULTI will still be in passive mode when you reconnect.

Setting the Passive Mode Password

The command:

```
passive password old_pw new_pw
```

can be used to change the passive mode password. Again, suppose the current password for passive mode is `opensesame`:

```
> passive password opensesame albatross
Passive mode state successfully changed.
```

This command changes the password for entering and exiting passive mode from `opensesame` to `albatross`. This use of the **passive** command has no meaning for operating system integrations that do not support passive mode passwords.

For more information about the **passive** command, see Chapter 21, “Tracepoint Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

For those operating system integrations that do support passive mode passwords, the default password is implementation specific. For more information, consult the documentation for your operating system integration.

Chapter 25

Run-Mode Debugging

Contents

Establishing Run-Mode Connections	578
Loading a Run-Mode Program	579
The Task Manager	580
Working with AddressSpaces and Tasks	581
Working with Task Groups in the Task Manager	584
Working with Run-Mode Breakpoints	587
Task Manager GUI Reference	592
Task Group Configuration File	600
OS-Awareness in Run Mode	601
Profiling All Tasks in Your System	603

When you debug a multitasking application, MULTI interacts with the target's operating system in run mode, in freeze mode, or in both run and freeze mode simultaneously. In run mode, the operating system kernel continues to run as you halt and examine individual tasks. In freeze mode, the entire target system stops when you examine tasks. For information about freeze-mode debugging, see Chapter 26, “Freeze-Mode Debugging and OS-Awareness” on page 605. For information about debugging in run mode and freeze mode, see “Automatically Establishing Run-Mode Connections” on page 69.

MULTI supports run-mode debugging for special target environments comprised of specific processor/RTOS/debug server combinations. The debug server must support multitasking applications.



Note

In this chapter, *task* is used as a general term to describe the real-time operating system's unit of execution. Specific terminology varies according to the target's operating system (for example, a task may also be called a *process* or a *thread*).

Establishing Run-Mode Connections

Before utilizing the run-mode debugging features described in this chapter, you must establish a run-mode connection to your target via a debug server that supports multitasking debugging, such as **rtserv** or **rtserv2**. For general information about connecting, see Chapter 3, “Connecting to Your Target” on page 39. For specific information about connecting with **rtserv** or **rtserv2**, see Chapter 5, “INDRT (rtserv) Connections” on page 77, or see Chapter 4, “INDRT2 (rtserv2) Connections” on page 59.

After you have connected to your target, the connection appears in the target list. It's usually denoted by an entry labeled **Run mode target**. For information about the way in which run-mode AddressSpaces and tasks appear in the target list, see “Debugging Target List Items” on page 16.



Note

If you want to simultaneously debug multiple targets in run mode, use one instance of MULTI to establish all of the corresponding run-mode connections. Distinct targets and applications are accessible for debugging

via the Debugger's target list (see “The Target List” on page 15). If you launch multiple instances of MULTI (for example, by double-clicking the MULTI shortcut twice or by launching MULTI from the command line twice), the result of operations between MULTI IDE tools is undefined.

Loading a Run-Mode Program

Some run-mode targets, including INTEGRITY, allow you to dynamically download programs onto the target after it has booted.

After establishing a run-mode connection, do one of the following to dynamically download a program:

- In the Debugger, select **Target** → **Load Module**.
- In the Task Manager, select **Load** → **Load Module**.
- In the Debugger command pane, enter the **load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

The exact requirements for loading a module depend on your target operating system. If you are running INTEGRITY, your target must be configured with a dynamic loader. For more information, see “Dynamic Downloading” in the *INTEGRITY Development Guide*. For general information about debugging dynamically downloaded INTEGRITY applications, see “Run-Mode Debugging” in the *INTEGRITY Development Guide* and “Troubleshooting” in the *INTEGRITY Development Guide*.



Note

Depending on your configuration, MULTI may attempt to restore breakpoints saved from previous debugging sessions when you download a program. (See the description of the **rememberBreakpoints** option in “Session Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.) INTEGRITY targets also allow programs to specify one or more tasks that may automatically start as soon as the program has finished downloading. Note that INTEGRITY versions 5 and earlier do not

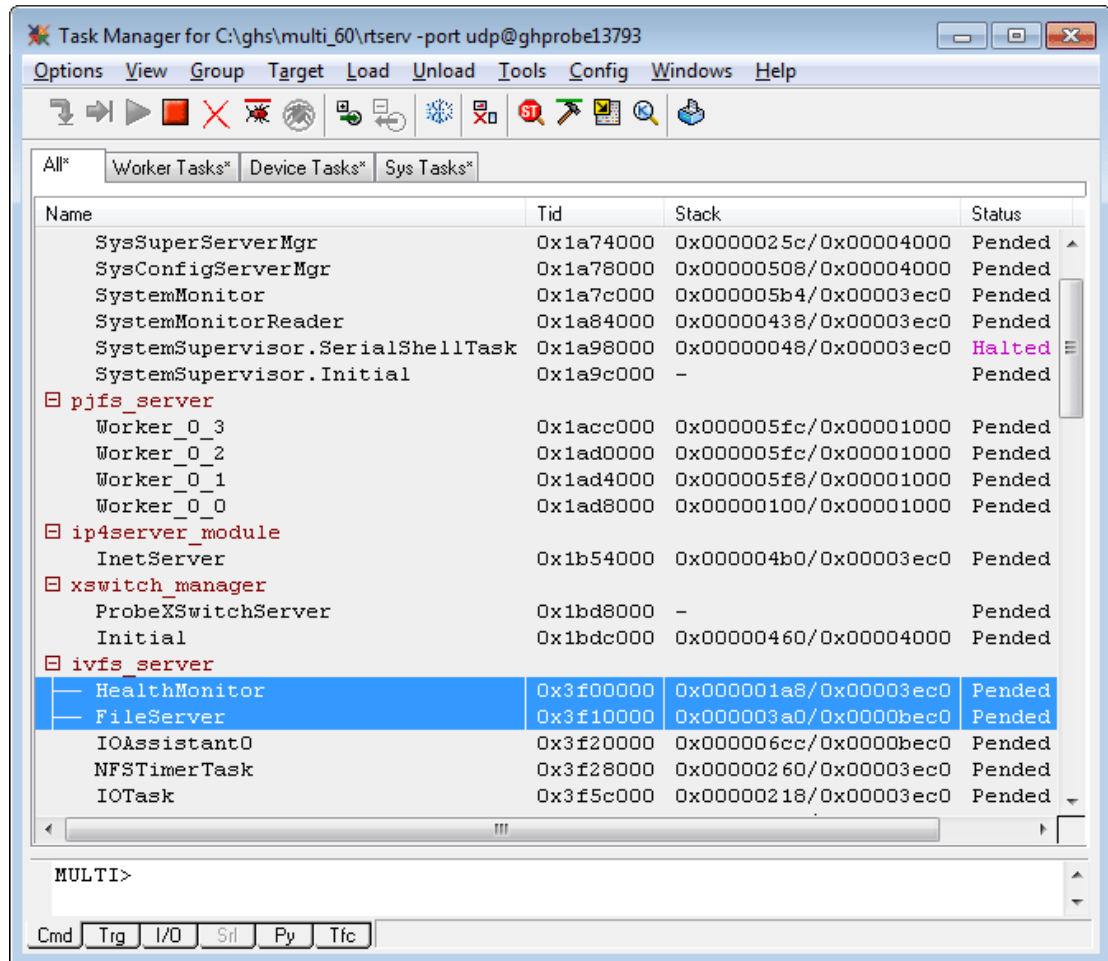
guarantee that MULTI will be able to install breakpoints on the target before tasks are allowed to begin execution. If you are using INTEGRITY version 5 or earlier and you want breakpoints to be restored, perform one of the following operations to prevent tasks from starting automatically:

- In the applicable `Task` section(s) of your Integrate configuration file (`.int`), set `StartIt` to `false`. Then rebuild your application.
- Before downloading your program in MULTI, enable **Debug** → **Target Settings** → **Stop on Task Creation**.

The Task Manager

During a run-mode debugging session, you can use the Task Manager to work with tasks created by your application. After connecting to a debug server that supports multitasking debugging (see “Establishing Run-Mode Connections” on page 578), do one of the following to open the Task Manager:

- In the **Connection Organizer**, select **Target** → **Show Task Manager**.
- In the Debugger, select **View** → **Task Manager**.
- In the Debugger command pane, enter the **taskwindow** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Each tab of the Task Manager corresponds to a *task group* (see “Working with Task Groups in the Task Manager” on page 584).

Working with AddressSpaces and Tasks

Run-Mode AddressSpaces

Run-mode AddressSpaces are located in the target list under a CPU node of a run-mode connection to the target, or they are located under an application (kernel image or dynamically downloaded module). The application is in turn located under a CPU node. A run-mode connection to a target is usually denoted by **Run mode target**.

If you are using INTEGRITY version 10 or later and you select a run-mode AddressSpace in the target list, the AddressSpace's source code appears in the source pane. In general, the following operations are available:

- Source browsing
- Breakpoint manipulation (All breakpoints set here are any-task breakpoints. For more information, see “Any-Task Breakpoints” on page 587.)

If you are not using INTEGRITY version 10 or later and you select a run-mode AddressSpace in the target list, the source pane is empty and the preceding operations are unavailable.

Attaching to Tasks

Attaching to a task allows MULTI to obtain debugging access to that task. When you attach to a task, MULTI automatically displays the task in a Debugger window.

Some operating systems require that you attach to a task before halting, killing, or running it, while other operating systems allow you to perform these actions on unattached tasks.

To attach to a task, do one of the following:

- In the Task Manager, double-click the task.
- In the Debugger's target list, select the task. (For information about identifying run-mode tasks in the target list, see “Debugging Target List Items” on page 16.)



Note

When you have attached to a task that is listed as **Halted** in the target list's **Status** column, you should avoid using programmatic means (for example, INTEGRITY's `RunTask()` kernel call) to cause that task to run. MULTI is not guaranteed to detect such target-initiated status changes. To work around this problem, issue the **detach** command to detach from the task, preferably before the target causes it to run. For information about the **detach** command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

Halting Tasks On Attach

Some operating systems require that you attach to a task before halting it. To automatically halt a task as soon as you attach to it, do one of the following:

- In the Task Manager, select **Options** → **Halt on Attach**.
- In the Debugger, select **Debug** → **Target Settings** → **Halt on Attach**.

When the attached task opens in the Debugger, it is already halted.



Note

The **Halt on Attach** option is not available when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).

Debugging Run-Mode Tasks

When you attach to a task, MULTI automatically displays the task in the Debugger window. To automatically display tasks in the Debugger as soon as the application creates them, enable **Stop On Task Creation** and **Attach on Task Creation** from the Task Manager's **Options** menu or from the Debugger's **Debug** → **Target Settings** menu.

If you open a new Debugger window to debug each new task (see “Opening Multiple Debugger Windows” on page 14), MULTI color-codes each task according to its corresponding Debugger window. Any window that relates to debugging a particular task is shaded in the same color as the task is shaded in the Task Manager. For example, suppose you attach to a task that is shaded blue in the Task Manager. The Debugger window that shows the task's code is also shaded blue. If you use a Browse window to look at the procedures of the task, the Browse window is also shaded blue.

Once a task is open in a Debugger window, you can halt or run the task from the Debugger window itself or from the Task Manager. If a breakpoint is hit in a task other than the one currently selected in the target list, the Debugger switches to that task by default (provided that no other Debugger window is open on it and that the currently selected task is not halted). For information about the configuration option that allows you to control this behavior, see **targetWindowSwitchViewOnBpHit** in “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

For information about debugging with TimeMachine, see “Using TimeMachine with OS Tasks” on page 418.

Freezing the Task Manager's Task List Display

Because run-mode debugging allows some tasks to continue running while you debug other tasks, the data in the Task Manager changes constantly. If you want to take a snapshot of all the tasks at a certain moment, choose **View → Freeze Task List**. This option freezes the task list display (see “Task List Pane” on page 596), not the underlying operating system. When the display is frozen, a message appears in the status bar at the bottom of the Task Manager. You can modify task groups while the window is frozen.



Tip

To change the rate at which MULTI refreshes the tasks in an unfrozen Task Manager:

1. In the Debugger or Project Manager, select **Config → Options**.
2. In the **Options** window that appears, click the **Debugger** tab.
3. Click the **More Debugger Options** button.
4. Edit the **Interval to refresh Task Manager** field. For more information, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Working with Task Groups in the Task Manager

Task groups allow you to organize tasks, making it easier to work with multiple tasks simultaneously. In addition, task groups provide the approach for group breakpoints. (For information about group breakpoints, see “Group Breakpoints” on page 588.)



Note

The Task Manager must be open in order to guarantee the availability of all task group features.

A task group can contain any task in the system, including tasks running on different processors or in different address spaces. As you create task groups, they are displayed as new tabs in the Task Manager. The name of the task group must consist of one or more characters and cannot contain square brackets ([and]). In addition, you cannot name a task group with one of the default task group names created by MULTI (see next section).

To quickly create a task group that contains the appropriate tasks, select the tasks in the Task Manager and click .

You can attach to, halt, kill, and run all the tasks in a task group with a single action. The **Group** menu contains several options that act on all of the tasks in the current task group.

MULTI supports task groups whether or not the underlying debug server and debug agent on the target support task groups. The ability of the underlying debug server and target debug agent to support task groups affects synchronous operation latency. For more information, see “Synchronous Operations” on page 589.

Default Task Groups

MULTI creates the following two default task groups:

- **All** — Contains all tasks that the operating system is using to run the application. There are some internal RTOS tasks, however, that the operating system may not include in the **All** group.
- **System** — Contains all tasks running on the target system, including the internal RTOS tasks that the operating system excludes from the **All** group. For some operating system/debug server combinations, the set of tasks contained in the **System** group is identical to the set of tasks contained in the **All** group.

In general, task group names are case-sensitive, but MULTI treats the two default groups as being case-insensitive. As a result, you cannot create a group named **All** or **System**, no matter what the case combination. The **All** group has a tab in the Task Manager, but the **System** group does not.

You cannot add tasks into or delete tasks from the **All** or **System** default groups.

**Tip**

When both the operating system and the debug server support task groups and distinguish between the **System** and **All** default groups, halting the group **System** freezes the target board and all application tasks. In this environment, you can execute operations such as reliably dumping the event log section, browsing kernel objects with the **OSA Explorer**, etc. To halt the **System** task group, select **Group** → **Halt System**.

MULTI reserves the following task group names. They are not shown as tabs in the Task Manager, but you should not use them for groups you create:

- **Current Task** – A label used to represent the special task group containing only the corresponding task in the **Software Breakpoint Editor** (see “Creating and Editing Software Breakpoints” on page 130).
- **Group N/A** – A label used to indicate that task group is not available in the **Software Breakpoint Editor** (see “Creating and Editing Software Breakpoints” on page 130).
- **__multi_tmp_op_grp_*** – A temporary group name created by MULTI to perform synchronous operations on the selected task if the underlying debug server supports task groups. Whenever MULTI sends the synchronous operation to the underlying debug server, the temporary group is automatically destroyed. For more information, see “Synchronous Operations” on page 589.
- **__multi_tmp_bp_grp_*** – A temporary group name created by MULTI to set a group breakpoint on one task. The group only contains the corresponding task. When the corresponding group breakpoint is deleted, the temporary group is automatically destroyed.
- **All AddressSpace names for INTEGRITY** — On INTEGRITY versions 5 and later, AddressSpace names can be used as groups when using group breakpoints. However, on INTEGRITY version 5 only, AddressSpace names cannot be used as groups when using the **groupaction** command. (For information about the **groupaction** command, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.) You cannot add tasks into or delete tasks from any AddressSpace group via the Task Manager.

Configuring Task Group Settings

For advanced configuration options, see “Task Group Configuration File” on page 600.

Working with Run-Mode Breakpoints

MULTI supports three types of run-mode breakpoints: task-specific breakpoints, any-task breakpoints, and group breakpoints. Each of these is described in the following sections.

Task-Specific Breakpoints

A task-specific breakpoint is a breakpoint that only a single task can hit. To set a task-specific breakpoint, perform the following steps:

1. Attach to a task by double-clicking it in the Task Manager or selecting it in the Debugger's target list.
2. In the Debugger window, click a breakdot ().

A task-specific breakpoint is indicated by a normal breakpoint icon ().

Any-Task Breakpoints

An any-task breakpoint is a breakpoint that every task in the AddressSpace (except system tasks, which cannot be debugged in run mode) can hit. An any-task breakpoint is indicated by a breakpoint icon with the letters **AT** inside of it ().

You can set an any-task breakpoint in a run-mode AddressSpace (if the target RTOS is supported) or in a task. To set the breakpoint from a run-mode AddressSpace, select the AddressSpace in the target list and then click a breakdot () in the Debugger window.

To set the breakpoint from a task, select the task in the target list and hold the **Shift** key as you click a breakdot () in the Debugger window.

To set an any-task breakpoint from the command pane, use the **sb at** command.

For example:

```
sb at function#5
```

sets an any-task breakpoint at line 5 of the function *function*. For more information about the **sb** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

When a task hits an any-task breakpoint, all other running tasks continue to execute.

Group Breakpoints

A group breakpoint is a breakpoint set on a group of tasks such that any task in the group can hit the breakpoint. When a task hits the breakpoint, the task, the whole group, or another group of tasks stops. In addition to saving time, applying a halt operation to an entire task group allows you to perform synchronous halts on multiple tasks (see “Synchronous Operations” on page 589). A group breakpoint is indicated by a breakpoint icon with the letters **GT** inside it (.

You can set a group breakpoint via the **Software Breakpoint Editor** (see “Creating and Editing Software Breakpoints” on page 130) or via the **b** command. For information about the **b** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Group breakpoints are not available in all environments. If you cannot set a group breakpoint but want to make use of a similar behavior, you can use the **b** command to set one or more individual breakpoints that invoke the **groupaction** command. The **groupaction** command allows you to halt all tasks belonging to the specified task group(s). For an example, see “Setting Breakpoints for Task Groups” on page 589. For more information about the **b** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book. For more information about the **groupaction** command, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.



Note

The Debugger may not be able to restore saved group breakpoints.

Synchronous Operations

Synchronous operations run and/or halt a group of tasks *almost* at the same time. The accuracy of synchronous operations varies for different debugging environments. If the debug server and the corresponding RTOS debug agent support task groups (as is the case with INTEGRITY), accurate synchronous operations and group breakpoints can be achieved; otherwise, group breakpoints are not supported. In this case MULTI sends out separate commands to each task, and the latency time for the operations on different tasks is unpredictable (depending on various factors such as network traffic, the RTOS debug agent's status, the target's speed, etc.).

On an RTOS that supports task groups, operations on a group will be synchronous if the group is *fine*. To determine whether a group is fine, select **View** → **Visualize Fine Groups** in the Task Manager. If the tab corresponding to the group is marked with an asterisk, the group is fine.

Manipulating a group may change its status. For example, if you are using a debug server that does not support tasks from different CPUs in the same group, and you try to add tasks from different CPUs into an existing fine group, the group's status becomes *not fine*.

When performing operations on multiple tasks selected in the Task Manager, MULTI attempts to take advantage of task group support in the debug server and in the corresponding RTOS debug agent. For example, when you halt selected tasks, MULTI attempts to create a temporary task group for the selected tasks. If such a fine group can be created on the debug server, MULTI sends one command to the debug server for the entire group, thus guaranteeing synchronization accuracy by the debug server and the RTOS debug agent. When performing operations on multiple tasks selected in the target list, however, MULTI does not attempt to construct a fine group. Instead, MULTI sends separate commands for each task selected, performing run-control operations one at a time.

Setting Breakpoints for Task Groups

This section contains specific information about using task groups when setting breakpoints. For information about task groups, see “Working with Task Groups in the Task Manager” on page 584. For more information about the **b** command referenced in this section, see Chapter 3, “Breakpoint Command Reference” in the

MULTI: Debugging Command Reference book. For more information about group breakpoints, also referenced in this section, see “Group Breakpoints” on page 588.

Task groups are very powerful when setting breakpoints. By using task groups, you can:

- Set a group breakpoint such that any task in the group can hit the breakpoint.

For example, to set a group breakpoint at address 0x1423 on the **Tasks in App1** task group, enter the following in the Debugger's command pane:

```
b /type_gt @"Tasks in App1" 0x1423
```

When any task that belongs to the **Tasks in App1** task group hits the breakpoint, that task is stopped.

- Stop an entire task group when a group breakpoint is hit.

For example, suppose you want to halt all tasks in the **Callers** task group whenever a task in the **Tasks in App1** task group hits a breakpoint. You would enter:

```
b /type_gt @"Tasks in App1" /trigger @"Callers" 0x1423
```

When any task that belongs to the **Tasks in App1** task group hits the breakpoint, that task, along with all tasks in the **Callers** task group, is stopped. (The `/trigger @task_group` argument simultaneously stops multiple tasks when a breakpoint is hit.)

If you want to create a group breakpoint only on the current task to stop all tasks in another group, it is not necessary for you to explicitly create a task group only containing the current task. MULTI does so automatically, and when the breakpoint is removed, the corresponding temporary task group is automatically destroyed.

For example, suppose you want to halt all tasks in the **Callers** task group whenever the current task hits a breakpoint. You would enter:

```
b /type_gt /trigger @"Callers" 0x1423
```

If, as in the following command, you do not specify `/trigger @task_group`:

```
b /type_gt 0x1423
```

the effect of the group breakpoint is equivalent to the effect of a normal breakpoint.

- Use an individual breakpoint to invoke an action on an entire task group. For example, to create a breakpoint on the current task that will halt all tasks in the **Pizza Tasks** group, enter:

```
b main#21 {groupaction -h @"Pizza Tasks"}
```

The **groupaction** command also supports running tasks (`-r`) and, for some operating systems, stepping tasks (`-s`). For more information about this command, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.

As long as the target operating system supports task groups, these actions are performed on individual tasks almost simultaneously. If an operating system does not support task groups, MULTI sends out separate commands to each task in the task group. In this case, the latency time for the operations on different tasks is unpredictable, depending on various factors such as network traffic, the RTOS debug agent's status, and the target's speed.



Note

When synchronous group operations are specified in a breakpoint command list (as in the preceding example), the amount of time that elapses between hitting the breakpoint and executing the synchronous group operations is not predictable.

Task Manager GUI Reference

This section describes the individual options available in the Task Manager.



Note

If a debugging environment does not support a certain option, that option is not displayed in the Task Manager.

Menu Bar

The Task Manager contains the following menus:

- The **Options**, **View**, and **Group** menus are described below.
- The **Target**, **Load**, and **Unload** menus contain menu items that are duplicated in the Debugger's **Target** menu. See “The Target Menu” on page 680.
- The **Tools** menu contains menu items that are duplicated in the Debugger's **Tools** menu. See “The Tools Menu” on page 685.
- The **Config** menu is the same as the Debugger's **Config** menu. See “The Config Menu” on page 687.
- The **Windows** menu is the same as the Debugger's **Windows** menu. See “The Windows Menu” on page 690.
- The **Help** menu contains menu items that are duplicated in the Debugger's **Help** menu. See “The Help Menu” on page 691. In addition, the **Task Manager Help** menu item has been added to show Task Manager online help information.

Options Menu

The Task Manager **Options** menu contains the following menu items.



Note

MULTI remembers the settings of the toggle menu items across debugging sessions.

Item	Meaning
Stop on Task Creation	Specifies whether you want the target's operating system to halt each task as soon as it is created by the application.

Item	Meaning
Attach on Task Creation	Specifies whether you want to debug the newly created task. If enabled, each newly created task is halted and opened in the Debugger window. This option is applicable only when the Stop on Task Creation menu item is selected.
Halt on Attach	Specifies whether you want MULTI to automatically halt tasks when you attach to them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).
Run on Detach	Specifies whether you want MULTI to automatically run tasks when you detach from them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).
Print	Prints the tasks currently listed in the Task Manager.
Close	Closes the Task Manager.

View Menu

The Task Manager **View** menu contains the following menu items.

Item	Meaning
Expand Selected Entries	Recursively expands all selected entries in the task list hierarchy.
Expand All Entries	Recursively expands all entries in the task list hierarchy.
Collapse Selected Entries	Collapses all selected entries in the task list hierarchy.
Collapse All Entries	Collapses all entries in the task list hierarchy.
Freeze Task List	Freezes or unfreezes the task list display. For more information, see “Freezing the Task Manager’s Task List Display” on page 584.
Flat View	Changes how tasks are displayed. When Flat View is selected, the tasks are combined together in a single list, regardless of whether they belong to the same processor or high-level RTOS object. When Flat View is cleared, tasks are organized according to processor or high-level object. For more information, see “Task List Pane” on page 596.
Visualize Fine Groups	Toggles the display of an asterisk (*) on Task Manager tabs that correspond to fine task groups. For more information, see “Synchronous Operations” on page 589.
Show AddressSpace IDs	Toggles the display of Address Space IDs in the second column of the Task Manager.

Item	Meaning
OSA Explorer	Opens the OSA Explorer for the RTOS running on the target. For more information, see “The OSA Explorer” on page 602. This menu item is only available for some RTOSes.
Group Breakpoints	Opens the Breakpoints window for group breakpoints set on the current connection. For reference information about the Breakpoints window, see “The Breakpoints Window” on page 146.

Group Menu

The Task Manager **Group** menu contains the following menu items.

Item	Meaning
Create New Group	Creates a new task group.
Delete Existing Group	Deletes an existing task group.
Add Selected Tasks into Another Group	Adds the selected tasks in the current task group into another task group.
Delete Selected Tasks	Removes the selected tasks from the current task group. You cannot delete tasks from the default task group All .
Continue Selected Tasks	Resumes the selected tasks in the current task group.
Halt Selected Tasks	Halts the selected tasks in the current task group.
Single Step (into function) Selected Tasks	Single-steps the selected tasks in the current task group, stepping into functions.
Single Step (over function) Selected Tasks	Single-steps the selected tasks in the current task group, stepping over functions.
Kill Selected Tasks	Kills the selected tasks in the current task group.
Attach to Selected Tasks	Attaches to the selected tasks in the current task group.
Detach from Selected Tasks	Detaches from the selected tasks in the current task group.
Continue Tasks in Current Group	Resumes all tasks in the current task group.

Item	Meaning
Halt Tasks in Current Group	Halts all tasks in the current task group.
Kill Tasks in Current Group	Kills all tasks in the current task group.
Attach to Tasks in Current Group	Attaches to all tasks in the current task group.
Detach from Tasks in Current Group	Detaches from all tasks in the current task group.
Halt System	Halts all tasks on the target system, thereby freezing the target. This option is not supported on all operating systems.
Run System	Restores the tasks on the target to the status they held before the system was halted.

For more information about task groups, refer to “Working with Task Groups in the Task Manager” on page 584.

Toolbar

The Task Manager toolbar contains the following buttons.

Button	Meaning
	Single-steps the selected tasks in the current task group, stepping into functions.
	Single-steps the selected tasks in the current task group, stepping over functions.
	Resumes the selected tasks in the current task group.
	Halts the selected tasks in the current task group.
	Kills the selected tasks in the current task group.
	Attaches to the selected tasks in the current task group.
	Detaches from the selected tasks in the current task group.
	Adds the selected tasks in the current task group into another task group.

Button	Meaning
	Removes the selected tasks from the current task group. You cannot delete tasks from the default task group All .
	Freezes or unfreezes the display of the Task Manager window. When the display is frozen (the button is down), a message appears in the status bar at the bottom of the window. For more information, see “Freezing the Task Manager's Task List Display” on page 584.
	Disconnects from the target.
	Opens a Breakpoints window for the group breakpoints set on the current connection. See “Viewing Breakpoint and Tracepoint Information” on page 128.
	Opens the current task's program in the Project Manager. If there is no current task, the default project opens in the Project Manager.
	Dumps the event log and displays events in the MULTI EventAnalyzer. This button is only available for some RTOSes.
	Opens the OSA Explorer on the RTOS running on the target. For more information, see “The OSA Explorer” on page 602. This button is only available for some RTOSes.
	Prints the tasks currently listed in the Task Manager.

Task List Pane

The area below the toolbar is the task list pane, where tasks and other kinds of objects (such as CPU nodes) are listed. The columns that are displayed for each task vary according to your operating system. For example, a column might be labeled **Thread** or **Process** depending on the terminology used by the operating system.

By default, tasks are listed in a hierarchical structure. The leaf nodes in the hierarchies are the tasks; the non-leaf nodes represent more general concepts in the corresponding RTOS (for example, CPUs and AddressSpaces on INTEGRITY). Non-leaf nodes cannot be sorted. To flatten the hierarchy and list all tasks in one list, choose **View** → **Flat View**. (This menu item is not available for customized task groups.)

Non-leaf nodes are color-coded according to MULTI's color settings. They are displayed as follows:

- First-level nodes (from the top of the hierarchy) are displayed in the color specified for strings.
- Second-level nodes are displayed in the color specified for characters.
- Third-level nodes are displayed in the color specified for integers.

Task information is displayed in the (foreground) text color. Whenever a task is displayed in a Debugger, the corresponding entry in the task list pane has the same background color as the corresponding Debugger window.

Debugging-related statuses in the **Status** column are colored so that you can easily identify them. RTOS statuses are not colored. Many of the statuses that appear in the Task Manager also appear in the Debugger target list. For more information, see “The Status Column” on page 19.

Each time the Task Manager is opened, MULTI automatically displays tasks with the hierarchies expanded. After that, you control how the hierarchies are displayed. Whenever the Task Manager window is refreshed, MULTI automatically resizes the columns so that all text is visible. However, once you manually change a column's width, MULTI no longer resizes the columns when it refreshes the display.

MULTI maintains changes that you make to the Task Manager window (such as modifications to its position) across debugging sessions.

Information Pane

The area below the task list pane is the information pane, in which the MULTI command pane, target pane, I/O pane, serial terminal pane, Python pane, or traffic pane can be displayed.

By default, the MULTI command pane is shown in the information pane area, but you can switch to another pane by clicking the corresponding tab at the bottom of the Task Manager window.

Tab	Meaning
Cmd or Cmd*	Shows the MULTI command pane in the information pane area. The MULTI command pane located in the Task Manager supports a subset of the commands that the command pane located in the Debugger does. If you issue a command in the Task Manager, and the output reports that it does not work in the current context, run it from the Debugger instead.
Trg or Trg*	Shows the target pane in the information pane area.
I/O or I/O*	Shows the I/O pane in the information pane area.
Srl or Srl*	Shows the serial terminal pane in the information pane area.
Py or Py*	Shows the Python pane in the information pane area.
Tfc or Tfc*	Shows the traffic pane in the information pane area.

As in the MULTI Debugger window, an asterisk (*) at the end of a tab name indicates that new messages have arrived in the corresponding pane.

You can change the height of the information pane by dragging the bar that separates it from the task list pane. To automatically resize the panes so that they share the vertical window space evenly, double-click the separator bar.

The Task Manager Shortcut Menu

You can use the mouse to perform useful operations on selected objects, such as expanding or contracting a node and attaching to tasks. Right-clicking in the task list pane opens a shortcut menu with the following items, which perform the same functions as the corresponding items in the menu and toolbar. The appearance of certain menu items is dependent upon your debug server.

Item	Meaning
Stop on Task Creation	Specifies whether you want the target's operating system to halt each task as soon as it is created by the application.
Debug on Task Creation	Specifies whether you want to debug newly created tasks. If selected, each newly created task is halted and displayed in a Debugger window. This option is applicable only when Stop on Task Creation is also selected.
Halt on Attach	Specifies whether you want MULTI to automatically halt tasks when you attach to them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).

Item	Meaning
Run on Detach	Specifies whether you want MULTI to automatically run tasks when you detach from them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).
Freeze Task List	Freezes or unfreezes the task list display. For more information, see “Freezing the Task Manager's Task List Display” on page 584.
Flat View	Changes how tasks are displayed. When Flat View is selected, the tasks are combined together in a single list, regardless of whether they belong to the same processor or high-level RTOS object. When Flat View is cleared, tasks are organized according to processor or high-level object. This menu item is not available for customized task groups. For more information, see “Task List Pane” on page 596.
Expand Selected Entries	Expands the entire sub-tree for all selected entries. This has no effect on the task entries because they have no sub-trees.
Attach to Selected Tasks	Attaches to the selected tasks. This has no effect on non-task entries.
Detach from Selected Tasks	Detaches from the selected tasks.
Halt Selected Tasks	Halts the selected tasks.
Continue Selected Tasks	Resumes the selected tasks.
Step Selected Tasks	Single-steps the selected tasks.
Detach from Tasks in Current Group	Detaches from all attached tasks in the current task group.
Halt Tasks in Current Group	Halts all tasks in the current task group.
Continue Tasks in Current Group	Resumes all tasks in the current task group.
Step Tasks in Current Group	Single-steps through all tasks in the current task group.
Halt System	Halts all tasks on the target system, thereby freezing the target. This option is not supported on all operating systems.
Run System	Restores the tasks on the target to the status they held before the system was halted.
Add Selected Tasks into Group	Adds the selected tasks into another task group.

Item	Meaning
Delete Selected Tasks from Group	Removes the selected tasks from the current task group.
Show AddressSpace IDs	Displays Address Space IDs in the second column of the Task Manager.
<i>Other entries</i>	Displays or hides the corresponding column.

**Note**

In the shortcut menu, the word `Task` is replaced with the corresponding terminology used by the underlying RTOS (such as `Process` or `Thread`).

Task Group Configuration File

The definitions for task groups are persistent across debugging sessions. These definitions are stored in a configuration file. When launching MULTI from the command line, use the **-tv** option to specify which configuration file is to be used in the debugging session. For example:

```
-tv mytaskview.tvc
```

If no file extension is given in the task view configuration filename, **.tvc** is appended to the filename. If no configuration file is specified from the command line, **taskview.tvc** from your local configuration directory is used.

For each group, MULTI keeps a task fingerprint for each task in the group. Whenever the group is to be displayed (when you select the corresponding tab in the Task Manager), MULTI uses this task fingerprint to determine which tasks to display. Considering different debugging environments, MULTI provides the following three criteria to match a task against a task fingerprint.

- **Task Id + Hierarchy** — When a task's identifier (a number) and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.
- **Task Name + Hierarchy** — When a task's name and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.

- **Task Id or Name + Hierarchy** — When a task's name or identifier and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.

See also the description of the **taskMatchCriteria** configuration option in “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

MULTI also provides a configuration option **deleteDeadTaskFromGroup** to tell the Task Manager to delete *dead* task fingerprints from the task group. When you display a task group by clicking the corresponding tab, if there is no live task for a task fingerprint, the task fingerprint is *dead*. For example, a group created in the last debugging session contains a fingerprint for `task1`, but `task1` is not on the task list at present because the task has not yet been created by the program in the current debugging session. If the configuration is on when you click the tab for the group, the Task Manager permanently removes the fingerprint for `task1` from the group. If this option is off, the fingerprint for `task1` is not removed and `task1` shows up when it is created. However, the fingerprints for zombied tasks are not cleaned up from the group. (For more information about the **deleteDeadTaskFromGroup** option, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.)



Note

The configuration file contains the information about group definitions and synchronous operations on tasks.

OS-Awareness in Run Mode

Two OS-aware tools allow you to browse OS information while debugging in run mode: the **OSA Explorer** and the OSA Object Viewer. The **OSA Explorer** is useful for browsing information about an entire operating system, while the OSA Object Viewer is useful for browsing information about an individual INTEGRITY object. The following sections briefly describe each tool.

The OSA Explorer

The **OSA Explorer** shows INTEGRITY operating system tasks and other objects on the target in run mode. You can launch an **OSA Explorer** anytime; however, if the system is not halted, doing so causes MULTI to halt the system. In this case, closing the **OSA Explorer** causes MULTI to resume system execution. For more information about **System** halts, see “Default Task Groups” on page 585.

To launch an **OSA Explorer** in run mode, do one of the following:

- In the Task Manager or Debugger, click the **OSA Explorer** button (.
- In the Task Manager or Debugger, select **View** → **OSA Explorer**.
- In the Debugger command pane, enter the **osaexplorer** command. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information about the **OSA Explorer**, see Chapter 26, “Freeze-Mode Debugging and OS-Awareness” on page 605 and “Object Structure Aware Debugging” in the *INTEGRITY Development Guide*.

The OSA Object Viewer

With the OSA Object Viewer, you can view INTEGRITY object attributes, inject messages into the kernel, and navigate between objects. The OSA Object Viewer also keeps a history of all the objects you have viewed. You can easily navigate this history. Because the OSA Object Viewer is typically used to display small chunks of information from the kernel, it is faster than the **OSA Explorer**.



Note

The OSA Object Viewer is only available if you are debugging a run-mode connection and using INTEGRITY version 10 or later.

To open the OSA Object Viewer, do one of the following:

- In the Debugger target list, select an object and click the **OSA Object Viewer** button (.

- In the Debugger command pane, enter the **osaview** command. For information about the **osaview** command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Debugger source pane, double-click an INTEGRITY object variable while the task is halted.

The OSA Object Viewer opens on an object summary, a list of attributes (under the **Attributes** tab), and a list of operations you can perform (under the **Operations** tab). Objects that are underlined in the OSA Object Viewer are linked to more detailed information. To view this information, click the underlined object. Similarly, clicking an underlined operation performs the operation.

By default, the OSA Object Viewer window is reused. However, if you freeze the window or click the **Reuse** button () so that it no longer appears to be pushed down, a new window appears the next time you open an OSA Object Viewer or the next time you click a linked (that is, underlined) object in the OSA Object Viewer.

For more information about the OSA Viewer, see “Object Structure Aware Debugging” in the *INTEGRITY Development Guide*.

Profiling All Tasks in Your System

If you are using INTEGRITY version 10 or later, MULTI allows you to view processor usage for all tasks in your system. The information is updated continually while the target is running, and it can be obtained without instrumenting the code: INTEGRITY periodically samples the program counter (PC) of running tasks, and the run-mode debug server delivers the resulting PC samples to MULTI.

The continually updated profiling information is displayed both in the **Profile** window's standard calls and source reports (see “Standard Calls Report” on page 365 and “Source Report” on page 371) and in a target list column labeled **CPU %**. This column displays the percentage of the CPU that each task and AddressSpace is currently using. The AddressSpace **CPU %** is simply the sum of the CPU percentages for all the tasks in that AddressSpace.

To begin profiling all tasks in your system, perform the following steps:

1. In the Debugger's target list, select the run-mode connection to your target.

2. Open the **Profile** window by selecting **View** → **Profile** or by entering the **profile** command. For information about the **Profile** window, see “The Profile Window” on page 361. For information about the **profile** command, see Chapter 12, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.



Tip

You can use the **ServerPollinterval** configuration option to control the interval between updates to displayed data. For more information about this option, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

To disable profiling, close the **Profile** window.

Chapter 26

Freeze-Mode Debugging and OS-Awareness

Contents

Starting MULTI for Freeze-Mode Debugging and OS-Awareness	607
The OSA Explorer	609
Debugging in Freeze Mode	613
Freeze-Mode and OSA Configuration File	632

When you debug a multitasking application, MULTI interacts with the target's operating system in freeze mode, in run mode, or in both freeze and run mode simultaneously. In freeze mode, the entire target system stops when a breakpoint is hit or a task is halted. In run mode, the operating system kernel continues to run as you halt and examine individual tasks. For information about run-mode debugging, see Chapter 25, “Run-Mode Debugging” on page 577. For information about debugging in run mode and freeze mode, see “Automatically Establishing Run-Mode Connections” on page 69.

During freeze-mode debugging, MULTI allows you to debug individual program tasks, and it provides OS-awareness for operating system objects, allowing you to examine detailed information about these objects. With freeze-mode debugging, you can:

- Display an **OSA Explorer** showing operating system tasks and other objects on the target (if any).
- Display an entry in the target list for each task, even if that task is suspended or otherwise not running.
- Set task-specific breakpoints that will stop the running process only when the chosen task is currently executing.
- Display call stacks and stack (local) variables for any task that can be debugged.
- Perform task-specific single-stepping.



Note

The **OSA Explorer** is also available in some run-mode debugging environments. For more information, see “The OSA Explorer” on page 602.

Currently, freeze-mode debugging is available for these target environments:

- INTEGRITY
- velOSity
- u-velOSity
- ThreadX

If your operating system is not on the list, contact your Green Hills sales representative.

**Note**

In this chapter, *task* is used as a general term to describe the target operating system's unit of program execution. Specific terminology varies according to the target's operating system (for example, a task may also be called a *thread*). When the term *process* is used in this chapter, it usually refers to the master process. For more information, see “Master Debugger Mode” on page 614.

Starting MULTI for Freeze-Mode Debugging and OS-Awareness

MULTI identifies the operating system in use on your target and then configures itself to debug that operating system. INTEGRITY and velOSity are identified by OSA debugging information in the operating system kernel. MULTI identifies other operating systems by examining programs for the following OS-specific symbols:

- u-velOSity — `uv_task_create` or `_uv_task_create`
- ThreadX — `_tx_thread_created_ptr`

Under special circumstances, an OSA integration package may be provided to allow freeze-mode OS-aware debugging. In that case, start MULTI with the following command line option:

`-osa osa_name [#cfg=configuration_file] [#lib=library_name] [#log=log_file]`

where:

- *osa_name* is case-insensitive.
- *configuration_file* is the configuration file for freeze-mode debugging and OS-awareness (see “Freeze-Mode and OSA Configuration File” on page 632). If you do not specify a full path to the configuration file, MULTI first searches for it in:
 - Windows 7/Vista — **`user_dir\AppData\Roaming\GHS\os_aware`**
 - Windows XP — **`user_dir\Application Data\GHS\os_aware`**
 - Linux/Solaris — **`user_dir/.ghs/os_aware`**

and then looks for it in the MULTI IDE installation directory. If you do not specify a configuration file, **`osa_name.osa`** is used by default.

- *library_name* is the library containing the OSA integration package (see “The OSA Integration Module Name” on page 635). The library may be a DLL file (Windows) or an SO shared library (Linux/Solaris). If you do not specify a library, *osa_name.dll* (Windows) or *osa_name.so* (Linux/Solaris) is used by default.

You can place the integration module in any location, and use an absolute or relative path to locate it from the Debugger's command line or from the configuration file. If the integration module is specified with only a base name, MULTI first searches for it in the MULTI IDE installation directory, and then in a way defined by the host machine.

- *log_file* is the name of the file in which you want to log the interaction between MULTI and the OSA integration package.

For example, to debug a program **my_program** in freeze mode with an OSA integration package **rtos**, which uses a DLL called **rtos.dll**, start MULTI with the following command line option:

```
multi -osa rtos#lib=rtos_lib my_program
```

If you are debugging multiple programs and you want to use OS-aware debugging features for some of the programs, but not for others, set the option `RequestOsaPackage` to `On`. With this option, you do not have to specify the **-osa** option on the command line. Instead, MULTI prompts you for the OS-awareness package name when you are debugging in freeze mode. MULTI resolves the name of the configuration file and of the library as described above. See also “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.



Tip

You can also use the **osasetup** command to specify an OSA integration package. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

The OSA Explorer

The **OSA Explorer** shows operating system tasks and other objects on the target (if any). The following sections explain how to display and customize the **OSA Explorer** and describe the operations that can be performed in the kernel object list.

Displaying an OSA Explorer

When the target is halted and you are debugging a multitasking application with OS-awareness, you can launch an **OSA Explorer** by doing one of the following:

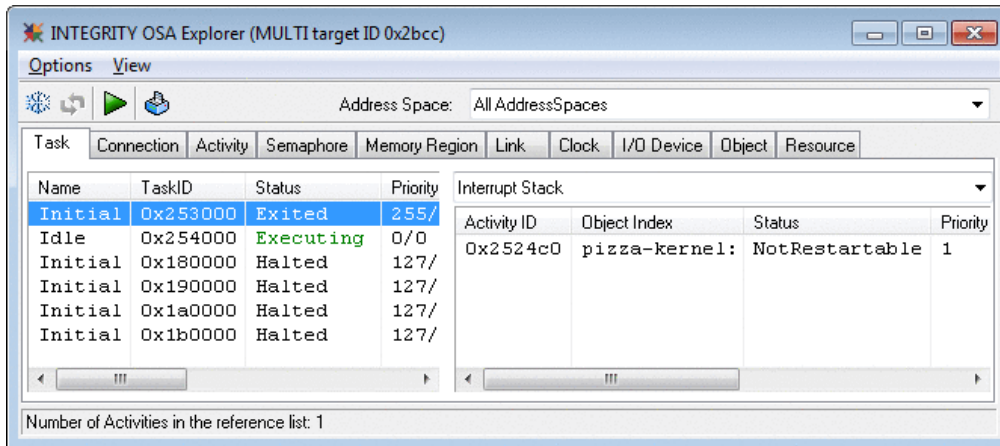
- In the Debugger, select **View → OSA Explorer**.
- In the Debugger command pane, enter the **osaexplorer** command. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

If the **View → OSA Explorer** menu item is dimmed out or the **osaexplorer** command prints an error such as:

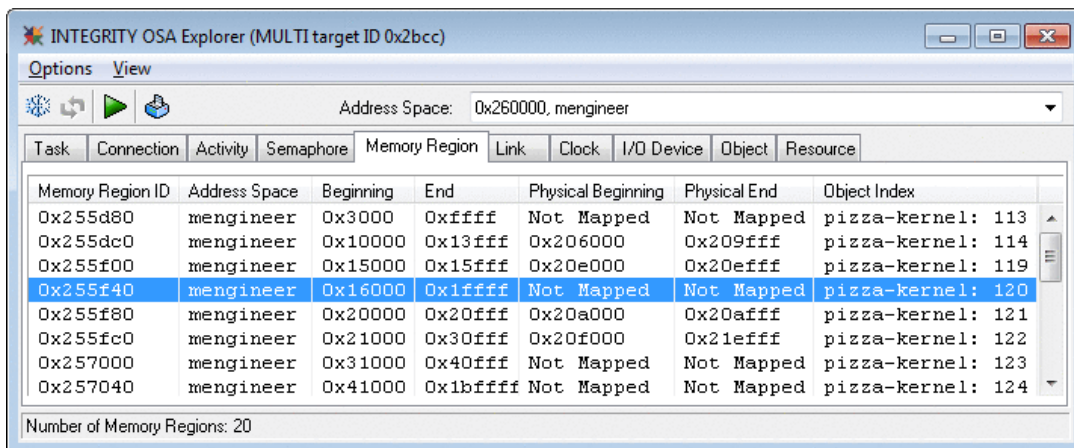
```
osaexplorer: not supported in the current environment
```

then MULTI does not recognize the target operating system as one for which MULTI can perform freeze-mode OS-aware debugging. For more information, see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 607.

The following is an **OSA Explorer** showing a task list of all AddressSpaces for the INTEGRITY operating system.



The following is an **OSA Explorer** showing the **MemoryRegion** (a type of kernel object) list for the **AddressSpace** `mengineer` on the **INTEGRITY** operating system.



Each type of object that can be explored is represented as a tab in the **OSA Explorer**. If an object type has no reference object, the **OSA Explorer** contains one object list pane to show the instances of this type of object. If an object type has a reference object, two object list panes are displayed:

- The master pane on the left side of the **OSA Explorer**.
- The reference pane on the right side of the **OSA Explorer**.

Whenever an object is selected in the master pane, the instances of the corresponding reference object are shown in the reference pane.

Because the object shown in the master pane could have references to more than one type of object, the drop-down list at the top of the reference pane indicates the

object type being shown in the reference pane. If you select an object type to be shown in the reference pane, your change is persistent for the duration of the debugging session.

For information about INTEGRITY objects available in the **OSA Explorer**, see the *INTEGRITY Development Guide*.

The OSA Explorer GUI Reference

The following table describes the menu items and toolbar buttons available in the **OSA Explorer**.

Menu Item	Button Equivalent	Meaning
Options → Print		Prints the object list(s) currently displayed in the OSA Explorer . If two object lists (the master list and the reference list) are displayed in the OSA Explorer , the print dialog box appears twice.
Options → Help	no equivalent button	Displays online help information about freeze-mode debugging and the OSA Explorer .
Options → Close		Closes the OSA Explorer . Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons . To access this option, select Config → Options → General Tab .
View → Freeze Object List		Freezes or unfreezes the automatic refreshing of the OSA Explorer . When the OSA Explorer is frozen, a message appears in the status bar at the bottom of the window, the contents of the window are preserved, and you cannot switch to another tab. The window is not updated again until it is unfrozen.
View → Manually Refresh Object List		Refreshes the OSA Explorer even if it is frozen. This button is not available if the target is halted.
View → Toggle Target Status		Runs/stops the underlying process on the target if you are debugging in freeze mode, or resumes/halts the underlying RTOS on the target if you are debugging in run mode.

Customizing the Object List

The **OSA Explorer** for each OS has a default configuration that indicates which columns to display and how to display them for each type of object. The object list pane behaves like other multiple-column panes in MULTI. You can customize it as follows:

- To sort a list based on a column's contents, click that column's header. The sorting toggles between ascending and descending order.
- To move a column, drag that column's header left or right.
- To change the width of a column, drag the column boundary.
- To show or hide a column, right-click the pane to open a shortcut menu, and select or clear the corresponding menu items.

Changes that you make to the **OSA Explorer** itself (such as its size and position) and to each object are maintained across debugging sessions.

Task information is displayed in the (foreground) text color. Entries in the object list pane have the same background color as the corresponding Debugger window (only applicable if you have opened multiple Debugger windows).

Object List Operations

When you double-click a task in the object list pane, the selected task loads in the current Debugger window. When you right-click an object in the object list pane, a shortcut menu appears.

The following table describes the menu items that may appear in the **OSA Explorer** shortcut menu. The actual menu contents vary depending on the RTOS and object type.

Menu Item	Meaning
<i>Column Name</i>	Toggles between showing or hiding the indicated column.
Freeze Object List	Freezes or unfreezes the automatic refreshing of the OSA Explorer . When the OSA Explorer is frozen, a message appears in the status bar at the bottom of the window, and the contents of the window are preserved. The window is not updated again until it is unfrozen.

Menu Item	Meaning
View Object Information	Displays the selected object in a Data Explorer.
Debug Task	Displays the selected task in the Debugger window. See “Debugging in Freeze Mode” on page 613 and “Shortcut Menu Entry Definitions” on page 638.
Inject Message	Injects a message to the underlying RTOS to change its behavior. For example, on INTEGRITY, you can dynamically inject a message into the system to: • Take a <code>Semaphore</code> object on behalf of a task • Release a <code>Semaphore</code> to break a dead lock • Send a message to a <code>Connection</code> object This option is only supported on some RTOSes.

To print the contents of the object list pane, select **Options** → **Print**.

Debugging in Freeze Mode

The following sections explain how to debug in freeze mode:

- “Master Debugger Mode” on page 614
- “Task Debugger Mode” on page 615
- “Task-Specific Single-Stepping” on page 616
- “Working with Freeze-Mode Breakpoints” on page 618
- “Program I/O” on page 620
- “Multi-Core Debugging” on page 620

If the currently executing task resides in the kernel address space, you can either use Master Debugger mode or Task Debugger mode to view information specific to the task. If you want to debug a task that is not currently executing or does not reside in the kernel address space, you must use Task Debugger mode to view task-specific information.

**Note**

The configuration option **osaTaskAutoAttachLimit** allows you to specify the maximum number of OSA tasks that are displayed in the target list (the default is 32). The tasks are displayed consecutively until this limit is reached. For example, if this option is set to 32 and there are 33 tasks, the 33rd task is not displayed in the target list. To manually display additional tasks in the target list, double-click the tasks in the **OSA Explorer Task** pane. For more information about the configuration option **osaTaskAutoAttachLimit**, see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

**Note**

The configuration option **osaSwitchToUserTaskAutomatically** controls whether the Debugger automatically displays the currently executing user-mode task when the target is stopped while executing user-mode tasks. This is in addition to automatically displaying all of the tasks in the system as controlled by the **osaTaskAutoAttachLimit** configuration option. For more information about these configuration options, see “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

For information about debugging with TimeMachine, see “Using TimeMachine with OS Tasks” on page 418 and “OSA Tasks and the Master Process in TimeMachine” on page 419.

Master Debugger Mode

During freeze-mode debugging, selecting the master process in the target list enables Master Debugger mode for the target. The master process is displayed in the target list after the CPU node of a freeze-mode connection and before the OSA address spaces whose names are prefixed with **OSA**. When the master process is selected, the kernel's source code is displayed in the Debugger window. Double-clicking the process displays the kernel source code in a new Debugger window.

The status of the master process (displayed in the **Status** column) corresponds to the status of a normal process or provides specific information about the mode in

which the CPU may be stopped. For example, the status may be **Running**, **Stopped**, **Stopped in Kernel Mode**, or **Stopped in User Mode**. All operations, such as memory access, register access, etc., are performed in the target's current status. If the master process is selected in the target list while the target is stopped in user mode (that is, in a non-kernel task), the PC pointer becomes gray.

The following operations are only available in Master Debugger mode:

- Reloading or restarting the program.
- Killing the process or detaching from it.
- Setting normal breakpoints (as opposed to task-specific and any-task breakpoints). All breakpoints set in Master Debugger mode are normal breakpoints.
- Setting hardware breakpoints.

When you reload or restart the program or detach from or kill the process, MULTI removes all OSA task entries from the target list and closes the **OSA Explorer** and any Debugger windows that you have opened on an OSA task.

Task Debugger Mode

To view source code for a task, do one of the following:

- In the **OSA Explorer Task** pane, double-click a task.
- In the **OSA Explorer Task** pane, right-click a task and choose **Debug Task**.
- In the Debugger's target list, select a task. Tasks are located under an address space node whose name begins with **OSA**.
- In the Debugger command pane, enter the command **osatask**. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

To open a separate Debugger window on a task, double-click the task in the target list. MULTI does not open multiple Debugger windows for the same task.

The following list describes behavior that is specific to Task Debugger mode:

- Call stacks, local variables, and register displays are specific to the task.

- The title bar displays the task ID (a number that uniquely identifies the task) and the task name, if the OS maintains a name for that task (see “The Object Attribute Definition” on page 637).
- When the RTOS's master process is halted, the target list's **Status** column displays the status of tasks in the RTOS. When the RTOS's master process is running, the **Status** column displays **<OS Running>**.
- Except for the  and  buttons, buttons related to execution (such as , , ) and their corresponding commands are only available when the master process is halted and the task displayed is the currently executing task.
- Single-stepping and breakpoints behave in a manner that is more appropriate to task-aware debugging. For more information, see “Task-Specific Single-Stepping” on page 616 and “Working with Freeze-Mode Breakpoints” on page 618.
- Some of the MULTI commands that affect the global debugging state are not available in Task Debugger mode. If you try to run one of these commands, MULTI displays the following error message:

```
This command cannot be run with an OSA Task selected.  
Select the target list entry corresponding to the connection  
or executable first.
```

Task-Specific Single-Stepping

When you single-step a process in Task Debugger mode, MULTI avoids stopping the process when another task is running. This makes debugging shared code more straightforward by allowing you to step through one task at a time.

MULTI generally single-steps through program source by setting temporary breakpoints and running until one of those breakpoints is hit. Sometimes a context switch takes place between the time the process is run and the time one of the temporary breakpoints is hit. (It is possible for the temporary breakpoint to be hit from another task's context.) In these cases, MULTI lets the target process run until one of the following occurs.

- The temporary breakpoint is hit while the original task is running.
- Another breakpoint is hit.

- The running process is halted.

This feature operates transparently, and no special action is required, aside from performing all task-specific single-steps in Task Debugger mode. In Master Debugger mode, single-stepping may occasionally step over context switches.

The following example helps to illustrate when task-specific single-stepping can be useful. This example uses u-velOSity's GH-API, but most other operating systems can perform similar operations.

```
335 1    void shared_func(char *s) {
336 2        printf("%s called shared_func\n",s);
-> 337 3        gh_task_yield();
338 4        printf("%s end of call to shared_func\n",s);
339 5    }
340
341 1    void task_1_entry(GH_ADDRESS input) {
342 2        for(;;)
343 3            shared_func("task_1");
344 4    }
345
346 1    void task_2_entry(GH_ADDRESS input) {
347 2        for(;;)
348 3            shared_func("task_2");
349 4    }
```

In this process, two tasks are running at the same priority without time slicing. Both tasks loop, calling `shared_func()`. The first task, `task_1`, has `task_1_entry()` as its entry point. The second task, `task_2`, has `task_2_entry()` as its entry point. Within `shared_func()`, each task calls `gh_task_yield()`, a GH-API call that allows other tasks at the same priority to run. In this case, `task_1` yields to `task_2` and then `task_2` yields to `task_1`. So in the steady state, one task is always suspended within a call to `gh_task_yield()` while the other task is running.

At the moment the process is stopped, `task_1` is the current task and is stopped on line 337, where it is just about to call `gh_task_yield()`. If you now press the  button from within `task_1`'s Debugger window, the following happens:

1. MULTI sets a temporary breakpoint on line 338 and restarts the process.
2. The call to `gh_task_yield()` causes `task_1` to suspend and `task_2` to run again.

3. `Task_2` returns from its previously suspended call to `gh_task_yield()` and the process hits the temporary breakpoint.
4. MULTI notices that `task_1` is not the current task and restarts the process again.
5. `Task_2` loops and calls `gh_task_yield()`, which causes `task_2` to suspend and `task_1` to run again.
6. `Task_1` returns from its call to `gh_task_yield()` and the process again hits the temporary breakpoint.
7. MULTI notices that `task_1` is the current task and stops the process.

Task-specific single-stepping makes it easy for you to concentrate on debugging the task you are interested in.

Working with Freeze-Mode Breakpoints

When performing freeze-mode debugging on multitasking applications, you can set task-specific breakpoints, any-task breakpoints, and normal breakpoints. Each type of breakpoint is described below:

-  — Indicates a task-specific breakpoint. A task-specific breakpoint can be set in an OSA task and can only be hit by the task it was set on. A task-specific breakpoint is implemented as a conditional breakpoint: the current task ID when the breakpoint is hit must match the task ID associated with the breakpoint. Otherwise, MULTI simply continues running the process.

If the breakpoint icon is red, the breakpoint can only be hit by the task currently selected in the target list (that is, the breakpoint was set on the current task). If the breakpoint icon is gray, the breakpoint can only be hit by a task other than the one currently selected in the target list (that is, the breakpoint was *not* set on the current task). Task-specific breakpoints in the kernel address space are gray in the OSA master process.

In the **Breakpoints** window, task-specific breakpoints are listed with a command parameter. Otherwise they are displayed like normal breakpoints.

-  — Indicates an any-task breakpoint. An any-task breakpoint can be set in an OSA task and can be hit by any task in the same address space.

To set an any-task breakpoint, select a task in the target list and hold the **Shift** key as you click a breakdot (•) in the Debugger window. Alternatively, you can use the **sb at** command.

For example:

```
sb at function#5
```

sets an any-task breakpoint at line 5 of the function *function*. For information about the **sb** command, see Chapter 3, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

-  and all other breakpoint icons — Represent normal breakpoints. A normal breakpoint can be set in the OSA master process and can be hit by OSA tasks that can access the breakpoint's address.

These breakpoints are similar to any-task breakpoints set in OSA tasks, but they are not the same. You cannot set an any-task breakpoint in the OSA master process. The freeze-mode debug server has no knowledge of RTOS concepts such as any-task breakpoints; it treats the program you are debugging as a stand-alone program. MULTI uses the OSA package to simulate an RTOS scenario (RTOS concepts are only available to MULTI). MULTI translates the semantics of RTOS operations into the normal operations supported by the debug server.

You can only set hardware breakpoints in the OSA master process. These breakpoints are not RTOS aware, so they will affect all tasks.

For information about breakpoints set in OSA tasks or the OSA master process while TimeMachine is enabled, see “OSA Breakpoints in TimeMachine” on page 420.



Note

Only one breakpoint can be set at a particular address. For example, if a task-specific breakpoint is set at the location `shared_func#2()`, the breakpoint must be removed before another breakpoint (of any type) can be set at the same address. If you do not remove the breakpoint, MULTI does so for you. On INTEGRITY, this limitation applies to each AddressSpace (that is, multiple breakpoints can be set at the same address as long as each breakpoint is set in a different AddressSpace).

Program I/O

In a freeze-mode environment, program I/O is usually redirected to the MULTI Debugger and immediately displayed in its **I/O** and **Cmd** panes. Operating systems such as INTEGRITY, however, capture and independently handle program I/O. In this case, I/O may not be immediately visible.

When an INTEGRITY program calls an I/O function such as `fprintf()` to print a message, input is neither immediately sent nor output immediately received. Instead, INTEGRITY translates the request into a console operation and queues it in the system. Even a flush operation, such as `fflush()`, is translated and queued in the same way. When the kernel is running and handling these operations, I/O is printed to the console approximately every second.

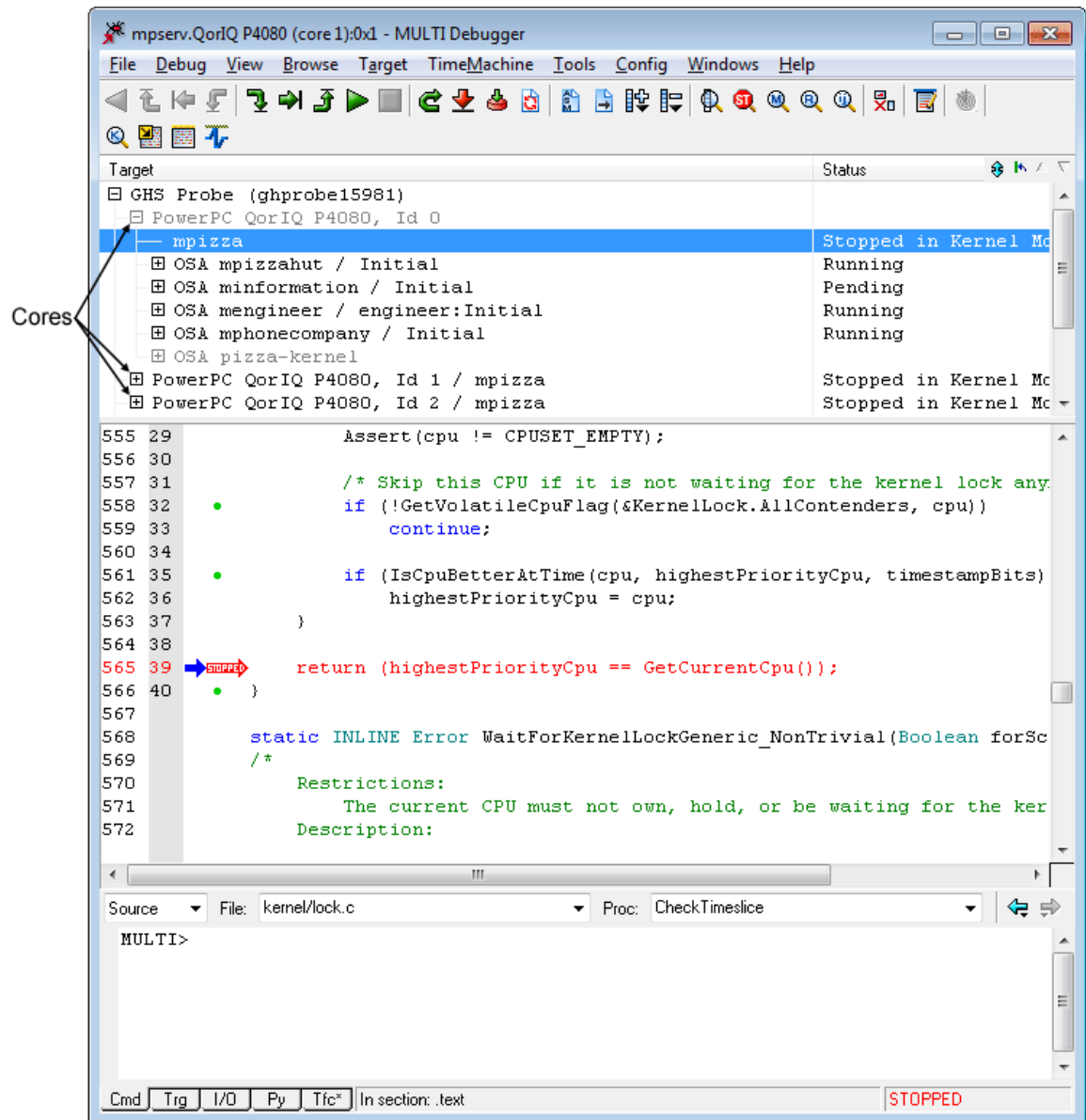
When you single-step on a task in freeze mode, you cannot immediately see the output of I/O functions because the kernel is halted. The output is displayed in the console only after you start the target running and the kernel has a chance to flush out queued console operations.

Multi-Core Debugging

A *multi-core target* is a target containing multiple processors, or *cores*, that can each be debugged independently. This section details techniques and limitations specific to debugging multi-core targets.

Multiple Cores in the Target List

When you connect to a multi-core target, MULTI displays the cores as separate entries in the target list, and it assigns each core a number, beginning with zero.



Multiple Cores in the Task Manager

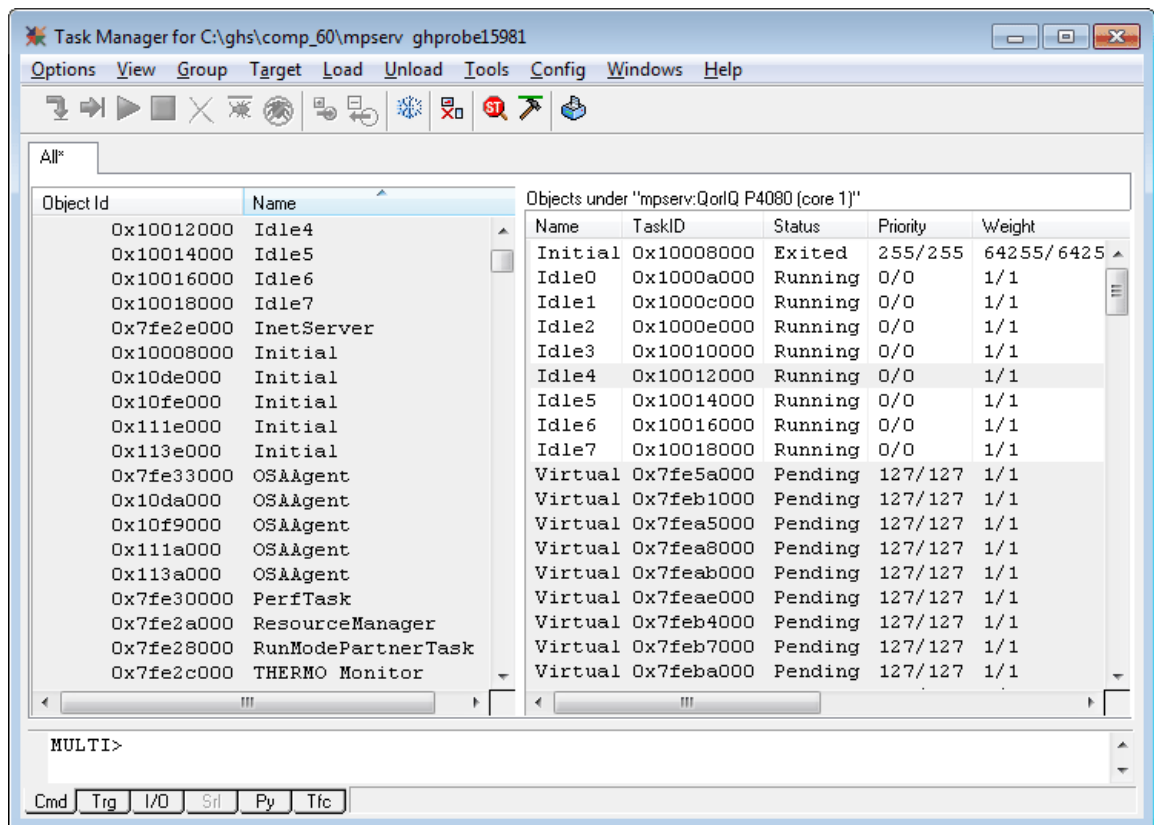
When you connect to a multi-core target, the Task Manager displays two separate panes:

- Master pane — Shows each core on the target and basic information (**Object Id** and **Name**) about the tasks on each core.

- Reference pane — Shows more information about the object selected in the master pane:
 - If a debug server is selected — Shows detailed information about the cores being controlled by the debug server.
 - If a core is selected — Shows additional information about the tasks running on the core.
 - If a task is selected — Shows all of the tasks belonging to the same core as the selected task, and briefly highlights the selected task.

Cores cannot be sorted in either pane.

The following Task Manager shows an application that is connected via the **mpserv** debug server to a multi-core target running the INTEGRITY operating system on each QorIQ core:



Using Hook Commands in Multi-Core Setup Scripts

Before loading a program onto a hardware target, MULTI typically runs the board setup script (if any) associated with the Connection Method. The board setup script may set up initial register values, memory controller configurations, etc. Board setup scripts for multi-core targets can take advantage of the hook commands documented in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

For example, suppose you want to run several MULTI commands on core 1 after resetting your target. You might add a command such as the following to your setup script:

```
addhook -core 1 -after reset {commands}
```

If you are using the hook commands, you may want your setup script to run once upon connection rather than before every download. For more information, see “Early MULTI Board Setup Scripts with Debugger Hooks” on page 101.

Multi-Core Configuration File

The MULTI Debugger can use information you provide in a multi-core configuration file (*.ghsmc) to automatically download programs to your multi-core target and load symbol files to the Debugger. This can simplify target setup considerably.

To start the MULTI Debugger on a multi-core configuration file, run the following command from the command line:

```
multi multi-core_config_file.ghsmc
```

An example configuration file for an RH850 multi-core environment follows:

```
version = 1

core {
  0 {
    download = {"/core0", "/shared.so"}
  }
  1 {
    download = {"/core1"}
    symbol = {"/shared.so"}
  }
}
```

```
    }
    2 {
        download = {"/core2"}
        symbol = {"/shared.so"}
    }
    3 {
        download = {"/core3"}
        symbol = {"/shared.so"}
    }
}
```

Here is another example, this time for a QorIQ P4080 SMP environment:

```
version = 1

core {
    0 {
        download = {"mpizza"}
    }
    "1-7" {
        symbol = {"mpizza"}
    }
}
```

In both examples, the following elements appear:

- The `version` field specifies the version of the configuration file. Only version 1 is supported at present.
- Information about cores is defined in the `core` block.
- The name for each block nested under `core` may be:
 - A core ID (an integer). You can find core IDs in the Debugger's target list. See “Multiple Cores in the Target List” on page 620.
 - A quoted string with an inclusive core ID range. For example, "1-7" indicates cores with IDs 1, 2, ..., 7.
- Each of the nested blocks may contain one or more fields in the format:

```
variable_name = value
```

- The `download` field lists the program(s) to be loaded to the target over the core; corresponding symbol information is used by the Debugger to debug the program(s). Begin the list with the program that must run first

(the main program). Enclose the comma-separated list in curly braces and quote each list item, as shown in the first example.

Note that a program may be loaded to the target over one core, but be accessible to other cores.

- The `symbol` field lists the symbol file(s) to be used on the core. Begin the list with the symbol file that should run first (the main symbol file). Format the list as described above.

Collective Treatment of Multi-Core Programs

Unlike an application on a single core, a multi-core application contains a set of programs for a set of cores. When the following operations are applied to a program on one core, the Debugger treats the programs as a single unit, automatically applying the operation to the programs on all cores:

- Loading a multi-core program in the Debugger
- Removing a multi-core program from the Debugger
- Preparing the target
- Binding/unbinding a multi-core program and a connection/CPU
- Detaching a multi-core program from a connection
- Killing a running program

The following commands, some of which correspond to one of the preceding operations, also affect all cores:

- **change_binding**
- **connect**
- **prepare_target**
- **k**
- **quit entry**
- **debug**
- **new**
- **detach**

Corresponding toolbar buttons and menu items (including right-click menu items) have the same effect as the commands.

Preparing Multiple Cores to Run a Single Executable

If multiple cores on your target share memory, or if one core controls the reset and/or run-control capabilities of others, you might choose to build the code for all the cores into a single executable file. Before loading this executable onto your target, you must associate the executable with every core. One method for doing so follows:

1. Select the executable in the target list, and click the **Connect** button ()
2. Using the **Connection Chooser** that appears, connect to your multi-core target.
3. In the **Use Which Connection/CPU?** dialog box that appears, select any one of the cores, and click **OK**.
4. In the Debugger command pane, issue the **new** command with no arguments. For information about the **new** command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
5. If the **Use Which Connection/CPU?** dialog box appears again, select any core. If the dialog box does not appear, MULTI automatically associated the executable with the only remaining core.
6. Repeat steps 4 and 5 until the executable has been associated with every core on the target.

For information about the **Use Which Connection/CPU?** dialog box, see “Associating Your Executable with a Connection” on page 105.

After associating the executable with every core on the target, you can load the executable onto the target:

1. In the target list, select the executable listed after the core that is supposed to run first.
2. In the Debugger command pane, issue the **prepare_target-load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

3. For each remaining core, select the executable listed after the core, and enter **prepare_target -verify=none** in the Debugger command pane.

Alternatively, you can use the **prepareAllCores** option to configure MULTI to automatically and silently run **prepare_target -verify=none** on all secondary cores. For more information, see the description of **prepareAllCores** in “Other Debugger Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Preparing Multiple Cores to Run Separate Executables

If the cores on your target do not share memory and if each can be independently controlled, or if they are otherwise intended to execute programs independently, you might choose to build the code for each core into its own separate executable file. Before loading the executables onto your target, you must associate each executable with the core that is supposed to run it. One method for doing so follows:

1. Select one of the executables in the target list, and click the **Connect** button .
2. Using the **Connection Chooser** that appears, connect to your multi-core target.
3. In the **Use Which Connection/CPU?** dialog box that appears, select the core that you want to run the selected executable, and click **OK**.
4. In the Debugger command pane, enter **new program_name**, where *program_name* is the name of another executable. For information about the **new** command, see Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
5. If the **Use Which Connection/CPU?** dialog box appears again, select the core that you want to run the second executable. If the dialog box does not appear, MULTI automatically associated the executable with the only remaining core.
6. Repeat steps 4 and 5 until an executable has been associated with every core on the target.

For information about the **Use Which Connection/CPU?** dialog box, see “Associating Your Executable with a Connection” on page 105.

After associating each executable with the core that is supposed to run it, you must separately load each program onto the corresponding core:

1. Select an executable in the target list.
2. In the Debugger command pane, issue the **prepare_target -load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
3. Select a different executable in the target list.
4. If your board setup script only affects the current core when it is run, and if it does not reinitialize any shared components on the board (such as memory) — Issue the **prepare_target -load** command again.

If your board setup script affects multiple cores when it is run, or if it reinitializes shared components on the board (such as memory) — Issue the **load -nosetup** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

5. Repeat steps 3 and 4 for each remaining executable.

Synchronous Run Control

Some targets support (or require) *synchronous*, or simultaneous, run-control operations on all cores. To initiate synchronous run-control operations, you must do one of the following:

- Use the options in the Task Manager's **Group** menu (for example, **Continue Tasks in Current Group**).
- Open the Task Manager, and then issue the **groupaction** command with appropriate arguments:

groupaction -r|-h|-s @All

where **-r**, **-h**, or **-s** specifies a run, halt, or single-step operation, respectively, and **@All** indicates that the designated operation be synchronously performed on all cores on the target. In a freeze-mode environment, synchronous run control is not supported on groups other than **All**.



Note

In freeze mode, both of these operations operate on cores, not on the tasks running on the cores.



Note

Selecting multiple tasks in the target list and then issuing a **MULTI** command such as **c** or **halt** does *not* initiate synchronous run-control operations.

If you are using a Green Hills Probe, see the *Green Hills Debug Probes User's Guide* for any additional configuration settings that apply to your target.

Using Trace Tools on a Multi-Core Target

Some multi-core targets are capable of tracing more than one core by multiplexing the trace data from many cores into a single trace stream. With supported targets, **MULTI** can use this feature to enable TimeMachine for multiple cores.

Bugs involving complex interactions among multiple processors are notoriously difficult to debug and are often also difficult to reproduce. With multi-core trace, you only need to reproduce the problem once to obtain a complete log of every

instruction executed on all cores. You can use the TimeMachine tools to analyze this data and figure out what went wrong.

When you trace a multi-core target, the trace controls for all of the cores are tied together. However, MULTI demultiplexes the trace data and provides a separate instance of each trace tool for each core. For example, you can open a separate PathAnalyzer window for each core and view separate path analysis of each core, but the **Enable Trace** (🟢) and **Retrieve Trace** (📄) buttons in all the windows are tied together. Clicking the **Enable Trace** button (🟢) in any of the windows enables trace collection from the target and causes the **Enable Trace** button (🟢) to toggle in the other windows as well.

By default, MULTI does not synchronize selections in the trace tools across cores. To enable multi-core trace synchronization, use the **trace sync on** command (see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book). If the trace data is timestamped, MULTI uses the timestamps to select instructions that were executing at approximately the same time on all traced cores each time an instruction is selected in a trace tool for any of the cores. If the trace data does not include timestamps, multi-core trace synchronization is not supported.

When you use TimeMachine on a multi-core trace target, the TimeMachine instances are independent, but they are synchronized if multi-core trace synchronization is enabled. For example, if you have a breakpoint set in the TimeMachine Debugger for core 0 and you run backwards in the TimeMachine Debugger for core 1, the breakpoint on core 0 is not hit. However, when the TimeMachine Debugger for core 1 stops, the TimeMachine Debugger for core 0 is synchronized to approximately the same point in time where core 1 stopped. It is possible to simultaneously run the TimeMachine Debugger for each core, but the result is unpredictable.



Note

The compact trace encodings required to trace multiple cores in the available trace bandwidth typically do not allow for timestamps on each instruction. In some cases, there may be as few as one timestamp every few thousand instructions. Therefore, it is not possible for MULTI to pinpoint exactly which instruction was executing on each core at a specific time, but its approximation is close.

Multi-core trace requires much more trace bandwidth than single-core trace. On some targets, it is not possible to capture complete trace of multiple cores. If you

attempt to trace too many cores at once, on-chip trace buffers overflow and gaps occur in the trace data (see “Dealing with Incomplete Trace Data” on page 411). To avoid this, you may want to trace a subset of cores. With most targets, you can control which cores are traced via the **Trace Options** window. Click the **Target Specific Options** button located on the **Collection** tab, or click the target-specific tab (only one or the other will appear). For more information, see the documentation about target-specific trace options in the *Green Hills Debug Probes User's Guide* or, if you are using a V850 target, the documentation about V850 trace options in the *MULTI: Configuring Connections* book.

Multi-Core Trace Notes

- Synchronous run control is very important when you are tracing multiple cores. If one core hits a breakpoint and the other cores continue running, they quickly fill up the trace buffer and push the data leading up to the breakpoint out of the buffer. Therefore if you think that you may want to look at the trace data leading up to a point where a core halted, you must halt all cores simultaneously or nearly simultaneously.
- Trace is not supported with INTEGRITY SMP.
- When a program is loaded onto any core, unretrieved trace data is cleared. Since the trace buffer is shared by all cores, unretrieved trace data for all cores is discarded.

Limitations

- If multiple cores on your target share memory and a single executable, then breakpoints, command line procedure calls, and host I/O system calls are only supported on the core that you initially loaded the program onto. Many advanced debugging features such as coverage, call count, and call graph profiling make use of breakpoints, command line procedure calls, and host I/O and are therefore also only supported on the first core.
- On targets that require synchronous run control, conditional breakpoints and breakpoints that resume the target are not supported.
- If one core holds another core in a reset state on your target, your debugging interface (for example, Green Hills Probe) may not be able to manipulate the latter core until the former brings it out of reset. In this case, the Debugger may

show the core held in reset as **Running**, even though it is not actually executing any code. You may not be able to halt a core that is being held in reset.

Freeze-Mode and OSA Configuration File

The configuration file for freeze-mode debugging and OS-awareness provides the Debugger with information about objects to be explored. For example, the file may:

- Assign identification numbers to each object and its attributes.
- Specify the relationships between objects.
- Define how the attributes of an object are to be displayed in the **OSA Explorer**.

The name of the configuration file consists of the lowercase name of the corresponding operating system or of your OS-awareness package. The filename ends with a **.osa** extension. It is stored in your personal configuration directory:

- Windows 7/Vista — ***user_dir\AppData\Roaming\GHS\os_aware***
- Windows XP — ***user_dir\Application Data\GHS\os_aware***
- Linux/Solaris — ***user_dir/.ghs/os_aware***

or in the MULTI IDE installation directory:

- Windows — ***ide_install_dir\defaults\os_aware***
- Linux/Solaris — ***ide_install_dir/defaults/os_aware***

MULTI searches for the configuration file in your personal configuration directory first, and then looks for it in the MULTI IDE installation directory.

In order to make the configuration file more readable, comment lines are supported. If the first two non-space characters are forward slashes (“//”), then the line is considered to be a comment.



Note

Two consecutive forward slashes (“//”) in the middle of a line is not considered to be a comment indicator as it would be in C++ code.

Each line in the configuration file can be a comment line, an empty line, or a line representing a configuration element. A backslash (“\”) at the end of a line combines

the next line with it, so multiple lines can be combined together to represent a single configuration element.

Even though MULTI recognizes the reserved words in a case-insensitive way, we recommend keeping all reserved words in uppercase for readability purposes as we do in the configuration files for the built-in OS integrations.

All syntax elements of the configuration file are separated by a colon (':'). If a string provided by the OSA integration provider contains a colon (':'), the string should be enclosed in double quotes.

The design of MULTI's OSA integration mechanism totally separates the GUI representation from program logic. Each object, attribute of an object and reference of an object is assigned a unique non-negative number. Negative numbers are reserved for special purposes. In the interaction between MULTI and the OSA integration module, only the identifier numbers are used. This makes it very easy for you to customize the strings displayed in an **OSA Explorer**, including internationalization.

The following subsections specify the syntax of the elements in a configuration file.

General Settings

The OSA Integration Version

Format:

```
OSA_CONFIG:OSA_VERSION:version_number
```

version_number is an integer. The current version number is 2.

Terminology for Tasks

Format:

```
OSA_CONFIG:OSA_TASK_ALIAS:terminology_for_task
```

The *terminology_for_task* specified here will be used at some places in the **OSA Explorer**. If the setting is absent in the configuration file, *task* will be used as the default.

When a task name needs to be used in other places of the configuration file, use *OSA_TASK* instead of *terminology_for_task*.

The OSA Explorer Title

Format:

```
OSA_CONFIG:OSA_EXPLORER_TITLE:osa_explorer_title
```

The *osa_explorer_title* argument specifies the title for the **OSA Explorer**. It is a string that can contain a number replacement (for target ID used by MULTI) in the syntax of `printf()` in the C library (`%d` or `0x%x`).

Initial OSA Commands

Format:

```
OSA_CONFIG:OSA_INIT_COMMAND:initial_osa_commands
```

When MULTI initializes the OSA integration module, *initial_osa_commands* are sent to it immediately.

Multiple initialization commands can be specified on multiple lines. When the module is initialized, MULTI sends the commands to the OSA integration module sequentially. If an initialization command fails, MULTI ignores the rest of the initialization commands, but continues as if the OSA integration module had been successfully initialized.

The Poll Interval

Format:

```
OSA_CONFIG:OSA_POLL_INTERVAL:interval_in_milliseconds
```

The OSA integration module can provide a function to be called periodically by MULTI, with an interval of *interval_in_milliseconds*. If the setting is absent, the default value (200 milliseconds) will be used.

The OSA Integration Module Name

Format:

```
OSA_CONFIG:OSA_MODULE_NAME:osa_integration_module_name
```

With the exception of the built-in integrations, most of the OSA integration modules are stored in a DLL file (Windows) or shared library (Linux/Solaris). This setting tells MULTI the integration module's name and location. If no file extension is specified in *osa_integration_module_name*, MULTI appends **.dll** (Windows) or **.so** (Linux/Solaris) to it automatically.

If the option is absent and the OSA integration is not built into MULTI, the default name (the lowercase name for the OSA integration) will be used.

You can set the OSA integration module's name and location with this option. If the integration module is specified with only a base name, MULTI first searches for it in the MULTI IDE installation directory, and then in a way defined by the host machine.

You can override the setting here by using the **-osa** MULTI command line option (see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 607). Alternatively, use the **osasetup** command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book).

The Memory Access Block Size

Format:

```
OSA_CONFIG:OSA_MEMORY_ACCESS_BLOCK_SIZE:number_in_bytes
```

This option allows you to customize the memory access blocks size between MULTI and the target to improve the performance of **OSA Explorer**.

The default memory access block size of MULTI is 64 bytes.

Log Interaction

Format:

```
OSA_CONFIG:OSA_DEBUGGING_LOGFILE:log_file
```

This option allows you to see the interaction between MULTI and the OSA integration module. All interactions will be saved in the file *log_file*.

In addition to the communication between MULTI and the OSA integration module, MULTI will also print some diagnostic information for the OSA integration module into the log file.

The Object Type Definition

Format:

```
OSA_OBJECT_TYPE:object_name:unique_identifier  
:OSA_GLOBAL|OSA_NOT_GLOBAL
```

Before you specify the detailed information about an object, it must be assigned a unique identifier.

The argument, *object_name*, is the string to be shown in the **OSA Explorer**. It can be `OSA_TASK` (for the *task* object) or a string for another object. You can change an object's name into any string you like.

An object can be either a global object (`OSA_GLOBAL`) or a local object (`OSA_NOT_GLOBAL`). A global object does not require any special context for the OSA integration module to access it, but a local object requires a global object as context in order to access it. So, only global objects are displayed as tabs in the **OSA Explorer**; local objects can be used only as references to other objects.

The Object Definition

A *task* is a special type of object. If there is no definition for it in the configuration file, MULTI will not be able to provide the multitasking debugging feature, but OS-awareness is still available for the objects defined in the configuration file.

An object's information contains three parts: references, attributes, and additional shortcut menu entries.

The Reference Specification

Format:

```
master_object_name:OSA_REFERENCE_LIST:reference_object_name  
:reference_id:reference_name
```

An object (the master object) can have more than one reference object, and each reference object is defined by a statement using this syntax.

The string *reference_id* should be unique for all references of the *master_object_name*.

The string *reference_name* will be displayed in the drop-down list above the reference pane in the **OSA Explorer**.

The Object Attribute Definition

Each object has a set of attributes. Some attributes are universal and important to objects for various OSA integrations. For example, tasks have the *identifier*, *name*, *status*, and *executable* attributes, while other objects just have the *identifier* attribute.

An object must have an identifier attribute. If one is not defined for an object, the configuration file is invalid.

Each attribute of an object is defined with a statement in the following syntax.

Format:

```
object_name:OSA_ATTRIBUTE_COLUMN:  
[OSA_ID=|OSA_NAME=|OSA_STATUS=|OSA_EXEC=]attribute_name  
:attribute_id:OSA_SHOW|OSA_NO_SHOW:sorting_type
```

OSA_ID=, OSA_NAME=, OSA_STATUS=, and OSA_EXEC= are used to tell MULTI that the defined attributes are the *identifier*, *name*, *status*, or *executable*, respectively, of the corresponding object. When Task Debugger mode is enabled,

`task identifier` and `task name` are shown in the title of the Debugger window, and `task status` is shown in the target list's **Status** column. If these attributes are not specified, MULTI uses default values in these places.

The string `attribute_id` must be unique for each attribute of an object.

`OSA_SHOW` and `OSA_NO_SHOW` tell MULTI which attribute columns are to be initially shown in the **OSA Explorer**.

`sorting_type` tells MULTI how to sort each column when it is clicked. The following are the valid sorting types:

- `OSA_LONG` — Sorts the corresponding column by long integer
- `OSA_STRING` — Sorts the corresponding column by string
- `OSA_DATE` — Sorts the corresponding column by date
- `OSA_FILENAME` — Sorts the corresponding column by filename
- `OSA_FLOAT` — Sorts the corresponding column by float value
- `OSA_ADDRESS` — Sorts the corresponding column by address (unsigned integers)

If no sorting type is defined for an attribute, its column will be sorted by string.

Shortcut Menu Entry Definitions

You can also define additional entries of the right-click shortcut menu in the object list pane of an **OSA Explorer**. Each such entry is defined by a statement in the following syntax.

Format:

```
object_name:OSA_MENU:menu_entry_string:multi_commands
```

Where *menu_entry_string* is the name to be displayed on the shortcut menu, and *multi_commands* is the command or commands to be executed whenever the menu option is selected.

If `menu_entry_string` is `"\x01"`, it defines a menu separator. For example, you can add a menu separator into **Semaphore** list's shortcut menu with the following line:

```
Semaphore:OSA_MENU:"\x01":{ }
```

Example: Configuration File

The following is the configuration file used by INTEGRITY.

```
// OSA Integration Package Configuration File for INTEGRITY
// Copyright (C) 2000-Present Green Hills Software

OSA_CONFIG:OSA_MULTI_MENU:INTEGRITY OSA Explorer...
OSA_CONFIG:OSA_EXPLORER_TITLE:INTEGRITY OSA Explorer (MULTI target ID 0x%x)
// Uncomment the following line to customize the memory access block
// size(in bytes) to turn up performance for your target.
// OSA_CONFIG:OSA_MEMORY_ACCESS_BLOCK_SIZE:512

// Uncomment the following line to generate an OSA log file
// OSA_CONFIG:OSA_DEBUGGING_LOGFILE:"integrity.log"

// Object Type List Section
// =====

// Define Object Type IDs and configure them for global display
// Format: OSA_OBJECT_TYPE:object name:object type id:global_display
OSA_OBJECT_TYPE:"Address Space":0:OSA_GLOBAL
OSA_OBJECT_TYPE:OSA_TASK:1:OSA_GLOBAL
OSA_OBJECT_TYPE:Connection:2:OSA_GLOBAL
OSA_OBJECT_TYPE:Activity:3:OSA_GLOBAL
OSA_OBJECT_TYPE:Semaphore:4:OSA_GLOBAL
OSA_OBJECT_TYPE:"Memory Region":5:OSA_GLOBAL
OSA_OBJECT_TYPE:Link:6:OSA_GLOBAL
OSA_OBJECT_TYPE:Clock:7:OSA_GLOBAL
OSA_OBJECT_TYPE:"I/O Device":8:OSA_GLOBAL
OSA_OBJECT_TYPE:Object:9:OSA_GLOBAL

// Address Space Window Format Sections
// =====
// Column Line Format:
//   type name:OSA_ATTRIBUTE_COLUMN:attribute name:attribute id:display:type
// List Line Format:
//   type name:OSA_REFERENCE_LIST:list type name:list id:list name
"Address Space":OSA_REFERENCE_LIST:OSA_TASK:0:Tasks
"Address Space":OSA_REFERENCE_LIST:Connection:1:Connection
"Address Space":OSA_REFERENCE_LIST:Activity:2:Activity
"Address Space":OSA_REFERENCE_LIST:Semaphore:3:Semaphore
"Address Space":OSA_REFERENCE_LIST:"Memory Region":4:"Memory Region"
"Address Space":OSA_REFERENCE_LIST:Link:5:Link
"Address Space":OSA_REFERENCE_LIST:Clock:6:Clock
```

```
"Address Space":OSA_REFERENCE_LIST:"I/O Device":7:"I/O Device"
"Address Space":OSA_REFERENCE_LIST:Object:8:Objects

"Address Space":OSA_ATTRIBUTE_COLUMN:OSA_ID="Domain ID":0:OSA_SHOW:OSA_ADDRESS
"Address Space":OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
"Address Space":OSA_ATTRIBUTE_COLUMN:Virtual:2:OSA_SHOW:OSA_STRING
"Address Space":OSA_ATTRIBUTE_COLUMN:Objects:3:OSA_SHOW:OSA_LONG
"Address Space":OSA_MENU:"View AddressSpace Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct DomainStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Domain ID is invalid.\n";}

// Task Space Window Format Section
// =====
// Column Line Format:
//   OSA_TASK:OSA_ATTRIBUTE_COLUMN:attribute name:attribute id:display:type
OSA_TASK:OSA_REFERENCE_LIST:Activity:0:"Interrupt Stack"
OSA_TASK:OSA_REFERENCE_LIST:Activity:1:"Other Activities"
OSA_TASK:OSA_REFERENCE_LIST:Semaphore:2:"Owned Binary Semaphores"
OSA_TASK:OSA_REFERENCE_LIST:Semaphore:3:"Owned HL Semaphores"
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_NAME=Name:0:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_ID=TaskID:1:OSA_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_STATUS=Status:2:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Priority:3:OSA_SHOW:OSA_LONG
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Weight:10:OSA_SHOW:OSA_LONG
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack Start:4:OSA_NO_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack End:5:OSA_NO_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack HWM/Size:6:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:"Address Space":7:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_EXEC=Executable:8:OSA_NO_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Object Index:9:OSA_SHOW:OSA_STRING
//OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack HWM:10:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_MENU:"View Task Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct TaskContextStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Task ID is invalid.\n";}
OSA_TASK:OSA_MENU:"\x01":{}
OSA_TASK:OSA_MENU:Debug Task:if ($_OSA_OBJ_ID) \
    {osatask $_OSA_OBJ_ID} \
    else {print "Task ID is invalid.\n";}

//Clocks Window Format Section
Clock:OSA_ATTRIBUTE_COLUMN:OSA_ID="Clock ID":0:OSA_SHOW:OSA_ADDRESS
Clock:OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
Clock:OSA_ATTRIBUTE_COLUMN:"Address Space":2:OSA_SHOW:OSA_STRING
Clock:OSA_ATTRIBUTE_COLUMN:Object Index:3:OSA_SHOW:OSA_STRING
Clock:OSA_MENU:"View Clock Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ClockStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Clock ID is invalid.\n";}

//IODevice Window Format Section
"I/O Device":OSA_ATTRIBUTE_COLUMN:OSA_ID="Device ID":0:OSA_SHOW:OSA_ADDRESS
"I/O Device":OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
"I/O Device":OSA_ATTRIBUTE_COLUMN:Object Index:2:OSA_SHOW:OSA_STRING
"I/O Device":OSA_MENU:"View I/O Device Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct IIODeviceStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "I/O Device ID is invalid.\n";}
```

```
//Generic Object List format section
Object:OSA_REFERENCE_LIST:Link:0:"Links to Object"
Object:OSA_ATTRIBUTE_COLUMN:OSA_ID="Object ID":0:OSA_SHOW:OSA_STRING
Object:OSA_ATTRIBUTE_COLUMN:Type:1:OSA_SHOW:OSA_STRING
Object:OSA_ATTRIBUTE_COLUMN:"Address Space":2:OSA_SHOW:OSA_STRING
Object:OSA_MENU:"View Object Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ObjectStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Object ID is invalid.\n";}

//Activity list format section
Activity:OSA_ATTRIBUTE_COLUMN:OSA_ID="Activity ID":0:OSA_SHOW:OSA_ADDRESS
Activity:OSA_ATTRIBUTE_COLUMN:"Object Index":1:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Status:2:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Priority:3:OSA_SHOW:OSA_LONG
Activity:OSA_ATTRIBUTE_COLUMN:"Address Space":4:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Task:5:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:"Waiting On":6:OSA_SHOW:OSA_STRING
Activity:OSA_MENU:"View Activity Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ActivityStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Activity ID is invalid.\n";}

//Semaphore window format section
Semaphore:OSA_ATTRIBUTE_COLUMN:OSA_ID="Semaphore ID":0:OSA_SHOW:OSA_ADDRESS
Semaphore:OSA_ATTRIBUTE_COLUMN:"Object Index":1:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Type:3:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Owner:2:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Value:5:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Priority:6:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:"Address Space":4:OSA_SHOW:OSA_STRING
Semaphore:OSA_MENU:"View Semaphore Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct SemaphoreStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Semaphore ID is invalid.\n";}
Semaphore:OSA_MENU:"\x01":{}
Semaphore:OSA_MENU:"Take Semaphore":if ($_OSA_OBJ_ID) \
    {substitute dialogue TakeSemaphore %EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Semaphore ID is invalid.\n";}
Semaphore:OSA_MENU:"Release Semaphore":if ($_OSA_OBJ_ID) \
    {substitute osainject -ObjType "Semaphore" -ObjId %EVAL{print /x $_OSA_OBJ_ID}
    -MsgType "Release"; \
    } else {print "Semaphore ID is invalid.\n";}

//Link list format section
Link:OSA_ATTRIBUTE_COLUMN:OSA_ID="Link ID":0:OSA_SHOW:OSA_ADDRESS
Link:OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target Type":2:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target ID":3:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target Index":5:OSA_SHOW:OSA_STRING
Link:OSA_MENU:"View Link Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct LinkStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Link ID is invalid.\n";}

//MemoryRegion list format section
"Memory Region":OSA_ATTRIBUTE_COLUMN:OSA_ID="Memory Region ID":0:OSA_SHOW:OSA_ADDRESS
"Memory Region":OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
"Memory Region":OSA_ATTRIBUTE_COLUMN:Beginning:2:OSA_SHOW:OSA_ADDRESS
```

```
"Memory Region":OSA_ATTRIBUTE_COLUMN:End:3:OSA_SHOW:OSA_ADDRESS
"Memory Region":OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
"Memory Region":OSA_MENU:"View Memory Region Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct MemoryRegionStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Memory Region ID is invalid.\n";}

//Connection list format section
Connection:OSA_ATTRIBUTE_COLUMN:OSA_ID="Connection ID":0:OSA_SHOW:OSA_ADDRESS
Connection:OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"Other End":2:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"AS of Other End":3:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"ObjIndex of Other End":5:OSA_SHOW:OSA_STRING
Connection:OSA_MENU:"View Connection Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ConnectionStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Connection ID is invalid.\n";}
Connection:OSA_MENU:"\x01":{}
Connection:OSA_MENU:"Inject Message":if ($_OSA_OBJ_ID) \
    {substitute dialogue GetMessageToConnection %EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Connection ID is invalid.\n";}
```

Chapter 27

Establishing Serial Connections

Contents

Starting the Serial Terminal Emulator (MTerminal)	644
Using Quick Connect	645
Creating and Configuring Serial Connections	646
The MTerminal Window	650
Running the Serial Terminal Emulator in Console Mode	653
mterminal Syntax and Arguments	654

There are many cases in which it may be necessary or convenient to establish a serial connection. You can do this easily by using MULTI's serial terminal emulator (**MTerminal**), which you can access from many of the MULTI tools or run as a stand-alone program. You can connect to a local serial port with **MTerminal's** Quick Connect, or you can configure and save multiple Serial Connection Methods, which you can later use to establish serial connections. This chapter describes how to use the serial terminal emulator and how to create, save, and use Serial Connection Methods.

Starting the Serial Terminal Emulator (MTerminal)

The following list describes different ways you can access the **MTerminal** serial terminal emulator. (The list makes frequent mention of the **Serial Connection Chooser**, the **MTerminal** window, and the Debugger's **Srl** pane. For more information about these interfaces, see “Using the Serial Connection Chooser” on page 646, “The MTerminal Window” on page 650, and “The Serial Terminal Pane” on page 31.)

- From the MULTI Launcher, click  and then select **Open Terminal** from the drop-down menu that appears. Alternatively, select **Components** → **Open Serial Terminal** from the menu bar. These actions start the **Serial Connection Chooser**, which allows you to create or edit a Serial Connection Method and then connect to the serial port.
- From the MULTI Launcher, click  and then select the name of a previously created Serial Connection Method from the drop-down menu that appears. This starts the **MTerminal** window with a terminal connected to the specified serial port. If the connection is unsuccessful, an error message appears.
- From the command line, execute **mterminal** (for command usage, see “mterminal Syntax and Arguments” on page 654). This opens the serial terminal emulator as a stand-alone program. The emulator usually opens in GUI mode, displaying either the **MTerminal** window or the **Serial Connection Chooser**, depending on the arguments specified. On Linux/Solaris hosts, you can also open the emulator in console mode. For more information, see “Running the Serial Terminal Emulator in Console Mode” on page 653.
- From the MULTI Debugger, select **Tools** → **Serial Terminal** → **Make Serial Connection**. Alternatively, issue the **serialconnect** command with no arguments in the command pane. (For information about the **serialconnect** command, see

“Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) These actions start the **Serial Connection Chooser**, which allows you to create or edit a Serial Connection Method and then connect to a serial port. If the connection is successful, the **Srl** tab becomes available in the bottom pane of the Debugger. Selecting the **Srl** tab displays a terminal connected to the specified serial port. If the serial connection is unsuccessful, an error message appears and the **Srl** tab does not become available.

- From the MULTI Debugger, select **Tools** → **Serial Terminal** and then select the name of a previously created Serial Connection Method from the drop-down list that appears. Alternatively, issue the **serialconnect** command with arguments specifying the parameters for the connection in the command pane. (For information about the **serialconnect** command, see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) If the connection is successful, the **Srl** tab becomes available in the bottom pane of the Debugger. Selecting the **Srl** tab displays a terminal connected to the specified serial port. If the serial connection is unsuccessful, an error message appears and the **Srl** tab does not become available.

Using Quick Connect

You can connect to a local serial port easily and quickly by using **MTerminal's** Quick Connect. With Quick Connect, you do not need to create and configure a connection. To use Quick Connect, open an **MTerminal** window and perform the following steps:

1. Select an available baud rate from the **Baud Rate** drop-down list.
2. Select one of the commonly used ports from the **Port** drop-down list, or type the name of a port in the **Port** field.
3. Click the **Quick Connect** button.

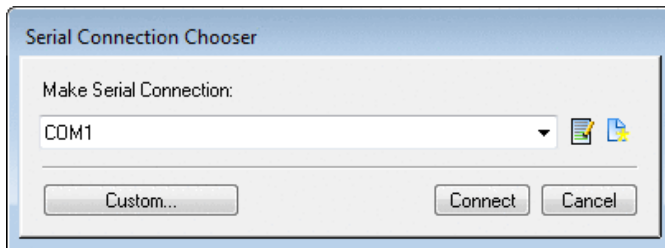
For information about serial connections that you can configure and save, see the next section.

Creating and Configuring Serial Connections

MULTI provides two graphical tools, the **Serial Connection Chooser** and the **Serial Connection Settings** dialog box, that allow you to configure and save one or more Serial Connection Methods and use those methods to establish serial connections and open terminal emulator windows. (These tools are analogous to the **Connection Chooser** and **Connection Editor**, which allow you to configure and save Connection Methods for connecting to your target. See Chapter 3, “Connecting to Your Target” on page 39.) The **Serial Connection Chooser** and the **Serial Connection Settings** dialog are described in the following sections.

Using the Serial Connection Chooser

You can use the **Serial Connection Chooser** to create a new Serial Connection Method, edit an existing Serial Connection Method, or establish a serial connection using a saved Serial Connection Method. For information about opening the **Serial Connection Chooser**, see “Starting the Serial Terminal Emulator (MTerminal)” on page 644.



To create a new standard Serial Connection Method from the **Serial Connection Chooser**, click . To edit an existing Serial Connection Method, select the Method from the drop-down list and click . Both of these actions open a **Serial Connection Settings** dialog, which allows you to configure your new or existing Serial Connection Method. For more information, see “Using the Serial Connection Settings Dialog” on page 647.

After you have created at least one Serial Connection Method, you can establish a connection by selecting your desired method from the drop-down list and then clicking **Connect**.

You can also create a Custom Serial Connection Method by entering a command in the **Serial Connection Chooser**, rather than using the graphical **Serial**

Connection Settings dialog. To do this, click the **Custom** button in the **Serial Connection Chooser** and use the syntax given below to describe the connection in the **Make Serial Connection** text field of the Chooser.

serialconnect *port_name* [*mterminal_parameters*]

This command accepts the same arguments as the **mterminal** command, which is used to launch **MTerminal** as a stand-alone application. For a description of the arguments that can be used to set the parameters of your connection, see “mterminal Syntax and Arguments” on page 654.

Saved serial connections are stored in:

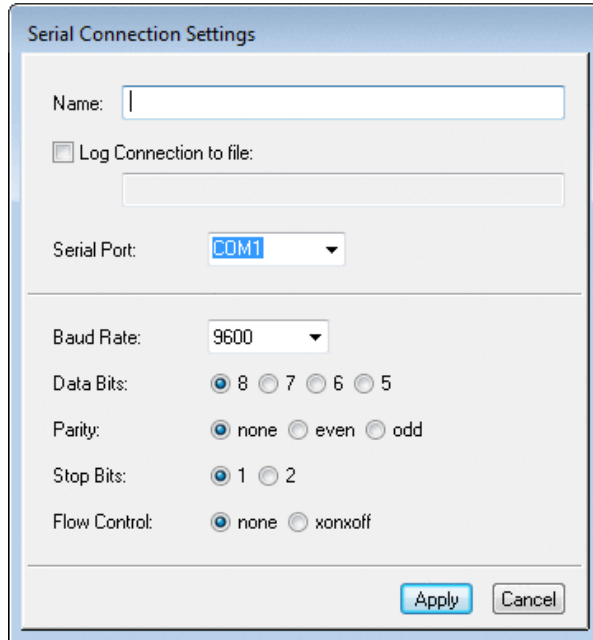
- Windows 7/Vista — *user_dir*\AppData\Roaming\GHS\serialconnection.odb
- Windows XP — *user_dir*\Application Data\GHS\serialconnection.odb
- Linux/Solaris — *user_dir*/.ghs/serialconnection.odb

Using the Serial Connection Settings Dialog

Like the Connection Methods described in Chapter 3, “Connecting to Your Target” on page 39, Serial Connection Methods serve as templates that specify the settings MULTI should use to establish a particular serial connection. You can use the **Serial Connection Settings** dialog to configure a Serial Connection Method.

To create a new Serial Connection Method or change the configuration of an existing Serial Connection Method:

1. Open the **Serial Connection Chooser** (see “Starting the Serial Terminal Emulator (MTerminal)” on page 644).
2. Click  to create a new connection or  to edit an existing connection. This will open a **Serial Connection Settings** dialog.



3. Enter or change the name of your Serial Connection Method. If no name is entered, the method you create will be a Temporary Serial Connection Method and will exist only for the duration of the current **MTerminal** session.
4. Enter or change the configuration settings for your Serial Connection Method. For a description of the setting options, see “Serial Connection Settings” on page 649.
5. Click **Create** or **Apply** to save the Serial Connection Method (the button label will depend on whether you are creating a new Method or editing an existing one). The new (or modified) connection will be displayed in the **Serial Connection Chooser** drop-down list.

Saved serial connections are stored in:

- Windows 7/Vista — *user_dir*\AppData\Roaming\GHS\serialconnection.odb
- Windows XP — *user_dir*\Application Data\GHS\serialconnection.odb
- Linux/Solaris — *user_dir*/.ghs/serialconnection.odb

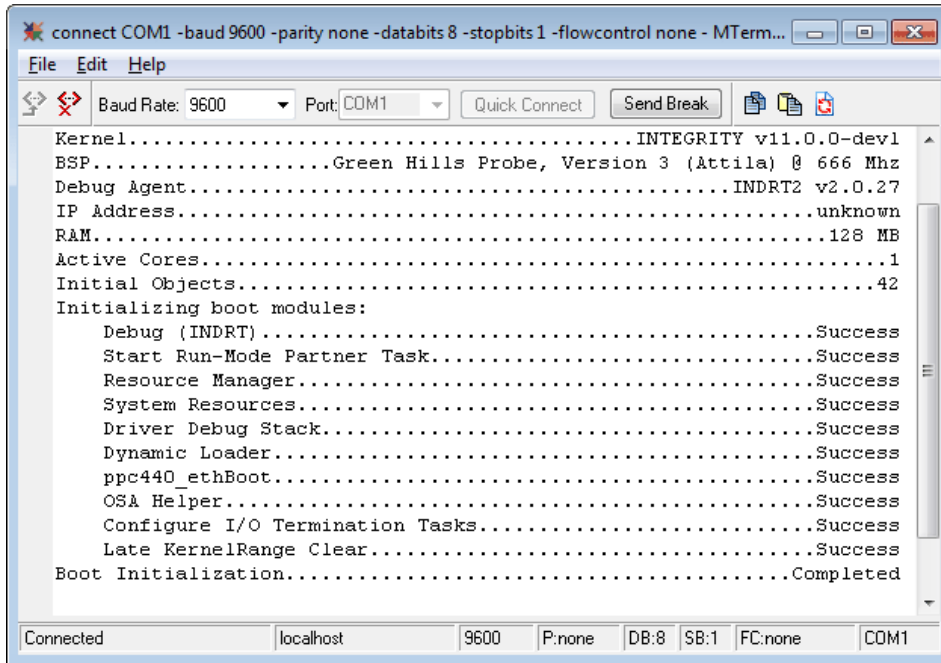
Serial Connection Settings

The following table describes the various settings that can be edited from the **Serial Connection Settings** dialog.

Name Specifies the name of the connection. If no name is entered, the method you create will be a Temporary Serial Connection Method and will exist only for the duration of the current MTerminal session.
Log Connection to file Logs the output from the serial port to the specified file. By default, this option is cleared (i.e., logging is disabled).
Serial Port Identifies the name of the serial port to connect to. The default values in the list are determined by the local host. If none of the values are applicable, you can type in the appropriate name. (Serial port names can be prepended with /dev.)
Baud Rate Specifies the baud rate for your connection. This setting defaults to 9600 .
Data Bits Specifies the data bits for your connection. This setting defaults to 8 .
Parity Specifies the parity for your connection. This setting defaults to none .
Stop Bits Specifies the stop bits for your connection. This setting defaults to 1 .
Flow Control Specifies the flow control for your connection. This setting defaults to none .

The MTerminal Window

The **MTerminal** window is a serial terminal window that opens when you use one of the methods described in “Starting the Serial Terminal Emulator (MTerminal)” on page 644.



You can use the main terminal window to interact with a serial port to which you have connected. This window provides full **VT100** support.

The MTerminal Menu Bar

The **MTerminal** menu bar contains three menus: the **File** menu, **Edit** menu, and **Help** menu. The following sections describe the items available from each of these menus.

The File Menu

The table below lists the items available in the **File** menu.

Item	Meaning
Connect	Opens the Serial Connection Chooser , which allows you to create and/or edit Serial Connection Methods. For more information, see “Using the Serial Connection Chooser” on page 646. This menu item is only available if the MTerminal window is not connected to any serial port.
Disconnect	Closes the current serial connection. This menu item is only available if the MTerminal window is connected to a serial port.
Close MTerminal	Closes the MTerminal window.

The Edit Menu

The table below lists the items available in the **Edit** menu.

Item	Meaning
Baud Rate	Opens a submenu that allows you to adjust the baud rate of a serial connection dynamically. The submenu items are only available if a serial connection has been established.
Send Break	Sends a serial break to the serial port. This menu item is only available if a serial connection has been established.
Copy	Copies text selected in the terminal window.
Paste	Pastes previously copied text into the terminal window. This menu item is only available if a serial connection has been established.
Clear	Clears text from the terminal window.

The Help Menu

The table below lists the items available in the **Help** menu.

Item	Meaning
MTerminal Help	Displays online help information about MTerminal .

Item	Meaning
About	Shows basic information about MTerminal , including the version number and revision date.

The MTerminal Toolbar

The **MTerminal** toolbar contains buttons and drop-down lists that allow you to connect to a serial port, disconnect from a serial port, Quick Connect, and copy, paste, and clear text in the terminal window. The following table describes the buttons and fields available on the **MTerminal** toolbar.

Item	Meaning
 Connect	Establishes a serial connection. This button is only available if the MTerminal window is not connected to any serial port. Clicking this button opens a drop-down menu that contains the following selections: • Connect — Opens the Serial Connection Chooser, which allows you to create and/or edit Serial Connection Methods. For more information, see “Using the Serial Connection Chooser” on page 646. • <i>Recently used Serial Connection Methods</i> — Attempts to establish a connection to the serial port described in the selected method.
 Disconnect	Closes the current serial connection. This button is only available if the MTerminal window is connected to a serial port.
Baud Rate	Allows you to adjust the baud rate of a serial connection dynamically.
Port	Specifies the local serial port to use for Quick Connect.
Quick Connect	Connects to a local serial port based on the information selected in the Baud Rate and Port drop-down lists.
Send Break	Sends a serial break to the serial port. This button is only available if a serial connection has been established.
 Copy	Copies text selected in the terminal window.
 Paste	Pastes previously copied text into the terminal window. This button is only available if a serial connection has been established.

Item	Meaning
 Clear	Clears text from the terminal window.

The MTerminal Status Bar

The **MTerminal** status bar lists information about the parameters of the serial connection. Status bar information, as it is displayed from left to right, follows:

- The status of the connection (either **Connected** or **Disconnected**)
- The name of the host on which the serial port is located (or **localhost** if the port is on the local host)
- The baud rate
- The parity (preended by **P:**)
- The data bits (preended by **DB:**)
- The stop bits (preended by **SB:**)
- The flow control (preended by **FC:**)
- The name of the serial port

Running the Serial Terminal Emulator in Console Mode

MTerminal can be run in a non-GUI console mode on Linux/Solaris hosts. This might be useful when running validations. To run **MTerminal** in console mode, launch the serial terminal emulator from the command line using the **-nodisplay** option. For example, to connect to the `ttys0` port using the default parameters for the serial settings, you would enter the command:

mterminal `ttys0 -nodisplay`



Note

Console mode does not have **VT100** support.

mterminal Syntax and Arguments

To launch **MTerminal** as a stand-alone application, run the **mterminal** command from the command line. The command usage is:

mterminal *port_name* [*mterminal_parameters*]

where *port_name* specifies what serial port is being used (for example, `ttysa`, `ttys0`, or `COM1`), and *mterminal_parameters* can be one or more of the options listed in the following table. (Parameters that are not specified use default values.)



Note

The following arguments can also be used with the **serialconnect** command, which you can enter in the **Make Serial Connection** text field of the **Serial Connection Chooser**. For more information, see “Using the Serial Connection Chooser” on page 646.

-baud <i>baudrate</i>	Specifies the baud rate, where <i>baudrate</i> can be any one of the following: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, or 230400. The default is 9600.
-databits <i>DB</i>	Specifies the data bits, where <i>DB</i> can be 5, 6, 7, or 8. The default is 8.
-flowcontrol <i>FC</i>	Specifies flow control, where <i>FC</i> can be <code>none</code> or <code>xonxoff</code> . The default is <code>none</code> .
-help	Prints help information. This option is only valid in console mode and does not work on Windows hosts.
-log_file <i>filename</i>	Enables logging and specifies the name of the file to which data is written. By default, logging is disabled.
-nodisplay	Runs mterminal in console mode. This option causes all output to be printed to standard output and all input to be received from standard input. In console mode, no graphical windows appear and the serial port must be specified. This option does not work on Windows hosts.
-parity <i>P</i>	Specifies parity, where <i>P</i> can be <code>none</code> , <code>even</code> , or <code>odd</code> . The default is <code>none</code> .
-stopbits <i>SB</i>	Specifies stop bits, where <i>SB</i> can be either 1 or 2. The default is 1.

The following example starts the GUI version of **MTerminal** on the serial port `ttyS0`. The baud rate is 38400.

```
mterminal ttyS0 -baud 38400
```


Part VI

Appendices

Appendix A

Debugger GUI Reference

Contents

Debugger Window Menus	660
The Debugger Window Toolbar	691
The Target List Shortcut Menu	702
Source Pane Shortcut Menus	704
The Command Pane Shortcut Menu	712
The Source Pane Search Dialog Box	712
The File Chooser Dialog Box (Linux/Solaris)	713

This appendix contains detailed descriptions of the menus and toolbar buttons available in the main Debugger window, as well as descriptions of some of the dialog boxes you can open via these menus and buttons. Most of these items are mentioned in the main text of this book in the context in which they are used. They are listed here together to provide a comprehensive reference.

For information about other GUI elements of the Debugger window, including descriptions of the target list, source pane, navigation bar, and information panes, see Chapter 2, “The Main Debugger Window” on page 11.



Note

Some menu items, shortcuts, commands, and dialog box buttons open a file chooser that allows you to browse to and specify a file for a particular operation. For information about how to use a Windows file chooser, see your Windows documentation. For information about the Linux/Solaris file chooser, see “The File Chooser Dialog Box (Linux/Solaris)” on page 713.

Debugger Window Menus

The sections below describe the menu items that are accessible from the Debugger window's menu bar.

Context-sensitive menu items are dimmed if they are unavailable in your current environment. For example, if you are debugging a running process, the **Go on Selected Items** item in the **Debug** menu is dimmed.

The menus on the main Debugger window, from left to right, are:

- The **File** menu (see “The File Menu” on page 661)
- The **Debug** menu (see “The Debug Menu” on page 665)
- The **View** menu (see “The View Menu” on page 673)
- The **Browse** menu (see “The Browse Menu” on page 679)
- The **Target** menu (see “The Target Menu” on page 680)
- The **TimeMachine** menu (see “The TimeMachine Menu” on page 683)
- The **Tools** menu (see “The Tools Menu” on page 685)

- The **Config** menu (see “The Config Menu” on page 687)
- The **Windows** menu (see “The Windows Menu” on page 690)
- The **Help** menu (see “The Help Menu” on page 691)

The File Menu

The following table describes the selections available from the **File** menu in the main Debugger window.

Menu Item	Meaning
Debug Program as New Entry	Opens a file chooser from which you can select a program to debug. MULTI adds the program to the target list and loads it in the Debugger window. The script file (if any) of the new program being debugged is executed and the final debugging environment is shared by the active Debugger window and secondary Debugger windows (if any). Corresponds to: the dbnew command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Debug Program	Similar to Debug Program as New Entry, except that the previously debugged program is removed from the target list and terminated or detached. The script file (if any) of the new program being debugged is executed to establish the final debugging environment. Any configuration changes made during the debugging session of the previous program remain in effect unless overridden. Corresponds to: the debug command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Print	Opens the Print (Windows) or Print Setup (Linux/Solaris) dialog box. The dialog box allows you to print the current source file, including interlaced assembly code if it is the current display mode, but not including non-ASCII characters. If no high-level source file is available (that is, there is only assembly code), this option is unavailable. For details, see “The Print Setup Dialog Box” on page 663.
Print Window	Opens the Print (Windows) or Print Setup (Linux/Solaris) dialog box to print the visible, ASCII contents of the source pane. For details, see “The Print Setup Dialog Box” on page 663.

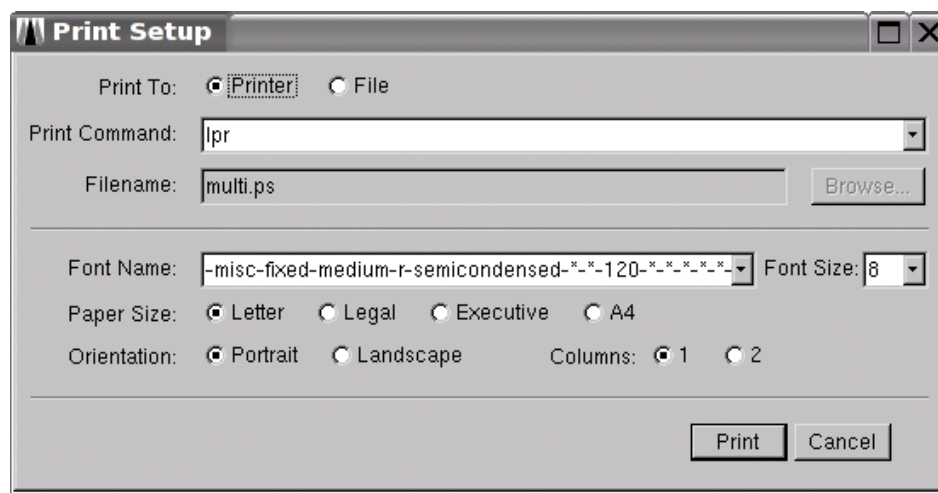
Menu Item	Meaning
Write to File	Opens a file chooser for you to specify a file to save to. You can save the source file, including interlaced assembly code if it is the current display mode, but you cannot save non-ASCII characters. If no high-level source file is available (that is, there is only assembly code), only the contents of the source pane are saved. Click Save in the file chooser to perform the save.
Attach to Process	Opens a dialog box for you to specify the ID of a running task. This menu option only works with a multitasking target. Clicking OK in the dialog box adds a new target list entry for the selected process and loads the task in the Debugger window. Corresponds to: the attach command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Detach from Process	In run mode, detaches the Debugger from the current process, and selects the next program in the target list. If you are not debugging in run mode, the entry for the current process is removed from the target list, but the debug server remains connected. All breakpoints are removed before detaching. Corresponds to: the detach command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
1 pathname 2 pathname 3 pathname 4 pathname	Lists the most recent programs opened in the Debugger. To debug one of these in the current Debugger window, select it.
Close Entry	Removes the currently selected entry from the target list. If the entry is a connection, selecting this option disconnects from it. In run mode, MULTI detaches from a selected process. If you are not debugging in run mode, MULTI terminates a selected process. If the entry is the only remaining entry in the target list, all Debugger windows close and the Debugger exits. Corresponds to: Ctrl+W and the quit entry command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Close Debugger Window	Closes the current Debugger window, but does not detach from or terminate any process unless the current Debugger window is the only one remaining. If the current Debugger window is the only one remaining, MULTI detaches from run-mode processes and terminates non-run-mode processes when it closes the window. Corresponds to:  , Ctrl+Q , and the quit window command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Exit All	Closes all MULTI IDE windows and exits. Corresponds to: the quitall command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Print Setup Dialog Box

On Windows, selecting a printing operation from the **File** menu opens a standard **Print** dialog box with settings and options appropriate for your version of Windows and your printer and printer driver. See your Windows and printer documentation for details about the settings on this dialog.

On Linux/Solaris systems, selecting a printing operation from the **File** menu opens the following **Print Setup** dialog box.



The settings of the **Print Setup** dialog box are described in the following table.

Item	Meaning
Print To	Specifies where to send the print output. Select either Printer (the default) or File (to write to a postscript file).
Print Command	Specifies the print command that will be run when you click Print. Enter the appropriate command and options for printing a file on your specific system (for example, lpr on Linux/Solaris). You can also set the print command with the configure printCommand command. For information about the configure command, see “Using the configure Command” in Chapter 7, “Configuring and Customizing MULTI” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book. This field is only available if you have chosen the Printer radio button above.
Filename	Specifies the output post-script file for print to file operations. This field is only available if you have selected the File radio button above. Enter the file to write to, or click Browse to open a chooser and navigate to an existing file. (See “The File Chooser Dialog Box (Linux/Solaris)” on page 713 for a description of the chooser.)
Font Name	Specifies the font for your printing operations. Select your desired font from the drop-down list, which contains a list of available fonts on your system. You can also type a font name into this field if it is not in the list.
Font Size	Specifies the font size used for printing. Select your desired size from the drop-down list. The default font size is 8 point.
Paper Size	Specifies your paper size. Select Letter , Legal , Executive , or A4 . The default setting is Letter .
Orientation	Specifies the layout for your printed document. Select Portrait or Landscape . The default setting is Portrait .
Columns	Specifies whether to print the output in one or two columns. The default setting is 1 .
Print	Executes the print request.
Cancel	Cancels the print request and discards any settings you have changed.

The Debug Menu

The following table describes the selections available from the **Debug** menu in the main Debugger window.

Menu Item	Meaning
Set Program Arguments	Opens a dialog box where you can specify the arguments for the current program. (Program arguments can only be passed to stand-alone applications that are started from MULTI.) The dialog box also allows you to control input and output redirection for the current process. See “The Arguments Dialog Box” on page 668. Corresponds to: the setargs command and the r command (see “General Program Execution Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Go on Selected Items	Starts or resumes items selected in the target list. This menu item cannot be used to start tasks on VxWorks systems. For information about starting tasks on these systems, see the runtask command in “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Corresponds to: , F5, and the c command (see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Restart	Starts or restarts the current program being debugged, with preset arguments (if any). If you select this menu item while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This menu item may not be available for relocatable modules. Corresponds to: , Ctrl+Shift+F5, and the restart command (see “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Halt on Selected Items	Halts the items selected in the target list. Corresponds to: , and the halt command (see “Halt Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Kill Selected Items	Kills the items selected in the target list. Corresponds to: Shift+F5 and the k command (see “Halt Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

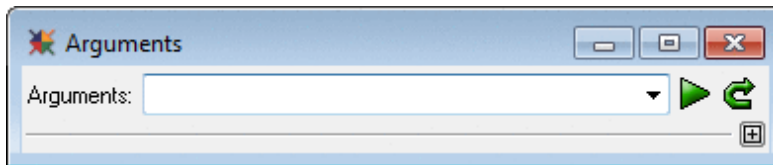
Menu Item	Meaning
Step (into Functions) on Selected Items	Executes single statements, stepping into procedure calls of items that are selected in the target list. Corresponds to: , F11, and the s command (see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Next (over Functions) on Selected Items	Executes single statements, stepping over procedure calls of items that are selected in the target list. Corresponds to: , F10, and the n command (see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Return on Selected Items	For items that are selected in the target list, continues execution to the end of the current subroutine and stops in the calling routine after returning to it. Corresponds to: , F9, and the cU command (see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Run to Cursor	Continues execution to the location of the current line pointer. Corresponds to: Ctrl+F10
Halt System	Halts all tasks on the target system, thereby freezing the target. This menu item is not supported on all operating systems and only appears if a run-mode connection is selected in the target list. Corresponds to: the groupaction -h @system command (see Chapter 19, “Task Group Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Run System	Restores the tasks on the target to the status they held before the system was halted. This menu item only appears if a run-mode connection is selected in the target list. Corresponds to: the groupaction -r @system command (see Chapter 19, “Task Group Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Send Signal	Opens a dialog box that specifies the signal name and then sends a signal to the current process. To list signal names, enter the l z (lowercase L lowercase Z) command. See the l command in Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Menu Item	Meaning
Set Watchpoint	Opens a dialog box that creates a watchpoint. Corresponds to: the watchpoint command (see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Remove All Breakpoints	Deletes software breakpoints: • If the OSA master process is selected in the target list — Deletes all normal software breakpoints (except group breakpoints) in the master process. For more information, see “Working with Freeze-Mode Breakpoints” on page 618. • If a run-mode AddressSpace is selected in the target list and you are using INTEGRITY 10 or later — Deletes all any-task breakpoints in the AddressSpace. • If a task is selected in the target list — Deletes all task-specific software breakpoints in the task. Corresponds to: the D command (see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Toggle All Breakpoints	Toggles the active state of all software breakpoints of the current program (i.e. all currently enabled breakpoints will be disabled, and all currently disabled breakpoints will be enabled). Corresponds to: the Tog command (see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Prepare Target	Opens the Prepare Target dialog box, which you can use to select a method for setting up your target. You can download your program to RAM, flash it to ROM, or verify that it is already on the target. This menu item opens the Prepare Target dialog box even if you saved an action for the selected program by clicking Automatically use these settings for this program next time in the dialog box the last time you prepared the target. For more information, see “Preparing Your Target” on page 108. This menu item may not be available for relocatable modules. Corresponds to:  and the prepare_target command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Use Connection	Opens the Use Connection submenu. See “The Use Connection Submenu” on page 670.

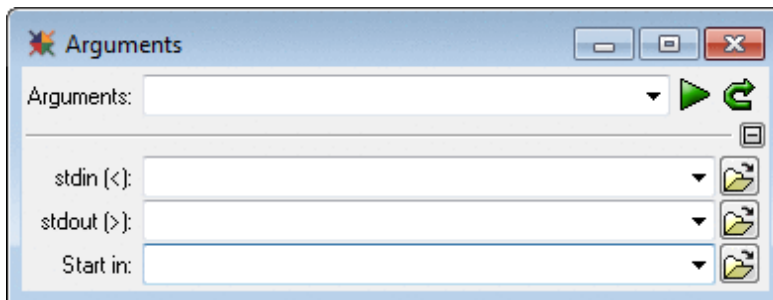
Menu Item	Meaning
Debug Settings	Opens the Debug Settings submenu. See “The Debug Settings Submenu” on page 670.
Target Settings	Opens the Target Settings submenu. See “The Target Settings Submenu” on page 672.

The Arguments Dialog Box

The **Arguments** dialog box opens when you select **Debug** → **Set Program Arguments**. When the dialog box first appears, it may be displayed in a contracted form:



To expand the dialog box to see the I/O setting fields, click the small plus sign icon (⊕) on the bottom right corner of the contracted dialog box. The expanded dialog box is shown next.



To hide the three I/O fields, click the minus sign (⊖) to the right of the line dividing the top and bottom sections of the dialog box.

The following table describes the items in the expanded **Arguments** dialog box.

Item	Meaning
Arguments	Specifies the arguments to be passed to the current program the next time you run it without specifying any arguments (for example, by clicking  or  , or by executing the r command, etc.). Program arguments can only be passed to stand-alone applications that are started from MULTI. If you specify any arguments in the next execution (with the r command, for example) the existing arguments in the Arguments field are replaced by the ones specified. These new arguments are displayed the next time the Arguments dialog box is opened.
The  button	Runs the program with the arguments specified in the Arguments text field.
The  button	Restarts the process with the arguments specified in the Arguments text field.
The  button	Closes the Arguments dialog box. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons . To access this option, select Config → Options → General Tab .
stdin (<)	Specifies a file on the host system that will be used as input to your process. Enter a filename, or click  to open a file chooser to browse to an existing file. If this field is not specified, the input comes from standard input.
stdout (>)	Specifies a file on the host system that will capture output from your process. Enter a filename, or click  to open a file chooser to browse to an existing file. If this field is not specified, the output goes to standard output.
Start in	Specifies the directory in which the process runs or, for embedded processes that use host I/O, specifies the directory that MULTI uses to perform host I/O operations. Enter a directory, or click  to open a directory chooser to browse to an existing directory. For information about the command equivalent of this field, see the rundir command in “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Use Connection Submenu

The following table describes the selections available from the **Debug → Use Connection** submenu.

Menu Item	Meaning
[Status] Connection_Name	Associates the current executable with the specified connection. Only compatible, currently active connections are listed. If a connection appears dimmed, the current executable cannot be associated with that connection, or it is already associated with that connection. Status is listed either as: Available (the connection can accept more executables), Current (the connection is associated with the current executable), or Full (using the connection will cause another executable to stop using it). For more information, see “Associating Your Executable with a Connection” on page 105.
Stop Using Current Connection	Disassociates the selected executable from the connection it is currently associated with. Selecting this menu item does not disconnect from the target debug server. For more information, see “Associating Your Executable with a Connection” on page 105. Corresponds to: the change_binding unbind command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Create New Connection	Opens the Connection Chooser , which you can use to establish a connection. For more information, see “Creating a Standard Connection Using the Connection Chooser” on page 43.

The Debug Settings Submenu

The following table describes the selections available from the **Debug → Debug Settings** submenu.

Menu Item	Meaning
Attach on Fork/Thread	Toggles the option to attach to child processes spawned by calls to <code>fork()</code>. This option is only supported on native targets. Corresponds to: the P c command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Memory Sensitive	Controls whether the Debugger attempts to limit memory reads from the target. See “Memory Sensitive Mode” on page 115. Corresponds to: the <code>_VOLATILE</code> system variable.
Cache Memory Reads	Controls whether the Debugger uses a cache to read memory more efficiently. When enabled, the Debugger reads memory in blocks, which are stored in a cache. This reduces the number of memory accesses required. For example, with the cache active, printing a structure with ten members would require only one memory read as opposed to ten memory reads. However, this could cause problems when accessing volatile memory. Corresponds to: the <code>_CACHE</code> system variable (see “System Variables” on page 310).
Disassemble From Host	Controls whether the Debugger displays disassembly for programs based on the code in the executable file, as opposed to the contents of memory. When enabled, the Debugger disassembles instructions found in the executable file. This has the advantage of not requiring any memory reads from the target. When disabled, the Debugger reads instructions from target memory when disassembling. Corresponds to: the <code>_ASMCACHE</code> system variable (see “System Variables” on page 310).
No Stack Trace	Controls whether the Debugger is allowed to access the stack for stack trace information. When the setting is enabled, stack traces are disallowed. This can be useful if you know that the stack does not contain a valid stack trace, and you want to prevent the Debugger from attempting to trace into invalid memory. See “No Stack Trace Mode” on page 116. Corresponds to: the <code>_NOSTACKTRACE</code> system variable.
Auto Check Coherency	Controls whether the Debugger checks coherency automatically. When automatic coherency checking is enabled, MULTI checks the coherency of the specified number of addresses at every stop (see Number of Addresses to Check below). If it finds discrepancies between the contents of memory and what you loaded onto your target, it highlights the differing lines in the source pane. See “Checking Coherency Automatically” on page 537. Corresponds to: the <code>_AUTO_CHECK_COHERENCY</code> system variable (see “System Variables” on page 310).

Menu Item	Meaning
Number of Addresses to Check	Opens a dialog box where you can enter the number of addresses you want checked for coherency on each stop. By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer). MULTI attempts to check an equal number of addresses before and after the current program counter; however, it does not cross procedure boundaries. For more information, see “Checking Coherency Automatically” on page 537. See also Auto Check Coherency above. Corresponds to: the <code>_AUTO_CHECK_NUM_ADDRS</code> system variable (see “System Variables” on page 310).
Fast Source Step	Controls whether the Debugger attempts to speed up stepping through source code. When enabled, on each source step, the Debugger analyzes the code, sets temporary breakpoints on all possible step destinations, and allows the process to run until one of the breakpoints is hit. This allows the process to run at nearly full speed during the source step. When disabled, the Debugger usually steps one machine instruction at a time until it reaches the next source line. However, even when disabled, the Debugger sets temporary breakpoints and runs to step over function calls. This method is generally much slower. Corresponds to: the <code>FASTSTEP</code> system variable (see “System Variables” on page 310).

The Target Settings Submenu

The following table describes the selections available from the **Debug → Target Settings** submenu.

Menu Item	Meaning
Stop on Task Creation	Specifies whether you want the target's operating system to halt each task as soon as it is created by the application.
Attach on Task Creation	Specifies whether you want to debug the newly created task. If enabled, each newly created task is halted and opened in the Debugger window. This option is applicable only when the Stop on Task Creation menu item is selected.
Halt on Attach	Specifies whether you want MULTI to automatically halt tasks when you attach to them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).

Menu Item	Meaning
Run on Detach	Specifies whether you want MULTI to automatically run tasks when you detach from them. You cannot use this option when MULTI is in passive mode (see “Debugging in Passive Mode” on page 574).

The View Menu

The following table describes the selections available from the **View** menu in the main Debugger window.

Menu Item	Meaning
Navigation	Opens the Navigation submenu. See “The Navigation Submenu” on page 676.
Display Mode	Opens the Display Mode submenu. See “The Display Mode Submenu” on page 677.
Breakpoints	Opens the Breakpoints window. See “The Breakpoints Window” on page 146. Corresponds to:  and the breakpoints command (see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Call Stack	Opens a Call Stack window. See “Viewing Call Stacks” on page 388. Corresponds to:  and the callsv command (see Chapter 5, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Local Variables	Opens a Data Explorer displaying all local variables. If the current function is a C++ instance method, the Data Explorer also shows information about the <code>this</code> pointer. See Chapter 11, “Viewing and Modifying Variables with the Data Explorer” on page 183. Corresponds to: , the view \$locals\$ command, and the localsview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Registers	Opens a Register View window. See “The Register View Window” on page 254. Corresponds to:  and the regview command (see “Register Commands” in Chapter 14, “Register Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Memory	Opens an empty Memory View window. See Chapter 15, “Using the Memory View Window” on page 323. Corresponds to:  and the memview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Allocations	Opens the Memory Allocations window. See “Using the Memory Allocations Window” on page 340. Corresponds to: the heapview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Profile	Opens the Profile window. See “The Profile Window” on page 361. Corresponds to: the profile command (see Chapter 12, “Profiling Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Task Manager	Opens a Task Manager window if you are debugging in a run-mode environment. Otherwise it opens a Process Viewer. See “The Task Manager” on page 580 and “Viewing Native Processes” on page 390. If you are debugging in run mode, corresponds to: the taskwindow command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book). If you are debugging in a native environment, corresponds to: the top command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
OSA Explorer	Opens an OSA Explorer on the current process in a freeze-mode debugging environment or on the current debug server in a run-mode debugging environment, or opens the Linux Threads window in Linux. The OSA Explorer shows information for objects recognized by the OSA integration module. For information about the OSA Explorer, see “The OSA Explorer” on page 609. Corresponds to: the osaexplorer command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Debugger Notes	Opens a Note Browser, which allows you to browse your Debugger Notes. See “Viewing Debugger Notes” on page 177. Corresponds to: the noteview command (see Chapter 7, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Caches	Opens the Cache View window. See “The Cache View Window” on page 392. Corresponds to: the cacheview command (see “Cache View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Find Address in Cache	Opens the Cache Find window. See “The Cache Find Window” on page 396. Corresponds to: the cachefind command (see “Cache View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Print Expression	Opens a dialog box for you to specify an expression to be printed. Corresponds to: the print command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
View Expression	Opens a dialog box for you to enter an expression to view in a Data Explorer. Corresponds to: Shift+F9 and the view command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
List	Opens the List submenu. See “The List Submenu” on page 678.
Source Path	Opens the Source Path window, which allows you to change the directories that are searched for source files. Corresponds to: the source command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Refresh Views	Refreshes all Data Explorers, Register View windows, Memory View windows, Call Stack windows, etc. Frozen Data Explorer variables and frozen Register View, Memory View, and Call Stack windows are not updated. Corresponds to: the update command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Auto Update Views	Toggles whether the Debugger automatically updates all Data Explorers, Register View windows, Memory View windows, Call Stack windows, etc. (If a Data Explorer variable is frozen, this menu item does not update that variable.) The default update interval time is 1 second. To change the interval time, enter the update command followed by an interval number in seconds. For more information about the update command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Close All Views	Closes all Data Explorers, Browse windows, Register View windows, Memory View windows, Call Stack windows, and Breakpoints windows. Corresponds to: the viewdel command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Navigation Submenu

The following table describes the selections available from the **View** → **Navigation** submenu.

Menu Item	Meaning
UpStack	Views the procedure one higher on the call stack. Corresponds to:  , Ctrl++ , and the E + command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
DownStack	Views the procedure one lower on the call stack. Corresponds to:  , Ctrl+- , and the E - command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Current PC	Views the procedure where the process is currently stopped. Corresponds to:  and the E command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
UpStack To Source	Views the first procedure higher on the call stack that has source code. You can use this feature if you are stopped inside a library function with no source code (such as <code>printf()</code>), and you want to return to viewing your program. Corresponds to: the uptosource command (see Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Goto Location	Opens a dialog box in which you can specify a procedure or an address, and displays the program at the specified location in the source pane. Corresponds to: the e command (see Chapter 11, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Display Mode Submenu

The following table describes the selections available from the **View → Display Mode** submenu.

Menu Item	Meaning
Source Only	Displays only source code in the source pane. Corresponds to:  (not selected) and the assem off command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Interlaced Assembly	Displays source code and the corresponding assembly instructions for each source code line in the source pane. Corresponds to:  (selected) and the assem on command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Assembly Only	Displays only assembly instructions in the source pane. Corresponds to: the assem nosource command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The List Submenu

The following table describes the selections available from the **View → List** submenu.

See also the **l** (lowercase L) command in Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning
Files	Lists all the filenames in the program.
Procedures	Lists the names and addresses of all procedures.
Mangled Procedures	Lists the mangled names and addresses of all procedures.
Globals	Lists the names and addresses of all global variables.
Statics	Lists the names and addresses of all static variables.
Locals	Lists the names and values of all local variables for the procedure you are viewing (that is, the procedure at the current line pointer) if the procedure is on the stack. If the procedure is a C++ instance method, this menu item lists information about the <code>this</code> pointer as well.
Local Addresses	Lists the names and addresses of the local variables specified above.
Registers	Lists the names and values of all registers.
Register Synonyms	Lists the register synonyms.
Variables In Procedure	Lists all the parameters and local variables of the specified procedure if it is on the stack.
Defines	Lists all the defined macros.
MULTI Variables	Lists the values of all MULTI special variables.
Processes	Lists the processes that the MULTI Debugger is currently attached to.
Signals	Lists the names and configuration settings of all signals.
Breakpoints	Lists all software breakpoints.
Dialog Boxes	Lists all loaded dialog boxes.
Source Paths	Lists the directories where MULTI looks for source files and scripts.
Included Files	Lists the source files included by the current file.

The Browse Menu

The following table describes the selections available from the **Browse** menu in the main Debugger window.

Menu Item	Meaning
Procedures	Opens a Browse window displaying all procedures in the program being debugged. See “Browsing Procedures” on page 220.
Globals	Opens a Browse window displaying all global variables in the program being debugged. See “Browsing Global Variables” on page 228.
Files	Opens a Browse window displaying the name and location of all of the source files containing procedures used in the program being debugged. See “Browsing Source Files” on page 230.
All Types	Opens a Browse window displaying all data types used in the program being debugged. See “Browsing Data Types” on page 233.
Classes	Opens a Tree Browser displaying the class hierarchy of the program being debugged. See “Browsing Classes” on page 243.
Static Calls	Opens a Tree Browser displaying the static calling relationships of the current procedure (i.e., which functions are called by and which functions call the procedure at the current line pointer in the Debugger). See “Browsing Static Calls By Function” on page 244.
Dynamic Calls	Opens a Tree Browser displaying which functions were actually called during run time. See “Browsing Dynamic Calls by Function” on page 246.
File Calls	Opens a Tree Browser displaying static calls by file. This view shows source files whose functions are called from other source files. See “Browsing Static Calls By File” on page 245.
Includes	Opens a Graph View window displaying the include file hierarchy. See “Browsing Includes” on page 247.
Procedures In File	Opens a dialog box that allows you to specify a source file, then opens a Browse window displaying all the procedures in the file you specified. See “Browsing Procedures” on page 220.
Type	Opens a dialog box that allows you to specify a type name, then opens a Data Explorer displaying the structure of the type you specified. See Chapter 11, “Viewing and Modifying Variables with the Data Explorer” on page 183.

The Target Menu

The following table describes the selections available from the **Target** menu in the main Debugger window.

Menu Item	Meaning
Connect	Opens the Connection Chooser, which allows you to create, edit, or invoke a Connection Method to connect to your target. See “Standard Connection Methods” on page 42. Corresponds to: , F4, and the connect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Show Connection Organizer	Opens the Connection Organizer, which configures how you connect to target hardware or simulators. For more information about the Connection Organizer, see “Using the Connection Organizer” on page 50. Corresponds to: the connectionview command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Disconnect from Target	Disconnects from the current target debug server. Corresponds to:  and the disconnect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Set Run-Mode Partner	Opens the Set Run-Mode Partner dialog box, in which you can select a run-mode connection that the Debugger automatically attempts to establish when you boot an INTEGRITY kernel via the current freeze-mode connection. The dialog box also allows you to disable this behavior. For more information, see “The Set Run-Mode Partner Dialog Box” on page 71.
Load Module	Opens a submenu that allows you to load a module into the target system's memory. This menu item is only available if the target supports and was configured with a dynamic loader (for example, the LoaderTask on INTEGRITY).
Unload Module	Opens a submenu that allows you to unload a module.
1 connection 2 connection 3 connection 4 connection	Lists the most recently connected debug servers. To connect to one of them, select it.

Menu Item	Meaning
IO Buffering	Toggles buffering for the I/O pane. Corresponds to: the iobuffer command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Manipulation	Opens the Memory Manipulation submenu. See “The Memory Manipulation Submenu” on page 681.
Memory Test	Opens the Memory Test Wizard . See “Quick Memory Testing: Using the Memory Test Wizard” on page 512.
Flash	Opens the MULTI Fast Flash Programmer , which allows you to enter parameters to transfer a file from the host to flash memory on the target. See Chapter 22, “Programming Flash Memory” on page 539. Corresponds to: the flash gui command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Memory Manipulation Submenu

The following table describes the selections available from the **Target → Memory Manipulation** submenu.

Menu Item	Meaning
Copy	Opens the Copy a Region of Memory dialog box, which allows you to copy memory contents from the From address to the Destination address in memory. The copying continues for the specified Number of blocks of memory, and you can specify the size of each memory block in bytes. Corresponds to: the copy command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Fill	Opens the Fill a Region of Memory dialog box, which allows you to fill memory starting from the Starting at address location with the given value Fill value . The filling continues for the specified Number of blocks of memory, and you can specify the size of each memory block in bytes. Corresponds to: the fill command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Find	Opens the Find a Region of Memory dialog box, which allows you to find a value in memory. The search begins at the Starting at address location and continues for Number of blocks of memory. You can specify the size of each memory block in bytes. For each memory location, MULTI performs a bitwise AND operation using the memory value and the mask as the operands, then compares the results with the Value to find. Corresponds to: the find command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Compare	Opens the Compare Memory Regions dialog box, which allows you to compare memory contents. You specify the two starting memory locations to compare (Address 1 and Address 2) and the Number of blocks of memory to compare. You can specify the size of each memory block in bytes. You can also specify the Comparison operation: == (equal), > (greater than), >= (greater than or equal), < (less than), <= (less than or equal), != (not equal). Corresponds to: the compare command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Load	Opens the Load Memory into Target dialog box, which allows you to copy a section of memory from the specified file (Load memory from file). Corresponds to: the memload command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Dump	Opens the Dump Memory into File dialog box, which allows you to copy a section of memory to the specified file (Dump memory to file), starting at the Start Address and continuing for Length bytes. Corresponds to: the memdump command (see “General Memory Commands” in Chapter 10, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The TimeMachine Menu

The following table describes the selections available from the **TimeMachine** menu in the main Debugger window. For information about the commands listed in the table, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning
Load Trace Session	Loads a previously saved trace session. Corresponds to: the traceload command.
Save Trace Session	Saves the trace session. Corresponds to: the tracesave command.
Enable Trace	Starts trace collection and clears any previously collected data on the target. Data that has already been retrieved is not cleared, but if trace retrieval is currently in progress, it is aborted. Corresponds to: the trace enable command.
Disable Trace	Stops trace collection and retrieves trace data. For more information, see “Enabling and Disabling Trace Collection” on page 405. Corresponds to: the trace disable command.
Retrieve Trace	Retrieves trace data from the trace probe or target. With SuperTrace Probe v3 targets, this either retrieves the amount of data set by the Target buffer size option (see “The Collection Tab” on page 481), or it retrieves twice as much data as has already been retrieved. In the latter case, all previously retrieved trace data is retrieved again. With all other targets, all trace data is always retrieved. Corresponds to: the trace retrieve command.
Abort Trace Retrieval	Aborts the retrieval of trace data. Corresponds to: the trace abort command.
Clear Data	Clears all current trace data on the host, trace probe, and target. Corresponds to: the trace clear command.
Set Triggers	Opens the Set Triggers window, which allows you to configure trace collection by setting triggers and other target-specific events. See “The Set Triggers Window” on page 493. Corresponds to: the trace triggers command.

Menu Item	Meaning
Trace Options	Opens the Trace Options window, which allows you to configure trace options related to trace collection and display. See “The Trace Options Window” on page 480. Corresponds to: the trace options command.
TimeMachine Debugger	Enables/disables TimeMachine mode for the currently selected item. In addition to the normal Debugger run-control operations, the TimeMachine Debugger allows you to step backwards in time. This option is only available if you have collected trace data. For more information, see “The TimeMachine Debugger” on page 413. Corresponds to:  and the timemachine command.
PathAnalyzer	Generates path analysis information from the current trace data and displays it in a PathAnalyzer. This option is available only if trace data is available. See “The PathAnalyzer” on page 424. Corresponds to: the tracepath command.
Trace Statistics	Opens the Trace Statistics window, which shows statistical information about the current trace data. This option is available only if trace data exists. For more information, see “Viewing Trace Statistics” on page 460. Corresponds to: the trace stats command.
Profile	Generates profiling data from the current trace data and displays it in the Profile window. This option is available only if trace data is available. See “Using Trace Data to Profile Your Target” on page 475. Corresponds to: the tracepro command.
Trace List	Opens the Trace List, which allows you to view and explore trace data at the assembly level. This window also provides access to advanced trace analysis tools. For more information, see “Viewing Trace Data in the Trace List” on page 434. Corresponds to: the trace list command.
EventAnalyzer	Generates an EventAnalyzer log from the current trace data and displays it in the MULTI EventAnalyzer. This option is available only if trace data is available. See “Viewing Trace Events in the EventAnalyzer” on page 474. Corresponds to: the tracemevsys command.

The Tools Menu

The following table describes the selections available from the **Tools** menu in the main Debugger window.

Menu Item	Meaning
MULTI EventAnalyzer	Opens the MULTI EventAnalyzer . This menu item is only available for some RTOSes. Corresponds to: 
MULTI ResourceAnalyzer	Opens the MULTI ResourceAnalyzer . This menu item is only available if you are using INTEGRITY. Corresponds to: 
Launcher	Opens the MULTI Launcher. Corresponds to: the multibar command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Project Manager	Opens the Project Manager on the project for the current program. If MULTI cannot find a project for the current program, it will ask which project to open. Corresponds to: the builder command (see Chapter 4, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Rebuild	Rebuilds the current program if the project for the program can be located. Corresponds to: the build command (see Chapter 4, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Editor	Opens an Edit File dialog box, which allows you to select a file to open in an Editor window. Corresponds to:  and the edit command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Launch Utility Programs	Opens the Utility Program Launcher , which provides a graphical interface to the Green Hills utility programs. Corresponds to: the wgutils command (see Chapter 4, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Serial Terminal	Opens the Serial Terminal submenu. See “The Serial Terminal Submenu” on page 686.

Menu Item	Meaning
Search	Opens MULTI's search dialog box. See “The Source Pane Search Dialog Box” on page 712. Corresponds to: the dialogsearch command (see Chapter 16, “Search Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Search in Files	Opens the Search in Files dialog box, which searches the source files of the program being debugged and any other open files for the specified string, using the grep utility. See “Searching in Files” on page 159. Corresponds to: the grep command (see Chapter 16, “Search Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Serial Terminal Submenu

The following table describes the selections available from the **Tools → Serial Terminal** submenu. For more information, see Chapter 27, “Establishing Serial Connections” on page 643.

Menu Item	Meaning
Make Serial Connection	Opens the Serial Connection Chooser , which allows you to create, edit, or invoke a Serial Connection Method. Corresponds to: the serialconnect command (see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Disconnect from Serial	Closes the current serial connection. Corresponds to: the serialdisconnect command (see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
1 connection 2 connection 3 connection 4 connection	Lists the four most recently used serial connections. To invoke one of these, select it.

The Config Menu

The following table describes the selections available from the **Config** menu in the main Debugger window.

Menu Item	Meaning
Options	Opens the Options dialog box, which allows you to configure the appearance and behaviors of the Debugger and other MULTI tools. For a description of all the options in this window, see Chapter 8, “Configuration Options” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book. Corresponds to: the configoptions command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Customize Menus	Opens the Customize Menus window, which allows you to configure the menus in a number of MULTI tools. For more information, see Chapter 7, “Configuring and Customizing MULTI” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book. Corresponds to: the customizemenus command (see “Button, Menu, and Mouse Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Customize Toolbar	Opens the Customize Toolbar window, which allows you to configure the Debugger toolbar. For more information, see “Configuring the Debugger Toolbar” on page 698. Corresponds to: the customizetoolbar command (see “Button, Menu, and Mouse Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Save Configuration as User Default	Saves the current configuration into the default user configuration file, so that it will be automatically loaded when MULTI starts. For more information, see “Saving Configuration Settings” in Chapter 7, “Configuring and Customizing MULTI” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book. Corresponds to: the saveconfig command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Clear User Default Configuration	Deletes the default user configuration file and restores the default system configuration. Corresponds to: the clearconfig command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Save Configuration As	Opens a file chooser in which you can choose the file where you want to save the current configuration. For more information, see “Saving Configuration Settings” in Chapter 7, “Configuring and Customizing MULTI” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book. Corresponds to: the saveconfigtofile command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Load Configuration	Opens a file chooser from which you can choose the file whose configuration you want to load. Corresponds to: the loadconfigfromfile command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Set INTEGRITY Distribution	Opens the Default INTEGRITY Distribution dialog box in which you can provide MULTI with the location of the installed INTEGRITY distribution. This information is used to add INTEGRITY documentation to MULTI's Help menu and to determine the default INTEGRITY distribution for use with the Project Wizard. For more information, see “Configuring MULTI for Use with INTEGRITY or u-velOSity” in Chapter 2, “MULTI Tutorial” in the <i>MULTI: Getting Started</i> book. Corresponds to: the setintegritydir command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Set u-velOSity Distribution	Opens the Default u-velOSity Distribution dialog box in which you can provide MULTI with the location of the installed u-velOSity distribution. This information is used to add u-velOSity documentation to MULTI's Help menu and to determine the default u-velOSity distribution for use with the Project Wizard. For more information, see “Configuring MULTI for Use with INTEGRITY or u-velOSity” in Chapter 2, “MULTI Tutorial” in the <i>MULTI: Getting Started</i> book. Corresponds to: the setuvelocitydir command (see “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
State	Opens the State submenu. See “The State Submenu” on page 689.

The State Submenu

The following table describes the selections available from the **Config → State** submenu.

Menu Item	Meaning
Show Command History	Prints the command history. Each Debugger window keeps a history of all the Debugger commands entered in that window. You can use the ! command to execute a command from the history (see “History Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book). You can also use the UpArrow and DownArrow keys to navigate through the history and choose a command for execution. Corresponds to: the h command (see “History Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Save State	Saves the state of the Debugger to the specified file. The saved information includes target connection and status, breakpoints, and the source directories list. (Note that the Debugger may not be able to restore saved group breakpoints.) Corresponds to: the save command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Restore State	Restores the state of the Debugger from the specified file. (Note that the Debugger may not be able to restore group breakpoints.) Corresponds to: the restore command (see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Record Commands	Records commands into the specified file. Corresponds to: the > command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Record Commands + Output	Records commands and their output to the specified file. Corresponds to: the >> command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Stop Recording Commands	Stops recording commands. Use this item to stop recording if it has been started by Record Commands. Corresponds to: the >c command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Stop Recording Commands + Output	Stops recording commands and their output. Use this item to stop recording if it has been started by Record Commands + Output. Corresponds to: the >>c command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Playback Commands	Plays back commands recorded in the selected file. Corresponds to: the < command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Windows Menu

The **Windows** menu is used to jump between all of the windows created by MULTI. If you choose an entry in the **Windows** menu, the corresponding window is brought to the front.

The following are menu items that may appear in the **Windows** menu:

Menu Item	Meaning
<i>debug window</i>	Gives focus to a debugging-related window that has been launched from the local executable.
<i>application name</i>	Gives focus to the main Debugger window for the local <i>application</i> .
<i>IDE tool</i>	Opens a submenu that lists windows corresponding to the <i>IDE_tool</i>. The windows listed in this submenu might be created from the local execution or from other executions. If only one MULTI Launcher is open, that tool does not have a submenu; it has a menu entry instead.
Misc	Opens a submenu listing other window types that do not fall into any of the above categories.

The Help Menu

The table below describes the selections available from the **Help** menu in the main Debugger window.

Menu Item	Meaning
Debugger Help	Opens online help for the Debugger. You may also press F1 to access the online help.
Manuals	Opens the Manuals submenu, which displays a list of all manuals included in your MULTI installation. Choose a manual to open its online help.
Bookmarks	Opens the Bookmarks submenu, which displays all the Help Viewer bookmarks you have created. Bookmarks function across manuals and MULTI sessions. Select a bookmark to display the bookmarked page in online help. Select Manage Bookmarks to open a window that allows you to modify your bookmarks.
About MULTI	Opens the About the MULTI Debugger dialog box, which contains basic information about MULTI, such as its version and copyright information. Corresponds to: the about command (see Chapter 9, “Help and Information Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
License Info	Opens the Active Licenses dialog box, which displays the licenses in use. Corresponds to: the aboutlic command (see Chapter 9, “Help and Information Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Debugger Window Toolbar

The Debugger toolbar contains buttons that allow you to access the most commonly used features of the Debugger. Placing the mouse pointer over a button on the toolbar displays a tooltip with a description of the button's function. The following table gives a full description of each button and its command equivalent. For instructions about how to add or remove buttons from the toolbar, see “Adding, Removing, and Rearranging Toolbar Buttons” on page 698.

Button	Effect
 Go Back	Runs backward to the previous breakpoint or, if no previous breakpoint exists, to the first instruction in the trace data. Corresponds to: Alt+F5 and the bc command. For information about this command, see “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step Up	Steps back up to the caller of the current function. Corresponds to: Alt+F9 and the bcU command. For information about this command, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Previous	Steps back one line. If the TimeMachine Debugger is in source display mode, clicking this button causes the TimeMachine Debugger to step back a single source line. If the TimeMachine Debugger is in an assembly mode, it steps back a single machine instruction. For more information about display modes, see “Source Pane Display Modes” on page 23. Corresponds to: Alt+F10 and the bprev command. For information about this command, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step Back	Steps back one line, stepping into a function if the previous line is in a different function. Corresponds to: Alt+F11 and the bs command. For information about this command, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step (into Functions) on Selected Items	For items selected in the target list, executes one statement. If the statement is a function call, it steps into the called function. When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to: Debug → Step (into Functions) on Selected Items, F11, and the s command. For information about this command, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Next (over Functions) on Selected Items	For items selected in the target list, executes until the next statement of the current function (that is, steps over function calls). When in interlaced source/assembly mode, executes until the next machine instruction of the current function. Corresponds to: Debug → Next (over Functions) on Selected Items, F10, and the n command. For information about this command, see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Return on Selected Items	For items selected in the target list, continues to the end of the current function, and stops in the calling function after returning to it. Corresponds to: Debug → Return on Selected Items, F9, and the cU command. For information about this command, see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Go on Selected Items	For items selected in the target list, begins execution of the program. If the process is stopped, this button continues execution. Corresponds to: Debug → Go on Selected Items, F5, and the c command. For information about this command, see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Halt on Selected Items	For items selected in the target list, halts program execution. Corresponds to: Debug → Halt on Selected Items and the halt command. For information about this command, see “Halt Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Restart	Restarts the process with the same arguments as before. If you click this button while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This button may not be available for relocatable modules. Corresponds to: Debug → Restart, Ctrl+Shift+F5, and the restart command. For information about this command, see “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Prepare Target	Automatically prepares your target or opens the Prepare Target dialog box, which allows you to download your program, flash your program, or verify that your program is present on the target. For more information, see “Preparing Your Target” on page 108. This button may not be available for relocatable modules. Corresponds to: Debug → Prepare Target and the prepare_target command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Reset	Resets the target board. This button is only available when you are connected to a target that supports the reset command. Corresponds to: the reset command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Reload Symbols	Reloads the current executable. Corresponds to: the debug command. For information about this command, see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Assembly	Toggles between displaying source code only and source interlaced with assembly code. This button appears to be pushed down if any assembly instructions are displayed. Corresponds to: View → Display Mode → Source Only, View → Display Mode → Interlaced Assembly, and the assem command. For information about this command, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Program Counter	Displays the current program counter (PC) in the source pane. Corresponds to: View → Navigation → Current PC and the E command. For information about this command, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Upstack	Displays the function up one stack frame. Hold this button down to view a menu showing the entire call stack. Corresponds to: View → Navigation → UpStack, Ctrl++, and the E+ command. For information about this command, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Downstack	Displays the function down one stack frame. Hold this button down to view a menu showing the entire call stack. Corresponds to: View → Navigation → DownStack, Ctrl+–, and the E- command. For information about this command, see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Call Stack	Displays the call stack in the Call Stack window. See also “Viewing Call Stacks” on page 388. Corresponds to: View → Call Stack and the callsvie command. For information about this command, see Chapter 5, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Breakpoints	Opens a Breakpoints window in which you can add and edit breakpoints. See “Viewing Breakpoint and Tracepoint Information” on page 128. Corresponds to: View → Breakpoints and the breakpoints command. For information about this command, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Memory	Opens a Memory View window. See Chapter 15, “Using the Memory View Window” on page 323. Corresponds to: View → Memory and the memview command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Registers	Opens a Register View window displaying machine registers. See Chapter 13, “Using the Register Explorer” on page 253. Corresponds to: View → Registers and the regview command. For information about this command, see “Register Commands” in Chapter 14, “Register Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Locals	Opens a Data Explorer displaying local variables. If the current function is a C++ instance method, the Data Explorer displays the <code>this</code> pointer as well. Corresponds to: View → Local Variables, the view \$locals\$ command, and the localsview command. For information about these commands, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Connect	Connects to a target. If the Debugger is not connected, pressing this button opens the Connection Chooser dialog box, which allows you to connect to a target board or simulator. Corresponds to: Target → Connect, F4, and the connect command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Disconnect	Disconnects from a target board or simulator. Corresponds to: Target → Disconnect from Target and the disconnect command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Edit	Opens an Editor window at the currently viewed location in the source pane. Corresponds to: Tools → Editor and the edit command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 TimeMachine Debugger	Enables/disables TimeMachine mode for the currently selected item. This button is only available if you have collected trace data. See Chapter 19, “Analyzing Trace Data with the TimeMachine Tool Suite” on page 401. Corresponds to: TimeMachine → TimeMachine Debugger and the timemachine command. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 OSA Explorer	Opens the OSA Explorer on the RTOS running on the target. This button is only available for some RTOSes. Corresponds to: View → OSA Explorer and the osaexplorer command. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Dump and Show Events	Dumps the event log and displays events in the MULTI EventAnalyzer. This button is only available for some RTOSes.

Button	Effect
 MULTI EventAnalyzer	Launches the MULTI EventAnalyzer. This button is only available for some RTOSes. Corresponds to: Tools → MULTI EventAnalyzer
 MULTI ResourceAnalyzer	Launches the MULTI ResourceAnalyzer. This button is only available if you are using INTEGRITY. Corresponds to: Tools → MULTI ResourceAnalyzer
 INTEGRITY OSA Object Viewer	Launches the OSA Object Viewer on the object currently selected in the target list. If multiple items are selected in the target list, the OSA Object Viewer displays information for the entire INTEGRITY target. This button is visible only if you are debugging a run-mode connection and using INTEGRITY version 10 or later. Corresponds to: the osaview -context command if one item is selected in the target list. Corresponds to: the osaview command if multiple items are selected in the target list. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Close Debugger	Quits the MULTI Debugger. If you are debugging a process, MULTI gives you the choice to Quit and kill process or Detach from process. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons. To access this option, select Config → Options → General Tab. Corresponds to: File → Close Debugger Window, Ctrl+Q, and the quit window command. For information about this command, see Chapter 2, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

For information about the buttons that appear to the right of the **Status** column, see “Repositioning and Hiding the Target List” on page 15.

Configuring the Debugger Toolbar

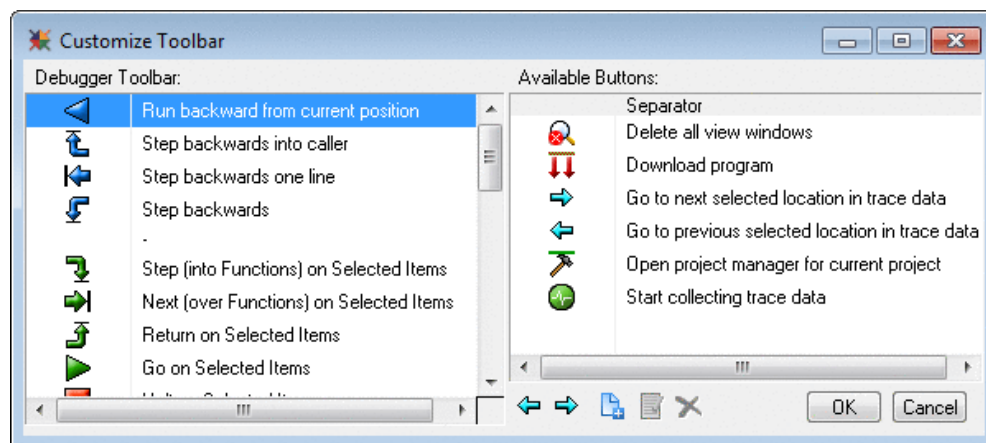
The toolbar appears just below the menu bar in the main Debugger window and displays the default buttons (documented in the previous section) with icon labels. As described next, you can add or remove toolbar buttons and modify the placement of buttons.

Adding, Removing, and Rearranging Toolbar Buttons

You can use the **Customize Toolbar** window to rearrange the order of buttons on the Debugger toolbar, add pre-defined and custom buttons, and delete buttons.

To open the **Customize Toolbar** window, do one of the following:

- Select **Config** → **Customize Toolbar**.
- In the Debugger command pane, enter the **customizetoolbar** command. For information about this command, see “Button, Menu, and Mouse Commands” in Chapter 6, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.



The pane on the left side of the window—**Debugger Toolbar**—displays all the buttons that may currently appear on your toolbar. If your toolbar does not contain all the buttons located in the **Debugger Toolbar** pane, your current debugging environment does not support them. For information about these buttons, see “The Debugger Window Toolbar” on page 691. The pane on the right side of the window—**Available Buttons**—displays a comprehensive set of pre-defined buttons

that you can add to your toolbar. For a description of these buttons, see the next section.

The following list provides instructions for using the **Customize Toolbar** window.

- To rearrange the order of buttons on the toolbar — In the **Debugger Toolbar** pane, drag a button to its new position.
- To delete a button from your toolbar — Drag the button from the left side of the window (**Debugger Toolbar**) to the right (**Available Buttons**), or select the button in the **Debugger Toolbar** pane and click **Remove button** (.
- To add a pre-defined button to your toolbar — Drag the button from the right side of the window (**Available Buttons**) to the left (**Debugger Toolbar**), or select the button in the right side of the window and click **Add new button** (). (You can add back deleted buttons in this way.)
- To add a custom button to your toolbar:
 1. Click **Add Custom Button** (.
 2. In the dialog box that appears, select a button icon from the **Icon** pane. Fill in the **Button Name** text field with the text you want to appear in a tooltip when the cursor moves over the button, the **Command** text field with the command you want to execute when you click the button, and the **Help String** text field with the text you want to appear at the bottom of the Debugger window when the cursor moves over the button.
 3. Click **OK**.
 4. Your new button appears in the **Available Buttons** pane. Drag it to the desired location in the **Debugger Toolbar** pane.
 5. Click **OK**.
- To edit a custom button:
 1. Drag the button to the **Available Buttons** pane if it is not already there.
 2. Click **Edit Custom Button** () and make the desired changes.
 3. Click **OK**.

4. To add the button to the toolbar, drag it to the desired location in the **Debugger Toolbar** pane.

5. Click **OK**.

Modifications that you make to the toolbar through the **Customize Toolbar** window are saved by default.

You can edit the **button.odb** configuration file to make the same toolbar changes that are detailed above. The **button.odb** file is located in:

- Windows 7/Vista — *user_dir*\AppData\Roaming\GHS\v6
- Windows XP — *user_dir*\Application Data\GHS\v6
- Linux/Solaris — *user_dir*/.ghs/v6

Within the current MULTI session, you can reprogram any of the pre-defined Debugger toolbar buttons except the **Close Debugger** button (✗). You can define a button's name, command string, corresponding icon (optional), etc. by entering the **debugbutton** command with appropriate arguments. For more information about this command, see Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

Pre-Defined Buttons

There are a number of pre-defined buttons that you can add to the Debugger toolbar via the **Customize Toolbar** window (see “Adding, Removing, and Rearranging Toolbar Buttons” on page 698). The following table contains descriptions of these buttons.

Button	Effect
Separator	Adds a vertical bar between adjacent icons on the toolbar.
 Delete all view windows	Closes all Data Explorers and Register View, Call Stack, Breakpoints, Memory View, and Browse windows. Corresponds to: View → Close All Views and the viewdel command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Download Program	Loads the current program into the target system's memory if the target supports and was configured with a dynamic loader (for example, the LoaderTask on INTEGRITY). Corresponds to: the load -setup command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Go to next selected location in trace data	Returns to the next location in the trace navigation history. This button is only available if you have previously clicked the Go to previous selected location in trace data button (described next). Corresponds to: the trace history + command. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Go to previous selected location in trace data	Returns to the previous location in the trace navigation history. This can be useful if you want to undo an action (such as running or stepping backwards in the TimeMachine Debugger) that brought you to an unexpected location in your source code. Corresponds to: the trace history - command. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Open project manager for current project	Opens a Project Manager on the current project. Corresponds to: Tools → Project Manager and the builder command. For information about this command, see Chapter 4, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Start collecting trace data	Toggles collection of trace data. The button appears to be pushed down while trace data is being collected. Corresponds to: TimeMachine → Enable/Disable Trace and trace toggle command. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Target List Shortcut Menu

The following table describes the menu items that are available in the target list right-click menu. These menu items are context sensitive; they may not all appear every time you open this menu.

Menu Item	Meaning
Open in New Window	Opens a new Debugger window on the selected item. See also “Opening Multiple Debugger Windows” on page 14.
Prepare Target	Opens the Prepare Target dialog box from which you can choose to download the selected program, flash it, or verify its presence on the target. This menu item opens the Prepare Target dialog box even if you saved an action for the selected program by clicking Automatically use these settings for this program next time in the dialog box the last time you prepared the target. For more information, see “Preparing Your Target” on page 108.
Use Connection	Opens a submenu with the following entries: • [Status] Connection_Name — Associates the current executable with the specified connection. Only compatible, currently active connections are listed. Status is one of: Available (the current executable can use this connection), Current (the current executable is using this connection), or Full (another executable is using this connection exclusively). • Stop Using Current Connection — Disassociates the selected executable from the connection it is currently associated with. Selecting this menu item does not disconnect from the target debug server. • Create New Connection — Opens the Connection Chooser and attempts to use the connection selected. For more information, see “Associating Your Executable with a Connection” on page 105.
Remove	Removes the currently selected executable from the target list and terminates the process. If the executable is part of the last remaining group of related items, the main Debugger window and its child windows close.
Disconnect from Target	Disconnects from the target debug server.

Menu Item	Meaning
Load New Module	Opens a file chooser from which you can select a module to load on the target. This option is only available if you are connected to INTEGRITY in run mode and if the target supports and was configured with the dynamic loader (the LoaderTask).
Load Module	Loads the selected module on the target. This option is only available if you are connected to INTEGRITY in run mode and if the target supports and was configured with the dynamic loader (the LoaderTask).
Unload Module	Unloads the selected module from the target. This option is only available if you are connected to INTEGRITY in run mode. It may not be available on rtserv connections (INTEGRITY version 5 and earlier) if all the tasks in the module have exited.
Reload Module	Unloads, then loads the selected module. This option is only available if you are connected to INTEGRITY in run mode. It may not be available on rtserv connections (INTEGRITY version 5 and earlier) if all the tasks in the module have exited.
Trace	Toggles trace collection for the AddressSpace in which the selected task resides. This option is only available when you connect to INTEGRITY in run-mode. For more information, see “Collecting Operating System Trace Data” on page 406.
Resume	Starts or continues execution of the selected executable.
Halt	Halts the selected process.
Kill	Kills the selected process. If you are debugging a stand-alone program on a hardware target (i.e. not a simulated target), you cannot kill a running process. When the program reaches its exit point, the process terminates and the program is left halted at the exit system call.
Attach	Attaches the Debugger to the currently selected executable. This option is only available on some run-mode targets.

Source Pane Shortcut Menus

When you right-click in the source pane, a shortcut menu appears.



Note

This is the default behavior. If you have configured a right-click to perform other functions, the shortcut menu will not appear.

This menu is context sensitive and depends on the object (such as a procedure, type, or variable) you right-click. Some menu items may appear dimmed, indicating that they are unavailable in the current context.

In the following shortcut menu discussion, the term “right-clicked line” refers to the source line where the right-click occurred.

Common Shortcut Menu Options

The following items appear in most shortcut menus available from the Debugger source pane.

Menu Item	Meaning
Display Program Visualizations	Opens a Graph View window (see “The Graph View Window” on page 247) displaying a graph with one tab for each defined custom <i>view description</i> . This is equivalent to the dataview command. For more information about how to define and load <i>view descriptions</i> , see Appendix E, “Creating Custom Data Visualizations” on page 741. For information about the dataview command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Edit File	Opens an editor window with the cursor on the right-clicked line of the current source file.
Search for Selected Text	Searches for the selected text in open files and in the source code of your current executable. Results appear in a new Search in Files Results window.
Properties	Opens a dialog box displaying basic information about the current source line (for example, the filename, language, and line number), the current target connection, and also basic information about the right-clicked object, if available (for example, its size, address, and whether its scope is static or global).

The Source Line Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a blank part of a source line or an unknown object.

Menu Item	Meaning
Run To This Line	Sets a temporary breakpoint on the line and then runs the program to it. The process stops on the line only if it attempts to execute that line. This menu item is only available if the right-clicked line contains executable code.
Change PC To This Line	Changes the program counter to the first executable instruction of the line. This menu item is only available if the right-clicked line contains executable code and the process is stopped within the procedure containing the right-clicked line.
Trace	Opens a submenu containing trace options. For a description of this submenu, see later in this section. This menu item is only available if you are connected to a trace-enabled target.
Set Breakpoint	Opens a submenu that allows you to set various types of breakpoints on the line. For a description of this submenu, see later in this section. This menu item is only available if the right-clicked line contains executable code and does not already have a breakpoint set on it.
Edit Breakpoint	Opens the Software Breakpoint Editor , which allows you to edit the software breakpoint set on the line. This menu item is only available if a software breakpoint is set on the right-clicked line.
Remove Breakpoint	Removes the software breakpoint from the line. This menu item is only available if the right-clicked line contains executable code and a breakpoint has already been set on it.
Enable/Disable Breakpoint	Toggles (enables or disables) the software breakpoint located on the line. This menu item is only available if a breakpoint is set on the right-clicked line.
Move Breakpoint	Allows you to move the software breakpoint located on the line to another line in the same function. This menu item is only available if one software breakpoint is set on the right-clicked line. For more information, see “Moving Software Breakpoints” on page 132.
Create Note	Allows you to add a Debugger Note at the line's location. For more information, see Chapter 10, “Using Debugger Notes” on page 173.
Edit Note	Allows you to edit the Debugger Note set at the line's location. For more information, see Chapter 10, “Using Debugger Notes” on page 173.

Menu Item	Meaning
Remove Note	Removes the Debugger Note from the line. For more information, see Chapter 10, “Using Debugger Notes” on page 173.
Browse Includers Of This File	Opens a Browse window displaying those files that include the current file. For more information, see “Browsing Source Files General Information” on page 231.
Browse Files Included By This File	Opens a Browse window displaying those files that are included by the current file. For more information, see “Browsing Source Files General Information” on page 231 .
Display Interlaced Assembly/Source Only	Toggles the display between source-only mode and interlaced assembly mode.

The following table describes the selections available from the **Trace** submenu.

Menu Item	Meaning
Trace Around This Line	Sets the trigger event to be any execution of the selected line and enables trace. See “Configuring Trace Directly from MULTI” on page 491.
Browse Traced Executions	After collecting trace data, displays a list showing each time the selected source line was executed and stored in trace data. See “The Trace Instruction Browser” on page 456.

The following table describes the selections available from the **Set Breakpoint** submenu.

Menu Item	Meaning
Set Breakpoint	Sets a software breakpoint. See also the b command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book
Set And Edit	Opens the Software Breakpoint Editor window, which allows you to specify and set a software breakpoint. See the editswbp command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Menu Item	Meaning
Set Jump Breakpoint	Queries for a location to jump to when the breakpoint is hit and sets a jump breakpoint. A jump breakpoint executes the g location;resume; commands when it is reached. For information about the g command, see “General Program Execution Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. For information about the resume command, see Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Set Any Task Breakpoint	Sets a software breakpoint which can be hit by any task. This option is available only when connected to a target which supports this type of breakpoint. See the sb command in Chapter 3, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Breakdot Shortcut Menu

In addition to certain menu options listed in “The Source Line Shortcut Menu” on page 705, the following menu options are available in the shortcut menu that appears when you right-click a breakdot. Tracepoint menu options appear if your connection supports tracepoints.

Menu Item	Meaning
Set Hardware Breakpoint	Sets a hardware breakpoint on the line. This menu item is only available if the right-clicked line contains executable code and does not already have a breakpoint set on it.
Edit Hardware Breakpoint	Opens the Hardware Breakpoint Editor , which allows you to edit the hardware breakpoint set on the line. This menu item is only available if a hardware breakpoint is set on the right-clicked line.
Remove Hardware Breakpoint	Removes the hardware breakpoint from the line. This menu item is only available if the right-clicked line contains executable code and a hardware breakpoint has already been set on it.
Enable/Disable Hardware Breakpoint	Toggles (enables and disables) the hardware breakpoint located on the line. This menu item is only available if a hardware breakpoint is set on the right-clicked line.
Set Tracepoint	Opens the Tracepoint Editor , which allows you to set a tracepoint on the line.
Edit Tracepoint	Opens the Tracepoint Editor , which allows you to edit the tracepoint set on the line. This menu item is only available if a tracepoint is set on the right-clicked line.

Menu Item	Meaning
Remove Tracepoint	Removes the tracepoint from the line. This menu item is only available if a tracepoint is set on the right-clicked line.
Enable/Disable Tracepoint	Toggles (enables and disables) the tracepoint located on the line. This menu item is only available if a tracepoint is set on the right-clicked line.

The Procedure Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a procedure.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the procedure's definition in the source pane.
Go To Declaration	Displays the source code, if available, for the procedure's declaration in the source pane.
Browse References	Shows the procedure's cross references in a Browse window. For more information, see “Browsing Cross References” on page 236.
Browse Other	For a description of this submenu, see later in this section.
Trace	This is available only if connected to a trace-enabled target. For a description of this submenu, see later in this section.
Step Into This Function	Attempts to step through the next source line, halting if execution enters the selected procedure. This is available only when right-clicking procedure calls on the currently executing line.
Edit Definition	Opens an editor window centered on the procedure's definition, if the appropriate source debugging information is available.

The following table lists the entries in the **Browse Other** submenu.

Menu Item	Meaning
Browse Callers	Opens a Browse window displaying the callers of the procedure.
Browse Callees	Opens a Browse window displaying the callees of the procedure.
Browse Static Calls	Opens a Tree Browser window displaying the static call graph of the procedure (which shows its callers and callees and their callers and callees). See “The Tree Browser Window” on page 239.

The following table lists the entries in the **Trace** submenu.

Menu Item	Meaning
Trace This Function	Enables trace only when executing the selected function. Tracing will not occur when executing subfunctions.
Trace Function Interval	Allows you to create conditions that enable and disable trace based on executions in specific functions. See “Specifying a Trace Function Interval” on page 492 for more information.
Browse Traced Calls	After you have collected trace data, displays a list of each call to this function in the trace data. See “The Trace Call Browser” on page 457.

For more information about collecting trace data, see “Enabling and Disabling Trace Collection” on page 405.

The Variable Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a variable (local, parameter, global, or static). If you can set a hardware breakpoint on the variable, the hardware breakpoint menu options listed in “The Breakdot Shortcut Menu” on page 707 appear as well.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the variable's definition in the source pane.
Go To Declaration	Displays the source code, if available, for the variable's declaration in the source pane. This is available only for global and static variables.
Browse References	Shows the variable's cross references in a Browse window. See “Browsing Cross References” on page 236 for more information.
Trace	This is available only if you are connected to a trace-enabled target. For a description of this submenu, see later in this section.
View Value	Opens a Data Explorer displaying the value of the variable.
Graph Data	If the variable matches a custom data description of the data type <code>graph</code> , opens a Graph View window displaying a graph of objects based on the data description. See Appendix E, “Creating Custom Data Visualizations” on page 741 and “The Graph View Window” on page 247.
Edit Definition	Opens an editor window centered on the variable's definition, if the appropriate source debugging information is available.

The following table lists the entries in the **Trace** submenu.

Menu Item	Meaning
Trace Around Data Access	Sets the trigger event to be any access of the selected global variable. This option is available only when connected to a trace-enabled target. For more information, see Chapter 20, “Advanced Trace Configuration” on page 479.
Browse Traced Accesses	After you have collected trace data, displays a list of each access to this global or static variable in the trace data. Note that this only displays accesses of the first address of the variable. If the variable is a struct or class with multiple member variables, only accesses of the first member variable are displayed. For more information, see “The Trace Memory Browser” on page 452.

The Type Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a type.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the type's definition in the source pane.
Browse References	Shows the type's cross references in a Browse window. See “Browsing Cross References” on page 236 for more information.
Browse Other	For a description of this submenu, see later in this section. This is available only if the type is a C++ class.
View Type	Opens a Data Explorer displaying the type's definition.
Edit Definition	Opens an editor window centered on the type's definition, if the appropriate source debugging information is available.

The following table lists the entries in the **Browse Other** submenu.

Menu Item	Meaning
Browse Superclasses	Shows the type's super-classes in a Browse window.
Browse Subclasses	Shows the type's sub-classes in a Browse window.
Browse Class Hierarchy	Opens a tree browser window to show class hierarchy information. See “The Tree Browser Window” on page 239.

The Type Member Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a type's member (such as the member reference of a struct variable, an enumeration member, or a member name in a class declaration).

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the member's definition in the source pane.
Browse Member References	Shows the member's cross references in a Browse window. See “Browsing Cross References” on page 236 for more information.
View Value	Opens a Data Explorer displaying the value of the member.
Graph Data	If the member variable matches a custom data description of the data type <code>graph</code> , opens a Graph View window displaying a graph of objects based on the data description. See Appendix E, “Creating Custom Data Visualizations” on page 741 and “The Graph View Window” on page 247.
Edit Definition	Opens an editor window centered on the member's definition, if the appropriate source debugging information is available.

The C/C++ Macro Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 704, the following menu options are available in the shortcut menu that appears when you right-click a C or C++ macro.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the macro's definition in the source pane.

The Command Pane Shortcut Menu

The following items appear in the shortcut menu when you right-click in the Debugger command pane (or the target pane, I/O pane, serial terminal pane, Python pane, or traffic pane).

Menu Item	Meaning
Cut	Cuts the current selection to the clipboard. Note that only the current input line may be cut.
Copy	Copies the current selection to the clipboard.
Paste	Pastes text from the clipboard into the current pane. The text will be treated as input.
Command Pane	Switches to the command pane.
Target Pane	Switches to the target pane.
IO Pane	Switches to the I/O pane.
Serial Pane	Switches to the serial terminal pane.
Python Pane	Switches to the Python pane.
Traffic Pane	Switches to the traffic pane.
Interleaved Output	When selected, output from each pane is displayed in the command pane in addition to its own pane.
Clear Pane	Clears all text from the current pane.
Show Separate Py Window	Opens the Python pane as a separate window.
Save As	Opens a file chooser in which you can choose the file where you want to save the contents of the current pane.

The Source Pane Search Dialog Box

To control searches in the source pane, open the **Source Pane Search** dialog box in one of the following ways:

- Select **Tools** → **Search**.
- In the command pane, enter the **dialogsearch** command. For information about this command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

For example, to search for a string such as `tree`:

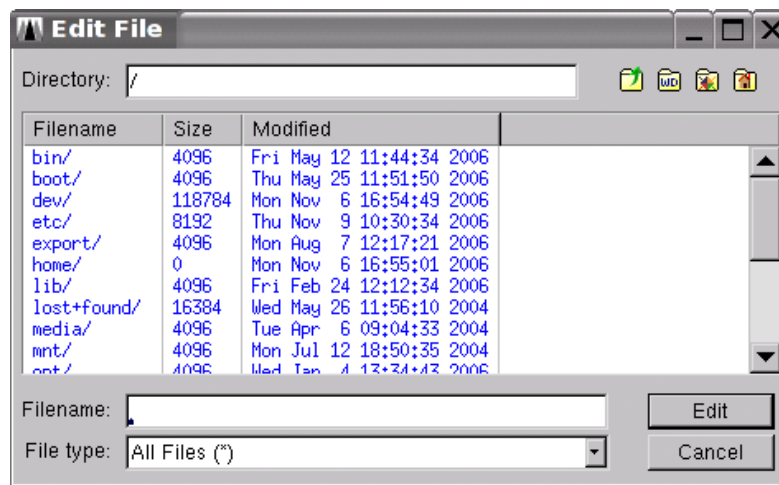
1. Type `tree` in the search text field.
2. Click **Find** to search for the next occurrence of the string.

The toggle buttons in this window are the same as those in the Editor's search dialog box.

To initiate or repeat searches without the search dialog box, you can also use the **Ctrl+F** or **Ctrl+B** key sequences in the command pane.

The File Chooser Dialog Box (Linux/Solaris)

A file chooser dialog box opens if you initiate an action (using a menu, button, command, or shortcut) for which a file must be specified, but you have not yet specified a file. A chooser also opens if you click a **Browse** or  button on a dialog box. A sample Linux/Solaris file chooser follows. (For information about Windows file choosers, see your Windows documentation.)



The title bar of the file chooser will vary depending on the action you are performing. The following table describes the main elements of the file chooser.

Item	Meaning
Directory	Displays the directory whose contents are shown in the File List . Type in a new directory name and press Enter to display a different directory.

Item	Meaning
Directory Buttons	With this set of buttons, you can quickly go to different important directories. The buttons that appear are:  — Up One Directory from the current directory.  — Jumps to the current Working Directory.  — Jumps to the IDE Installation Directory.  — Jumps to the User Home Directory.
File List	Below the directory text field is the file list. To enter a directory, double-click the directory. To choose a file, single-click the filename. You can click any column header to sort the list in ascending or descending order. If multiple files are allowed for the present operation (for example, Open is selected in the editor menu), use the Shift key to select a consecutive list of files; use the Ctrl key to select non-consecutive files.
Filename	Type a filename or directory name into this text field. As you type in this field, the selection in the file list will change to reflect the closest match. If you type a directory name and press Enter, or follow the directory name by a slash (/), the file list will change to the specified directory.
File type	If you select a file type, the File List will only display files with suffixes that match the selected file type.
Action buttons	There are two buttons in the lower right corner of the file chooser window. The upper button displays the action that will occur, such as Edit (in the example above), OK, or Open. The lower button is the Cancel button, which closes the window without taking any action.

Appendix B

Keyboard Shortcut Reference

Contents

Main Debugger Window Shortcuts	716
Source Pane Shortcuts	717
Command Pane Shortcuts	718

This appendix contains descriptions of the Debugger's default keyboard shortcuts.

Main Debugger Window Shortcuts

The following shortcuts are available from the Debugger window.



Note

If you have clicked in the source pane or command pane, these shortcuts may have different effects. See “Source Pane Shortcuts” on page 717 and “Command Pane Shortcuts” on page 718.

Shortcut	Effect
F1	Displays help about the current window. Corresponds to: the help command (see Chapter 9, “Help and Information Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F4	Connects to a target. If the Debugger is not connected, pressing this button opens the Connection Chooser dialog box, which allows you to connect to a target board or simulator. Corresponds to:  and the connect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F5	Begins execution of the program. If the process is stopped, it continues execution. Corresponds to:  and the c command (see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Ctrl+Shift+F5	Restarts the process with the same arguments as before. If you use this shortcut while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This shortcut may not be available for use with relocatable modules. Corresponds to:  and the restart command (see “Run Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Shortcut	Effect
F9	Continues to the end of the current procedure, and stops in the calling procedure after returning to it. Corresponds to:  and the cU command (see “Continue Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F10	Executes until the next statement of the current procedure (that is, steps over procedure calls). When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to:  and the n command (see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F11	Executes one statement. If the statement is a procedure call, it is stepped into. When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to:  and the s command (see “Single-Stepping Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Ctrl++	Displays the procedure up one stack frame. Corresponds to:  and the E + command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Ctrl+-	Displays the procedure down one stack frame. Corresponds to:  and the E - command (see Chapter 8, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Source Pane Shortcuts

The following shortcuts are available from the Debugger window's source pane.

Shortcut	Effect
PageUp	Scroll the source pane up by one page.
PageDown	Scroll the source pane down by one page.
Shift+UpArrow	Scroll the source pane up by one line.
Shift+DownArrow	Scroll the source pane down by one line.
Ctrl+F	Search forward in the source pane.

Shortcut	Effect
Ctrl+B	Search backward in the source pane.
Esc	Abort current operation or halt process.

Command Pane Shortcuts

The following shortcuts are available from the Debugger window's command pane.

Shortcut	Effect
UpArrow	If nothing is entered on the command line, retrieves the previous command from the command history, moving back in the list. Press repeatedly to retrieve older commands. If you have already begun typing, MULTI attempts to auto-complete the current string, working backwards through commands that have been entered.
DownArrow	If nothing is entered on the command line, retrieves the next command from the command history, moving forward in the list. Press repeatedly to retrieve more recent commands. If you have already begun typing, MULTI attempts to auto-complete the current string, working forwards through commands that have been entered.
Ctrl+UpArrow	Scrolls up four lines.
Ctrl+DownArrow	Scrolls down four lines.
Ctrl+U	Cuts to the beginning of the current line or the selection.
Tab	Accepts auto-completion of the current word.
Ctrl+P	Attempts to auto-complete the current string, working backwards through the command history for commands beginning with the typed characters.
Ctrl+N	Attempts to auto-complete the current string, working forwards through the command history for commands beginning with the typed characters.
Ctrl+D	Displays a list of possible completions for the current word.

Shortcut	Effect
F6	Cycles between the available panes in the command pane area. For more information, see “The Cmd, Trg, I/O, Srl, Py, and Tfc Panes” on page 26. Cycles between the command, I/O, target, and serial terminal panes. For more information, see “The Cmd, Trg, I/O, Srl, Py, and Tfc Panes” on page 26 .
F12	Displays a pop-up menu showing all Debugger Notes set in the program.
Click with the right mouse button	Opens a shortcut menu for the command pane. For a full description of the shortcut menu, see “The Command Pane Shortcut Menu” on page 712.

Appendix C

Command Line Reference

Contents

Using a Specification File	727
----------------------------------	-----

The following command line options can be used when starting MULTI from the command line, as described in “Starting the Debugger in GUI Mode” on page 6.

-- args Passes all the arguments that follow the double dash to the program being debugged as though the arguments were set with the setargs command. For more information, see the setargs command in “General Program Execution Commands” in Chapter 13, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
-build file Opens the Project Manager on default.gpj in your current working directory and starts building <i>file</i> , which may be anything you can build within default.gpj . The Debugger does not open when you use this option, so you should not specify a program to debug.
-c file -config file -configure file Configures MULTI using the information in the configuration file <i>file</i> . For more information, see “Creating and Editing Configuration Files” in Chapter 7, “Configuring and Customizing MULTI” in the <i>MULTI: Managing Projects and Configuring the IDE</i> book.
-C core_file Linux/Solaris only Specifies the core file, where <i>core_file</i> is a Linux/Solaris core image of the program to be debugged. For more information, see “Core File Debugging (Linux/Solaris only)” on page 117.
-cmd commands Runs the specified commands when the Debugger is up.

-connect[=*target* | =*connection_method_name*]

Connects to the target debug server specified by *target*; connects using the specified Connection Method; or, if neither a target nor a Connection Method is specified, opens the **Connection Chooser**.

For more information about connecting to targets, see Chapter 3, “Connecting to Your Target” on page 39. For information about the **connect** command, which corresponds to this option, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Note: The Debugger ignores the deprecated `mode` argument in Connection Methods that specify it. To remove the `mode` argument from a MULTI 4 Connection Method, edit and save the Connection Method in MULTI 6. For more information, see the **connect** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

-connectfile *file*

Specifies that connections should be loaded from the file *file*.

For more information, see “Creating and Managing Connection Files” on page 52.

-D

Ignores all currently specified alternative directories.

See also the **-I** command line option later in this table and the **source** command in “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

-data *offset*

Sets the default offset for all data addresses to *offset*, which is assumed to be in decimal unless preceded by `0x`. Do not set *offset* to `-1`. This option is only useful when debugging programs built with position-independent data; it should not be used in other kinds of programs.

-display *disp*

Linux/Solaris only

Specifies that the Debugger will open in the alternative display *disp*.

-e *entry*

Specifies the entry label. The default is `main`. In C++ programs, only non-mangled names such as those declared with `extern "C"` may be specified for *entry*.

-E *file*

Opens the Debugger on multiple files. Use this option for each file after the first that you want to debug. For example, if you want to debug **foo**, **bar**, and **baz**, use the command:

```
multi foo -E bar -E baz
```

This opens a single Debugger window and lists **foo**, **bar**, and **baz** in the target list of that window.

The maximum number of files you can specify is 256. The first file you specify, which is not preceded by **-E**, is included in this maximum number.

-h

Displays a concise description of all command line options. This is equivalent to the **-usage** option (below).

-H**-help**

Opens the MULTI Help Viewer on the *MULTI: Debugging* book.

-I *directory*

(This option is a capital letter “i.”)

Names an alternative directory for the Debugger to search for files in. You can specify multiple alternative directories by using multiple **-I *directory*** arguments. The Debugger searches alternative directories in the order given. If it does not find a file in the specified alternative directory or directories, it also searches the current directory.

See also the **-D** command line option earlier in this table and the **source** command in “General Configuration Commands” in Chapter 6, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

-m *file*

Sets *file* as the default specification file.

For more information, see “Using a Specification File” on page 727.

-nocfg

Causes MULTI to ignore all **.cfg** files on startup.

-nodisplay**Linux/Solaris only**

Specifies that the Debugger will open in non-GUI mode.

For more information, see “Starting the Debugger in Non-GUI Mode (Linux/Solaris only)” on page 8.

-norc

Causes MULTI to ignore all **.rc** script files upon startup except for those specified with **-rc** (below).

For more information, see “Using Script Files” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

-noshared

Indicates that MULTI should not debug shared libraries. This option is relevant only for targets that support shared libraries.

See also the `DEBUGSHARED` system variable in “System Variables” on page 310.

-nosplash

Prevents MULTI from displaying the startup banner.

-osa *osa_name* [#cfg=*configuration_file*] [#lib=*library_name*] [#log=*log_file*]

Starts MULTI with freeze-mode debugging and OS-awareness enabled. *osa_name* specifies an object structure-aware debug package. *configuration_file* specifies the configuration file for freeze-mode debugging and OS-awareness, *library_name* specifies the library containing the OSA integration package, and *log_file* specifies a file in which to log the interaction between MULTI and the OSA integration package.

For more information, see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 607.

-p *file*

Runs the commands in the playback file *file* on startup.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-P *pid***Native targets only**

Attaches to the process with the process identification number *pid*. The maximum number of processes you can specify is 256.

-preload *module*

Downloads *module* after connecting to an operating system target, which must be specified elsewhere on the command line.

This option is only valid for certain operating system targets. Additionally, the target must support and be configured with a dynamic loader (for example, the LoaderTask on INTEGRITY).

-r *file*

Records commands to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-R *file*

Records commands and output to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-rc *file*

Reads the command script *file* when the first Debugger window appears. The file is read after the global and user script files.

For more information, see “Using Script Files” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

-RO *file*

Records output to the playback file *file*. (Commands are not recorded.)

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-run *debug_server* [-timeout=*num*] --

Connects to *debug_server*, runs *program* in the Debugger (see “Starting the Debugger in GUI Mode” on page 6), runs any commands specified on the command line or in any relevant *.rc* scripts, and then exits.

-timeout specifies the number of seconds until the operation times out. On Windows and Linux, the default is 600; on Solaris, the default is 2400. The maximum value of *num* is $2^{31}/1000$. If *num* is -1, the operation never times out.

This option should be specified after other options and before *program* (see “Starting the Debugger in GUI Mode” on page 6).

-servertimeout *time*

Sets the default debug server timeout to *time* seconds.

-socket *port*

Creates a socket connection on the port *port*. This socket connection can be used to send Debugger commands and receive Debugger output.

For more details, see the **socket** command in Chapter 2, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

-text *offset*

Sets the default offset for all text addresses to *offset*, which is assumed to be in decimal unless preceded by 0x. Do not set *offset* to -1. This option is only useful when debugging programs built with position-independent code; it should not be used with other kinds of programs.

-top**Native targets only**

Opens the Process Viewer, which displays a snapshot of the processes running on your native target and allows you to attach to native processes. For more information, see “Viewing Native Processes” on page 390.

-tv file

Specifies the task view configuration file for run-mode debugging.

For more information about task view configuration files, see “Task Group Configuration File” on page 600.

-usage

Displays a concise description of all command line options. This is equivalent to the **-h** option (above).

-V

Prints Debugger version information.

Using a Specification File

You can use a specification file to set up a default set of command line arguments, which are used when you open the Debugger on a specified executable. You can create default command line options for more than one executable in the same specification file.

Your specification file should have an **.mc** extension. To indicate the command line arguments associated with an executable, type the name of the executable followed by a space and the list of command line arguments, all on a single line. If you need to continue a list of arguments on a second line, the second line must begin with a tab. To create a list of commands for another executable, begin a new line and enter the executable name followed by a space and the list of command line options. For example, a specification file that uses separate command line arguments when debugging **foo** and when debugging **bar** might contain the following lines.

Windows:

- `foo -I C:\usr\joebob -I C:\usr\foodir`
- `bar -text 10000 -data 40000`

Linux/Solaris:

- `foo -I /usr/joebob -I /usr/foodir`
- `bar -text 10000 -data 40000`

To use a specification file, pass the **-m file** option and launch the MULTI Debugger from the command line. This results in MULTI searching the specified file for a line that begins with the name of the executable on which you are opening the Debugger. If it finds a line that begins with the executable name, the Debugger uses the specified command line options on that line when it opens on the program. For example, if the specification file **albatross.mc** contains the lines given above, entering the command:

```
multi -m albatross.mc foo
```

has the same effect on your operating system as entering one of the following commands.

- Windows — `multi foo -I C:\usr\joebob -I C:\usr\foodir`
- Linux/Solaris — `multi foo -I /usr/joebob -I /usr/foodir`

Appendix D

Using Third-Party Tools with the MULTI Debugger

Contents

Using the Debugger with Third-Party Compilers	730
Running Third-Party Tools from the Debugger	731
Using MULTI with Rhapsody	732

This appendix describes how to use various third-party tools with the MULTI Debugger.

Using the Debugger with Third-Party Compilers

Green Hills provides Debug Translators that allow you to use the MULTI Debugger to debug applications compiled with third-party compilers. The MULTI Debugger provides superior debugging capabilities by using special debugging information files generated by Green Hills Compilers. If you do not compile with Green Hills Compilers, the Green Hills Debug Translators allow you to convert DWARF or Stabs debugging information generated by third-party compilers into the format required by the MULTI Debugger.

If you have licensed one of the Green Hills Debug Translators, you can configure MULTI to automatically convert DWARF or Stabs debugging information when an executable compiled entirely by a third-party compiler that generates that type of information is loaded into the Debugger. To enable automatic conversion:

1. Select **Config** → **Options** from the Debugger.
2. Select the Debugger tab, and then click **More Debugger Options**.
3. Select either **Translate DWARF debugging information** or **Translate stabs debugging information**, as desired. (For more information about these options, see “The More Debugger Options Dialog” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.)
4. Click **OK**.

For more information, including instructions about what to do if you compiled some of the object files of your executable with a third-party compiler and others with a Green Hills Software compiler, see the documentation about generating debugging information for applications compiled with third-party compilers in the *MULTI: Building Applications* book.

Running Third-Party Tools from the Debugger

By using the Debugger's **shell** command, you can configure menus and buttons to run arbitrary external commands from within the Debugger. For information about the **shell** command, see “Command Manipulation and Macro Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book. For information about defining your own menu items and toolbar buttons to invoke commands, see “Customizing the GUI” in Chapter 7, “Configuring and Customizing MULTI” in the *MULTI: Managing Projects and Configuring the IDE* book.

You can dynamically construct command line arguments by using the `%EVAL{multi_commands}` escape sequence of the **shell** command. Useful *multi_commands* include:

- System variables such as `_SELECTION`, which obtains the current selection in the Debugger source pane
- Commands such as **dialog** that prompt you for information (see “Dialog Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book)

To display the contents of output files produced by third-party tools, issue the **cat file** command. The Debugger displays the contents of *file* in the command pane. For information about the **cat** command, see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

If you have the **make** utility installed on your system, you can use it to build programs from within the MULTI Debugger by issuing the Debugger's **make** command. For information about the **make** command, see “External Tool Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

You can configure the MULTI IDE to invoke the editor of your choice instead of the MULTI Editor. For more information, see “Third-Party Editor Configuration Options” in Chapter 8, “Configuration Options” in the *MULTI: Managing Projects and Configuring the IDE* book.

Using MULTI with Rhapsody

Rhapsody and MULTI are integrated to provide a complete Model Driven Development (MDD) solution for a number of target environments.

Rhapsody provides a UML-based development environment that can generate source files and has many other capabilities not listed here. In contrast, MULTI provides the capabilities to compile that code, download and run it on a target, and debug it at the source level.

Ultimately, the integration between these two tools provides automation and a seamless path from phases such as requirements analysis, UML design, and model-driven development through to code generation, compilation, loading, running, debugging, validation, and testing.

Supported Environments

At the time of writing, supported versions of Rhapsody are:

- 7.5.1
- 7.5.3

The supported host is Windows.

Supported target environments include:

- INTEGRITY 10 and 5
 - Power Architecture, ColdFire, ARM, MIPS, x86



Note

While INTEGRITY 10 is supported, the Rhapsody GUIs only refer to INTEGRITY 5.

Integration Description

The three significant points of integration between Rhapsody and MULTI can be summarized as follows:

- **Build** — Rhapsody can generate source files and the necessary MULTI project files (**.gpj** files) and invoke the build processes automatically.
- **Target** — The Rhapsody *Adapters* (framework libraries) provide the adaptation layer between the generated model and the underlying OS, such as INTEGRITY. The Adapters take into account the Green Hills Compiler and are customized for the underlying OS.
- **Debug** — Rhapsody provides model-level debugging called *Animation*, whereas MULTI provides source-level debugging. While these debugging methods can be used independently and effectively, the integration provides the additional capability to synchronize them.

Installation and Configuration

For information about installing Rhapsody, please refer to Rhapsody's installation guide from IBM Rational. There are no special installation requirements or configuration instructions for MULTI or INTEGRITY.

Rhapsody must be configured to use a particular compiler (such as a Green Hills Compiler) and a particular OS (such as INTEGRITY). During the Rhapsody installation process, you should supply the path to the Compiler directory when prompted for the location of **GHS MULTI 4.x for PPC**. You may supply the path to an INTEGRITY 10 installation when you are prompted for the location of **GHS INTEGRITY 5.x**.

The Rhapsody installer records the location of the Compiler and/or INTEGRITY directories in one or more **.bat** files located in the **Rhapsody\Share\etc** directory (depending upon your target environment and versions of tools). They include **Integrity5Make.bat** and **Integrity5MakefileGenerator.bat**. If you enter the wrong directory during installation, or if the Compiler or INTEGRITY directories are subsequently moved, you may manually correct these files.

The Rhapsody build system for INTEGRITY requires the following system environment variables to be set when you run Rhapsody:

- `INTEGRITY_ROOT` — The INTEGRITY installation directory
- `MULTI_ROOT` — The Compiler installation directory

For further information about configuring Rhapsody to use MULTI and INTEGRITY, please refer to Rhapsody's installation guide from IBM Rational.

Licensing

Standard MULTI and Compiler licenses are required for MULTI and the Green Hills Compilers, respectively. To enable the synchronous debugging functionality, MULTI requires the additional “Rhapsody Integration License.” If you do not already have these licenses, contact your Green Hills sales representative to obtain them.

Additional Notes

Much of the information in the following sections is covered by Rhapsody documentation, but some highlights are presented here.

Building Framework Libraries for INTEGRITY

To build the framework libraries for a particular INTEGRITY target, invoke **IntegrityBuild.bat** from the command line in the appropriate Rhapsody language directory and with pointers to the Compiler directory, the INTEGRITY directory, and the target BSP. For example:

```
> cd Rhapsody\Share\LangC
> IntegrityBuild C:\GHS\intxxxx sim800 C:\ghs\comp_xxxxxx -trg ppc_integrity.tgt
> cd Rhapsody\Share\LangCpp
> IntegrityBuild C:\GHS\intxxxx sim800 C:\ghs\comp_xxxxxx -trg ppc_integrity.tgt
```

The resulting Adapter libraries are linked into your models when you build them.

Configuring a Model for INTEGRITY

A few options need to be configured for a model before you can build the model for an INTEGRITY target. Double-click the configuration you want to build for INTEGRITY (configurations are located at **Project Name** → **Components** → **Component Name** → **Configurations** → **Configuration Name**), and set the appropriate options—or ensure that they are set—as follows.

Under the **General** tab:

- The **Executable** radio button should be selected if you want to generate an INTEGRITY Dynamic Download with a single AddressSpace that contains a Rhapsody component.
- The **Library** radio button should be selected if you want to generate a basic library that can then be linked to some other executable.

Under the **Settings** tab:

- **Instrumentation Mode** should be set to **Animation** to enable model-level debugging.
- **Time Model** may optionally be set to **Simulated**. When you are building a model for ISIM, it is often useful to set this option. For example, because a one-second timeout is calculated by a simulated processor, the actual wall-clock time could be noticeably longer.
- **Environment** should be set to **INTEGRITY5**. (Note that this is the case even if you are using INTEGRITY 10.)
- **Build Set** should be set to **Debug** to enable source-level debugging.

Under the **Properties** tab:

- **C_CG** (or **CPP_CG**, etc.) → **INTEGRITY5** → **BLDtarget** should be set to your desired BSP (sim800, for example).
- **C_CG** (or **CPP_CG**, etc.) → **INTEGRITY** → **IDEInterfaceDLL** should be set to MULTI's DLL (C:\ghs\multi_xxx\rhapsody_multi_ide.dll, for example) to enable synchronous debugging. This must include a full path.
- **C_CG** (or **CPP_CG**, etc.) → **INTEGRITY5** → **PrimaryTarget** should be set to an INTEGRITY .tgt file (ppc_integrity.tgt, for example).

Animated, Source, and Synchronous Debugging

Model-level debugging (known as *Animation*) is covered by the Rhapsody documentation. By default, it depends upon TCP/IP, and thus requires a TCP/IP stack on an INTEGRITY target.

Source-level debugging is covered in this book and is a standard capability of the MULTI environment.

Model-level and source-level debugging can be done independently, and even at the same time, with only the following constraints on how the image is loaded onto the target. These constraints apply to synchronized model-level and source-level debugging only:

- Components must be loaded onto an INTEGRITY target via the **Code** → **Target** menu in Rhapsody (for Rhapsody 6.x, see **Code** → **IDE**). This allows Rhapsody to load the image via MULTI and to maintain synchronization of the two debug environments.
- If you want to reload an image while a model-level session is running, you must select **Code** → **Stop Execution** before reloading or unloading the image; otherwise the model-level session cannot be run after the reload.

Example: Using MULTI 6, INTEGRITY 10, and Rhapsody 7

A number of example models are provided with Rhapsody. The following numbered list illustrates how to configure and build the pingpong example model with MULTI for an INTEGRITY ISIM ARM target. For more details about how to configure and use Rhapsody in general, please refer to Rhapsody's user manual.

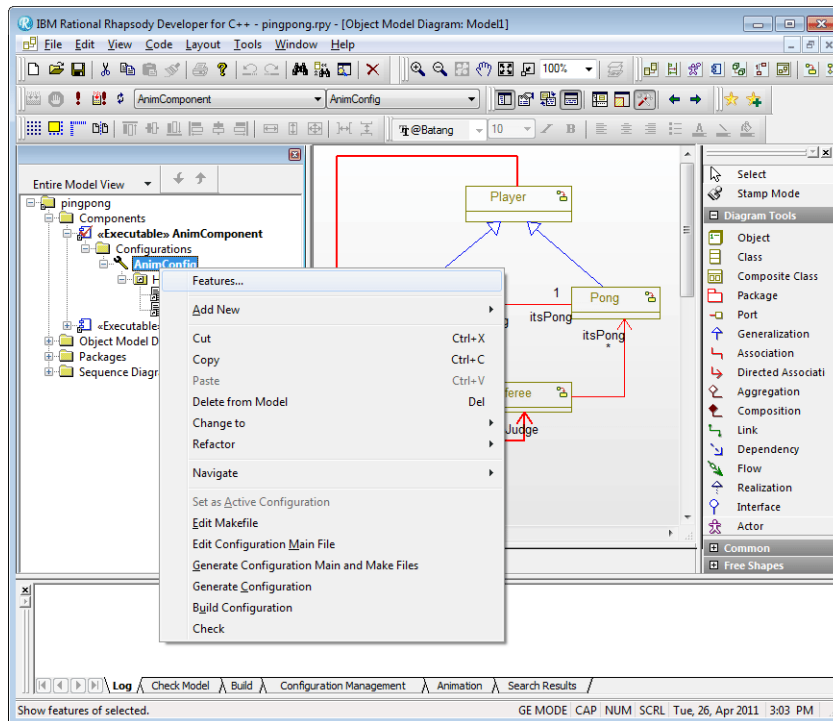
1. Build the OXF framework libraries:

From the command line, build the OXF framework libraries for an ISIM ARM target. Using your particular Compiler and INTEGRITY directories, enter:

```
> cd Rhapsody\Share\LangCpp
> IntegrityBuild C:\GHS\intxxxx simarm C:\ghs\comp_xxxxxx -trg arm_integrity.tgt
```

2. Open the example:

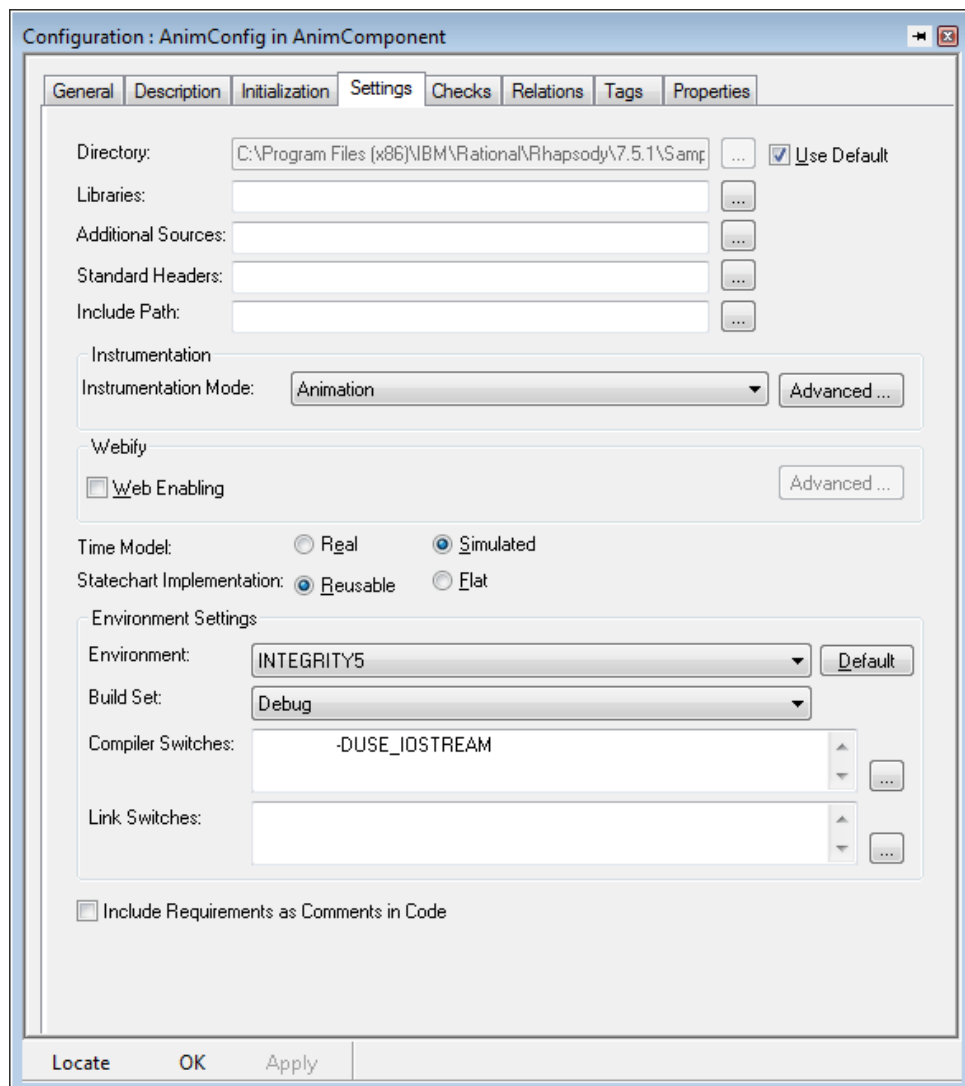
Open the **pingpong.rpy** model in the **Rhapsody\Samples\CppSamples\PingPong** directory. Right-click the **AnimConfig** configuration and select **Features**.



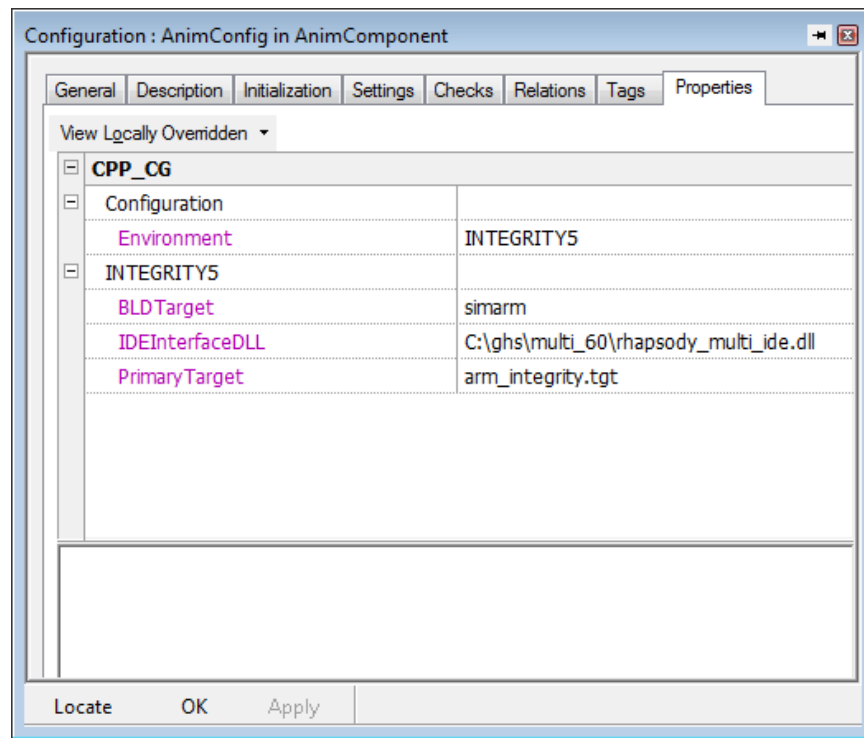
3. Configure for INTEGRITY:

a. Under the **Settings** tab:

- i. Ensure **Instrumentation Mode** is set to **Animation**.
- ii. To make timeouts run faster in ISIM, you may select **Simulated** as the **Time Model**.
- iii. Set the **Environment** to **INTEGRITY5**. (Note that this is the case even if you are using INTEGRITY 10.)
- iv. Ensure **Build Set** is set to **Debug**.



- b. Under the **Properties** tab:
 - i. Select **View All**.
 - ii. Set **CPP_CG** → **INTEGRITY5** → **BLDTarget** to **simarm**.
 - iii. Set **CPP_CG** → **INTEGRITY5** → **IDEInterfaceDLL** to **C:\ghs\multi_xxx\rhapsody_multi_ide.dll** (specify a full path).
 - iv. Set **CPP_CG** → **INTEGRITY5** → **PrimaryTarget** to **arm_integrity.tgt**.



4. Build the model:
 - a. To generate the source code and MULTI build files and to build it, select **Code** → **Generate/Make**.
 - b. By default, this creates an INTEGRITY Dynamic Download with a single AddressSpace that encapsulates the pingpong model.
5. Run the model:
 - a. The Dynamic Download can be loaded onto an INTEGRITY target (ISIM) via MULTI using a standard **rtserv2** connection. This provides source-level debugging. If the model is run, it communicates with

Rhapsody via TCP/IP. Rhapsody then provides model-level debugging. Each debugging method is independent and can coexist with the other.

- b. Or, to synchronize the debugging activity, select **Code** → **Target** → **Connect** (then **Load**, then **Run**). This allows Rhapsody and MULTI to coordinate activities. Select the first task listed as **no-name** in the target list. This task represents the running model. When attached to it, clicking the **Go/Halt** buttons in either environment runs/halts the program at the source level and at the model level. When a breakpoint in the model is hit, MULTI navigates to the corresponding source code in the Debugger window.

Appendix E

Creating Custom Data Visualizations

Contents

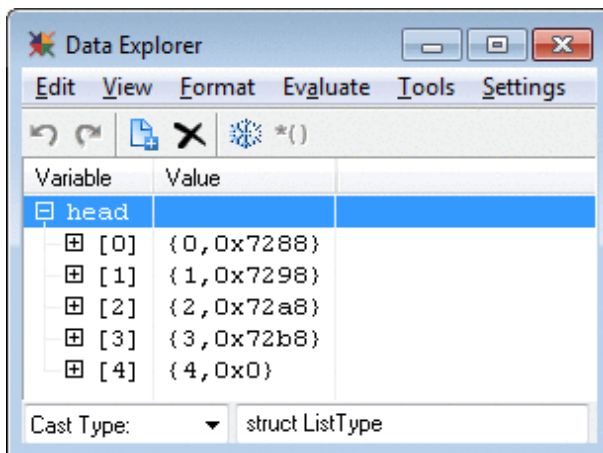
Using MULTI Data Visualization (.mdv) Files	743
Invoking Customized Data Visualizations	746
MULTI Data Visualization (.mdv) File Format	747

MULTI data visualization (**.mdv**) files can be used to create customized views which display certain types of variables according to your specifications. Depending upon the types of data you want to display, and your specific settings, these custom views will be shown in the Data Explorer or **Graph View** windows. The following two examples use custom visualizations to define the way data types are displayed.

A linked list using the following structure definition in your source code:

```
struct ListType {  
    int value;  
    struct ListType* next;  
}
```

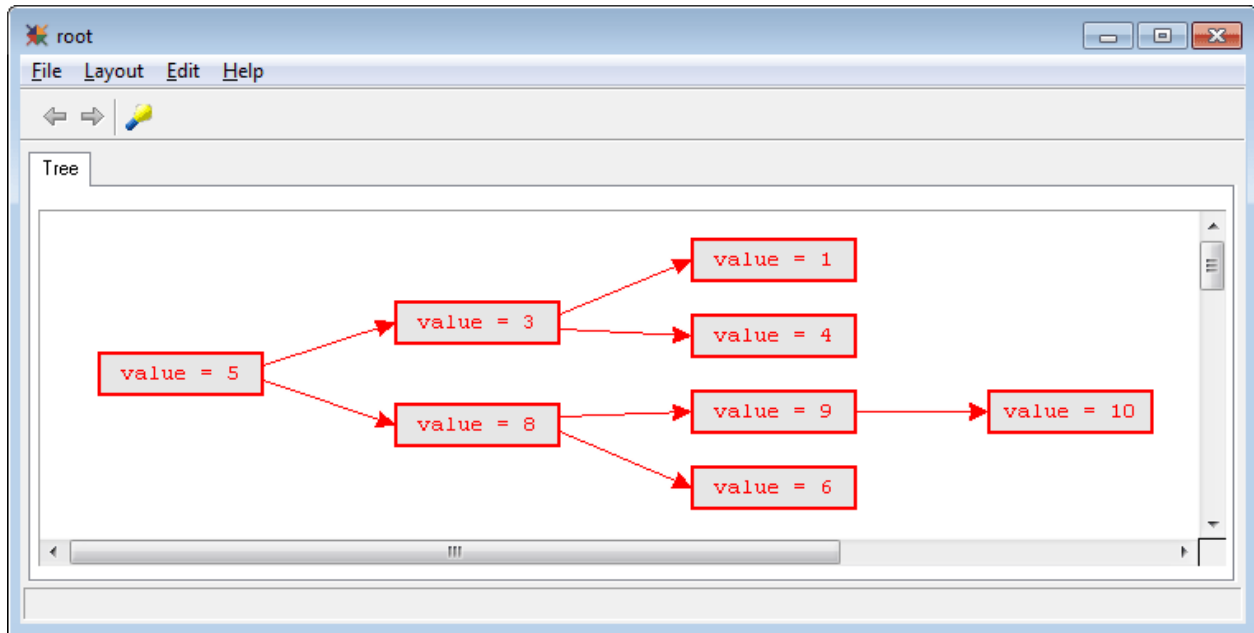
could be shown in a Data Explorer as:



A binary tree using the following structure definition in your source code:

```
struct TreeNode {  
    int value;  
    struct TreeNode *left, *right;  
}
```

could be shown in a **Graph View** window as:



Using MULTI Data Visualization (.mdv) Files

You can define a custom data visualization for any data type. (MULTI has built-in support for certain types of data structures. For information about using MULTI's built-in visualizations, see “Displaying Linked Lists” on page 191.) To use customized data visualizations, you must first create one or more MULTI data visualization (**.mdv**) files and then load them during your debugging session. These files can contain three kinds of descriptions that are used to create custom data visualizations:

- *Data descriptions* are the basic building blocks of custom data visualizations. They describe how MULTI should display variables of a particular type.

- *Profile descriptions* contain a collection of data descriptions, as well as some basic information about hierarchical relationships and the layout of the visualization(s) for the variables specified in the data descriptions.

You can create multiple profiles, but only one profile can be *active* at a time. Custom visualizations are only available for the variables described in the active profile or in no profile. (Data descriptions that are not included in a profile are considered to be part of a global profile that is always accessible.) You can change profiles easily by using the **dvprofile** *prof_name* command. For information about the **dvprofile** command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

- *View descriptions* specify a configuration of profiles to use for displaying variables in a **Graph View** window. These allow you to specify a profile to be active during evaluation of the view. They also allow you to create a “dual pane” display, where selecting an object in the first pane causes a new graph rooted on that object to be displayed in the second pane, using a different profile and therefore potentially providing a different view of the data.

Each of these potential components of **.mdv** files is described in more detail in “MULTI Data Visualization (.mdv) File Format” on page 747, but the following outline of the basic syntax will provide a context for understanding the descriptions. (Lines enclosed in square brackets in the following template are optional; the square brackets are not part of the syntax. The curly brackets, however, are part of the syntax.)

```
unique_prof_id {
  profile_name = prof_name
  [parent_profile = parent_prof_name]
  [default_root = def_root_prof]
  [vertical_layout = bool_expr]
  [add_siblings_of_root = bool_expr]

  data_descriptions {
    data_desc_name_1 {
      signature = {"sig1", "sig2", ... }
      [required_fields = {"field1", "field2", ... }]
      type = "data_desc_type"
      [predicate = "pred"]
      [required and optional fields depending on type]...
    }
    ...
  }
}
```

```
unique_view_id {
  initial_pane = "pane_name"
  default_root = "def_root_view"
  [tab_name = "tab"]

  panes {
    pane_name1 {
      profile_name = "prof_name1"
      [child_pane = "child_pane_name1"]
    }
    ...
  }
}
```

For an example **.mdv** file, see “.mdv File Examples” on page 770.

Loading and Clearing MULTI Data Visualization (.mdv) Files

To make the data descriptions, profiles, and views that you have defined available in the Debugger, you must load your **.mdv** file(s) by entering the following command in the Debugger command pane:

dvload *filename*

where *filename* is the name of a custom MULTI data visualization (**.mdv**) file. You can load multiple **.mdv** files by issuing this command repeatedly, once for each file you want to load. If the file is specified with a relative path, MULTI searches for it in the user configuration directory, then in the Compiler installation's configuration directory, and finally in the IDE installation's configuration directory.

All of the data descriptions, profile descriptions, and view descriptions included in all loaded **.mdv** files will be available during your MULTI session, but will not persist across re-launches. However, only those data descriptions contained in the active profile or in no profile will be accessible for viewing, and only one profile can be active at a time. These are the data descriptions that MULTI will match your program data against when selecting data to be displayed in your custom view. (You can change the active profile to any profile in a loaded **.mdv** file by issuing the **dvprofile** *prof_name* command. See “Profile Descriptions” on page 766.)

To clear MULTI data visualization files you have loaded, use the following command:

dvclear

This command clears all loaded data descriptions.

Invoking Customized Data Visualizations

Most types of data descriptions are used to describe how a type should be displayed in Data Explorers and by the **print** command (see Chapter 8, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book). These data descriptions are used automatically when a variable matching the data description is viewed in a Data Explorer, or printed in the Debugger command pane.

The `graph` data description type, on the other hand, describes how to display a graphical representation of the variable in a **Graph View** window (see “The Graph View Window” on page 247). Data descriptions with the `use_for_print_and_view` field set to `false` are also not used when viewing a variable in a Data Explorer or printing to the Debugger command pane. They are typically used to define containers that are used in `graph` data description types. For example, the children of a node can be specified in a container, which would need a corresponding data description.

For information about how MULTI matches a data description to a variable, see “Data Descriptions” on page 747.

To view the graphical representation of data for which you have `graph` data descriptions, use one of the following methods:

- Right-click a variable and select **Graph Data** from the shortcut menu that appears. If you have right-clicked a variable matching a `graph` data description, MULTI will create a graph of objects based on the data description, and display it in a **Graph View** window. The graph will be rooted on the clicked variable. If no data description is found for the variable's type, the text of the node will be the output of **print variable**.
- Right-click in the Debugger source pane and select **Display Program Visualizations** or enter the command **dataview** in the Debugger command pane (see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book). This will display a graph with one tab for each defined custom *view description*, using the default root and profile specified for each view.

- Enter the following command in the Debugger command pane:

`dataview` *prof_name* | *view_name*

This will open a graph displaying the specified profile (*prof_name*) or view (*view_name*). The graph will use the default root (*def_root_prof* or *def_root_view*) specified in the profile or view description.

MULTI Data Visualization (.mdv) File Format

A MULTI data visualization (.mdv) file can contain data descriptions, profile descriptions, and/or view descriptions. MULTI uses the information in these descriptions to filter which data to display and to format custom visualizations of the filtered data according to your specifications. The sections below give the exact syntax for each of these types of descriptions.



Note

Many of the fields used to describe data types, profiles, and views in .mdv files take expressions. See “Using Expressions in MULTI Data Visualization (.mdv) Files” on page 770 for information about valid expressions.

Data Descriptions

Data descriptions describe how data is accessed for a particular type. A data description has the following format (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {  
  signature = {"sig1", "sig2", ... }  
  [required_fields = {"field1", "field2", ... }]  
  type = "data_desc_type"  
  [predicate = "pred"]  
  [required and optional fields depending on type]  
}
```

The meaning of the elements in this format are described in the following table.

<code>data_desc_name</code>
Uniquely identifies the data description. Data descriptions with the same name will overwrite each other.
<pre>signature = {"sig1", "sig2", ... }</pre> Specifies the signature or signatures (<i>sig1</i>, <i>sig2</i>, etc.) of the data type being described in the data description. The signature is used to match against the type of a variable, to determine whether the data description should be used for the variable. Wildcards can be used when matching. For example, to match an STL list, use the format: <pre>signature = {"std::list<*>"}</pre> To match objects or pointers of either type <code>DeviceType1</code> or <code>DeviceType2</code>, use the format: <pre>signature = {"DeviceType1", "DeviceType2"}</pre> Variables are always cast to derived types and dereferenced before being used in a data description. This field is required.
<pre>required_fields = {"field1", "field2", ... }</pre> Specifies a list of fields (<i>field1</i>, <i>field2</i>, etc.) that must be present in the type to match this data description. This line is not required, but specifying required fields can be useful for sanity checking.

```
type = "data_desc_type"
```

Specifies what type of data description is being defined, where `data_desc_type` is one of the following:

- `container`
- `list`
- `null_terminated_list`
- `circular_list`
- `binary_tree`
- `structure`
- `alias`
- `singleton`
- `function_definer`
- `graph`

See “Data Description Types and Their Type-Specific Fields” on page 750 for more information about each data type.

This field is required.

```
predicate = "pred"
```

Specifies a predicate expression to determine whether the data description should be used. For example, if this data description should only be used when the `value` field of the struct is 0, the predicate line of the description would be:

```
predicate = "return self.value == 0"
```

Specifying a predicate is optional.

required and optional fields depending on type

Specifies required and optional fields according to the type of data description. See the following sections for a description of the required and optional fields for each data type.

When checking whether a data description matches the variable, MULTI first casts the variable to its derived type. Then MULTI compares the type name to the `signature` entries in the data description. If the type name matches, MULTI checks that all fields listed in `required_fields` are present in the structure, and then evaluates the predicate expression, if any. If the type matches the `signature`, `required_fields`, and the predicate expression returns `true`, the data description is used to display that type.

Data Description Types and Their Type-Specific Fields

The available data types are described in the sections below, along with the required and optional data description fields for each.

The container Data Type

The `container` type is used to describe a type which is a container of elements, such as a list, vector, or map. A data description for a container includes information about how to create an *iterator* for the container, how to walk the iterator through the container, and how to retrieve a value from the iterator at each step. An iterator is a value that represents a position in the container. For example, for a linked list, the iterator might be the address of a node.



Tip

If extra information about the parent structure is needed by an iterator, set a `MULTI` special variable to contain this information in the `begin_iter` expression (see below).

An example of a `container` data description for an STL list is shown below.

```
list {
  signature = {"list<*>", "std::list<*>"}
  required_fields = {"_Mysize", "_Myhead"}
  type = "container"

  size = "return self._Mysize;"
  begin_iter = "return self._Myhead._Next;"
  next_iter = "return self._Next;"
  value_from_iter = "return self._Myval;"
}
```

The type-specific fields for the `container` data description type are described in the following table.

<pre>size = "size_expr"</pre>
Specifies an expression, <i>size_expr</i> , that returns an integer value that equals the number of elements in the container.
This field is required.

```
begin_iter = "begin_iter_expr"
```

Specifies an expression, *begin_iter_expr*, that returns some representation of a beginning iterator over the container. For example, a list data description might return a pointer to the first list node. A vector might return an index to the first element.

This field is required.

```
next_iter = "next_iter_expr"
```

Specifies an expression, *next_iter_expr*, that is used to retrieve the next iterator from the current iterator. In this expression, the `self` variable refers to the current iterator, not to the original structure.

For example, for a list, this line might be:

```
next_iter = "return self.next"
```

This field is required.

```
value_from_iter = "value_from_iter_expr"
```

Specifies an expression, *value_from_iter_expr*, that is used to retrieve the element value from the current iterator. In this expression, the `self` variable refers to the current iterator, not the original structure.

For example, for a list, this line might be:

```
value_from_iter = "return self.value"
```

This field is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in *graph* data descriptions), but do not want the container to display in a Data Explorer.

The list Data Type

The `list` type is a specialized type of container used to describe C-style lists. The list is made up of structures that are both list nodes and elements, each connected to its successor by a *next* pointer. This type of data description is useful for lists that rely on a termination condition, rather than a size parameter.

For example, consider code using the following structure definition:

```
struct ListType {
    int value;
    struct ListType *next;
};
struct ListType EndOfListNode;
```

To view a `ListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
ListTypeDataDescription {
    signature = {"ListType"}
    required_fields = {"value", "next"}
    type = "list"

    next = "next"
    termination_condition = "return &self == &EndOfListNode"
}
```

The type-specific fields for data descriptions for lists are described in the table below.

<pre>next = "next_field_name"</pre> Specifies <i>next_field_name</i> as the field within the type that points to the next element in the list. This line is required.
<pre>termination_condition = "term_condition_expr"</pre> Specifies the expression <i>term_condition_expr</i> as the condition that ends the list. This field is required.
<pre>use_for_print_and_view = bool_expr</pre> Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where <i>bool_expr</i> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in <code>graph</code> data descriptions), but do not want the container to display in a Data Explorer.

The `null_terminated_list` Data Type

The `null_terminated_list` type is a specialized type of list used to describe standard C-style null-terminated lists.

For example, consider code using the following structure definition:

```
struct NullTerminatedListType {  
    int value;  
    struct NullTerminatedListType *next;  
};
```

To view a `NullTerminatedListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
NullTerminatedListTypeDataDescription {  
    signature = {"NullTerminatedListType"}  
    required_fields = {"value", "next"}  
    type = "null_terminated_list"  
  
    next = "next"  
}
```

The type-specific fields for data descriptions for null-terminated lists are described in the table below.

<pre>next = "next_field_name"</pre> Specifies <i>next_field_name</i> as the field within the type that points to the next element in the list. This line is required.
<pre>use_for_print_and_view = bool_expr</pre> Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where <i>bool_expr</i> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in <code>graph</code> data descriptions), but do not want the container to display in a Data Explorer.

The `circular_list` Data Type

The `circular_list` type is a specialized type of list used to describe C-style circular lists (i.e., lists in which the end is discovered by returning to an element that has already been seen).

For example, consider code using the following structure definition:

```
struct CircularListType {
    int value;
    struct ListType *next;
};
```

To view a `CircularListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
CircularListTypeDataDescription {
    signature = {"CircularListType"}
    required_fields = {"value", "next"}
    type = "circular_list"

    next = "next"
}
```

The type-specific fields for data descriptions for circular lists are described in the table below.

<pre>next = "next_field_name"</pre> Specifies <i>next_field_name</i> as the field within the type that points to the next element in the list. This line is required.
<pre>use_for_print_and_view = bool_expr</pre> Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where <i>bool_expr</i> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in <code>graph</code> data descriptions), but do not want the container to display in a Data Explorer.

The `binary_tree` Data Type

The `binary_tree` type is used to describe a type of container used to describe a binary tree structure. The tree is traversed in order — first the `left` subtree is traversed, then the value of the node is displayed, then finally the `right` subtree is traversed.

For example, consider code using the following structure definition:

```
struct TreeNode {
    int value;
    struct TreeNode *left, *right;
}
```

To view a `TreeNode` variable as an in-order list of elements rather than simply as a structure, you could use the following data description:

```
TreeNodeDataDescription {
    signature = {"TreeNode"}
    required_fields = {"value", "left", "right"}
    type = "binary_tree"

    left = "left"
    right = "right"
}
```

The type-specific fields for data descriptions for binary trees are described in the following table.

<pre>left = "left_field_name"</pre> Specifies <i>left_field_name</i> as the field representing the left child pointer. This field is required.
<pre>right = "right_field_name"</pre> Specifies <i>right_field_name</i> as the field representing the right child pointer. This field is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in `graph` data descriptions), but do not want the container to display in a Data Explorer.

The structure Data Type

The `structure` type is used to describe a list of artificial fields that can be viewed for the data type. The data type is seen as a structure containing the fields defined in the `structure` data description.

The syntax for the `structure` type data description follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {  
    signature = {"sig1", "sig2", ... }  
    [required_fields = {"field1", "field2", ... }]  
    type = "structure"  
    [predicate = "pred"]  
  
    fields {  
        field1{  
            name = name_string  
            value = "value_expr"  
            mutable = bool_expr  
        }  
        ...  
    }  
}
```

An example of a `structure` data description for an STL list iterator is shown below.

```
list_iterator {  
    signature = {"list<*>::*iterator"}  
    required_fields = {"_Ptr"}  
}
```

```

type = "structure"
predicate = "return self._Ptr != 0"

fields {
  next {
    name = "_Next"
    value = "return self._Ptr->_Next"
  }
  prev {
    name = "_Prev"
    value = "return self._Ptr->_Prev"
  }
  value {
    name = "_Myval"
    value = "return self._Ptr->_Myval"
  }
}
}

```

The type-specific fields for data descriptions for structures are described in the table below.

field1
A unique identifier for the field description.
<code>name = "name_string"</code> Specifies <i>name_string</i> as the name of the artificial field you are creating. This name must not contain any whitespace. This field is required.
<code>value = "value_expr"</code> Specifies the expression, <i>value_expr</i>, as the value to be displayed for the field. This field is required.
<code>mutable = bool_expr</code> Specifies whether the value can be edited in Data Explorer windows, where <i>bool_expr</i> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which allows the values to be edited.

The alias Data Type

The `alias` type is used to specify that one type should be viewed as if it were a different type.

An example of an `alias` data description for an STL string is shown next. This data description causes an STL string to be displayed as its underlying C-style string.

```
string {
    signature = {"basic_string<*>"}
    required_fields = {"_Bx", "_Myres"}
    type = "alias"

    value = "if (_BUF_SIZE <= self._Myres) {"
    value += "    return self._Bx._Ptr;"
    value += "} else {"
    value += "    return self._Bx._Buf;"
    value += "}"
    mutable = false;
}
```

The type-specific fields for data descriptions for aliases are described in the table below.

<pre>value = "val_expr"</pre> Specifies that the result of the expression <code>val_expr</code> will be displayed in place of the actual value of the item matching this data description. This field is required.
<pre>mutable = bool_expr</pre> Specifies whether the value can be edited in Data Explorer windows, where <code>bool_expr</code> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which allows the value to be edited.
<pre>replace_self = bool_expr</pre> Specifies whether the variable being viewed is replaced with the result of the expression specified by the line <code>value = "val_expr"</code>. The argument <code>bool_expr</code> is either <code>true</code> or <code>false</code>. If <code>bool_expr</code> is <code>true</code>, the variable being viewed is replaced with the result of <code>val_expr</code>. If <code>bool_expr</code> is <code>false</code>, the result of <code>val_expr</code> is used as the value of the variable, but the original variable's type is still displayed. Defining this field is optional. It defaults to <code>false</code>.

The singleton Data Type

The `singleton` type is used to specify a data type that should be viewed as a singleton type. Since the data type is a singleton, only one instance of it is ever created. Specifying a data description for the singleton allows you to view the instance of the data type by viewing the type.

For example, a `singleton` data description for the following type:

```
class SingletonClass {
private:
    SingletonClass();
    static SingletonClass* instance;
public:
    static SingletonClass* getInstance() {
        if (!instance)
            instance = new SingletonClass;
        return instance;
    }
};
```

would be:

```
singleton_class {
    signature = {"SingletonClass"}
    type = "singleton"

    instance = "SingletonClass::instance"
}
```

This data description would allow you to see the singleton instance of the class by viewing the class type. There is only one type-specific field for data descriptions for singletons, which is described in the table below.

```
instance = "variable_name"
```

Specifies *variable_name* as the instance variable of the type. This must be a valid variable identifier.

This field is required.

The `function_definer` Data Type

The `function_definer` type is used to define macros that can be called from the Debugger command line as if they were member functions of the data type.

The syntax for the `function_definer` data description follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {
    signature = {"sig1", "sig2", ... }
    [required_fields = {"field1", "field2", ... }]
    type = "function_definer"
    [predicate = "pred"]

    functions {
        unique_name {
            name = "name_string"
            [arguments = {"arg_string1", "arg_string2", ...}]
            body = "body_expr"
        }
    }
}
```



Note

The functions block shown above can also be added to any of the other data description types (except `graph`), rather than being placed in a `function_definer` type.

A sample `function_definer` data description for an STL list is shown next.

```
list_funcs {
    signature = {"list<*>"}
    required_fields = {"_Mysize", "_Myhead"}
    type = "function_definer"

    functions {
        size {
            name = "size"
            body = "return self._Mysize"
        }
    }
}
```

```
    front {  
        name = "front"  
        body = "return self._Myhead->_Next->_Myval"  
    }  
    back {  
        name = "back"  
        body = "return self._Myhead->_Prev->_Myval"  
    }  
}
```

The type-specific fields for data descriptions for function definers are described in the table below.

<code>unique_name</code>
A unique name for the function entry.
<code>name = "name_string"</code> Specifies the string <code>name_string</code> as the name of the function. For example, if the name is <code>size</code> , then the function can be called by <code>obj.size()</code> . This field is required.
<code>arguments = {"arg_string1", "arg_string2", ...}</code> Specifies additional arguments (<code>arg_string1</code> , <code>arg_string2</code> , ...) to the function. Like C++ class member functions, there is always one implicit argument to the function— <code>self</code> —which contains a pointer to the object being referenced (taking the place of the <code>this</code> parameter in C++). This line is optional.
<code>body = "body_expr"</code> Specifies an expression, <code>body_expr</code> , that is evaluated when the function is called. This field is required.

The graph Data Type

The `graph` type is used to describe a data type that should be displayed graphically in a **Graph View** window. The data description defines how a data type will be represented in nodes in a graph, and contains information about the text that should appear in the nodes, the children of the nodes, and the parents of the nodes.

Consider the following type:

```
class Device {
public:
    const char* deviceName;
    int deviceId;

    Device(const char *name, const int id);
    void addInputBus(Bus* input);
    void addOutputBus(Bus* output);

protected:
    std::list<Bus*> inputs;
    std::list<Bus*> outputs;
};
```

This type might have the following data description:

```
device {
    signature = {"Device"}
    type = "graph"
    node_text = "mprintf(\"%s\nDevice ID: %d\", self.deviceName, self.deviceId)"
    children {
        outputs {
            value = "return self.outputs"
            is_container = true
        }
    }
    parents {
        inputs {
            value = "return self.inputs"
            is_container = true
        }
    }
}
```

See “.mdv File Examples” on page 770 for a complete example of the `graph` data description.

The type-specific fields for data descriptions for graphs are described in the following table.

```
replace_with = "repl_node_expr"
```

Specifies a value, *repl_node_expr*, to replace (in **Graph View**) the node being defined. If this line is present, all other fields in the data description are ignored. The node being defined will not show up in the graph, but will instead be replaced by a node representing the value returned by the expression *repl_node_expr*.

To make a node not show up at all, use the line:

```
replace_with = "return (void*) 0"
```

This field is optional.

```
node_text = "commands"
```

Specifies one or more MULTI commands, separated by semicolons, that generate output which will appear on the node being defined. Commands that can be used for output include **echo**, **print**, and **mprintf**. Any textual output from these commands will be captured and displayed in the node.

This field is required.

```
vertical_layout = bool_expr
```

Specifies whether the graph should use a vertically-oriented layout or a horizontally-oriented layout, where *bool_expr* is true or false. If *bool_expr* is true, the graph will use a vertically-oriented layout.

This field is optional and defaults to false. This field can be set on a profile-by-profile basis (see “Profile Descriptions” on page 766).

```
children {
  child_name {
    value = "child_val_expr"
  }
  ...
}
```

Specifies information about the node's children. Each child must have a unique name, *child_name*, but the number of children is unlimited.

For more information about the syntax and fields for describing children, see the next section.

Specifying children is optional.

```
parents {  
  parent_name {  
    value = "parent_val_expr"  
  }  
  ...  
}
```

Specifies information about the node's parents. Each parent must have a unique name, *parent_name*.

If an edge is specified by both the parent and the child, it will only appear once. This provides an easy mechanism for describing complex graphs without having to worry about duplication.

For more information about the syntax and fields for describing parents, see the next section.

Specifying parents is optional.

```
next_sibling = "sibl_expr"
```

Specifies information about the node's siblings, where *sibl_expr* is an expression.

Siblings are laid out orthogonally to the primary layout orientation. If the graph is horizontally oriented, siblings will be connected vertically. This provides a convenient way to display data in a list of trees. There can be at most one sibling edge entering a node, and one sibling edge leaving a node.

Specifying siblings is optional.

```
max_display_size = int
```

Specifies the maximum number, *int*, of siblings displayed in a chain.

This field is optional and defaults to 20, meaning that if you have a list being displayed as siblings, it will get cut off after the twentieth element.

Describing Parents and Children of the Node

The core functionality of the `graph` type is describing children and parents of a node. The `children` of a node may be significant pointers contained in the structure, entries from a lookup table, or any other value that can be calculated given the structure as a starting point.

The syntax for defining a child follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
children {  
  child_name {  
    value = "child_val_expr"  
    [is_container = bool_expr]
```

```
[is_array = bool_expr]  
[array_len = "length_expr"]  
[invisible_edge = bool_expr]  
}  
...  
}
```

The lines in this syntax are described in the following table.

<pre>value = "<i>child_val_expr</i>"</pre> Specifies the value of the child, where <i>child_val_expr</i> is an expression. For example, in a binary tree you might define one child with: <pre>value = "return self.left"</pre> and another with: <pre>value = "return self.right"</pre> You must specify a value for every child.
<pre>is_container = <i>bool_expr</i></pre> Specifies whether this child should be treated as a container of children, where <i>bool_expr</i> is either true or false. If <i>bool_expr</i> is true, the child is treated as a container — MULTI looks up a data description for that type and adds each element of the container as a child. STL container descriptions have been provided, so any time you have a structure with a list, vector, or map of children, you can simply set this value to true and each element will be added as a child. This field is optional and defaults to false. This field is mutually exclusive with <i>is_array</i>; only one of <i>is_array</i> and <i>is_container</i> can be true.
<pre>is_array = <i>bool_expr</i></pre> Specifies whether the child should be treated as an array of children, where <i>bool_expr</i> is either true or false. This field is optional and defaults to false. This field is mutually exclusive with <i>is_container</i>; only one of <i>is_array</i> and <i>is_container</i> can be true.

```
array_len = "length_expr"
```

Specifies an expression, *length_expr*, to be used to calculate the length of the array. An example child using arrays might be:

```
value = "return self.myChildArray"
is_array = true
array_len = "return self.myChildArrayLen"
```

This field is required if *is_array* = true.

```
invisible_edge = bool_expr
```

Specifies whether the edge defined by this parent or child relationship will be displayed, where *bool_expr* is either true or false.

This field gives you greater control over the layout of a graph. If *bool_expr* is true, the edge will not be displayed, but will still restrain the layout of the graph, allowing you to, for example, enforce an ordering on nodes without having visible edges connecting them.

This field is optional and defaults to false.

Parents can be described in the same way as children by using the `parents` keyword in place of the `children` keyword. A single data description can contain both `parents` and `children` blocks.

Profile Descriptions

A data visualization profile is a collection of data descriptions. You can create several profiles, but only one is *active* at a given time. Only the active profile is used when searching for data descriptions. Using profiles allows you to have multiple data descriptions for the same type, which you use in different circumstances. You can change which profile is active by using the command:

```
dvprofile prof_name
```

where *prof_name* is the name of the profile to be made active.

Any data description not in a profile is placed in a global profile, and is always accessible.

Profiles are defined in profile descriptions in **.mdv** files. A profile description has the following form. (Lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are part of the syntax.)

```
unique_prof_id {
  profile_name = "prof_name"
  [parent_profile = "parent_prof_name"]
  [default_root = "def_root_prof"]
  [vertical_layout = bool_expr]
  [add_siblings_of_root = bool_expr]

  data_descriptions {
    [series_of_data_descriptions]
  }
}
```

The lines in this syntax are described in the table below.

<code>unique_prof_id</code>	Specifies a unique identifier, <code>unique_prof_id</code>, for the profile. Reading in a new profile with the same identifier will overwrite the original. This field is required.
<code>profile_name = "prof_name"</code>	Specifies <code>prof_name</code> as the string used to refer to the profile when you are using the dvprofile command or working with different views. See the dvprofile command in “Data Visualization Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book, and see “View Descriptions” on page 768. This field is required.
<code>parent_profile = "parent_prof_name"</code>	Specifies that the profile with the name <code>parent_prof_name</code> is the parent of the profile. A profile can have at most one parent. When searching for a data description, MULTI starts at the active profile and then searches its parent, then the parent of that parent, and so on. If no parent is specified, the parent is set to the global profile by default. This field is optional.
<code>default_root = def_root_prof</code>	Specifies a variable identifier <code>def_root_prof</code> as the default root variable for graphs using this profile. This line is optional, but is used by the dataview command. For information about the dataview command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

```
vertical_layout = bool_expr
```

Specifies whether the graph should use a vertically oriented layout or a horizontally oriented layout, where *bool_expr* is either `true` or `false`. If *bool_expr* is `true`, the graph will use a vertically oriented layout.

This field is optional and defaults to `false`.

```
add_siblings_of_root = bool_expr
```

Specifies whether the siblings of the root node for the graph of this profile should be added, where *bool_expr* is either `true` or `false`. If *bool_expr* is `true`, siblings will be added.

This field is optional and defaults to `true`.

View Descriptions

Views are high-level visualizations in graph form. You can create views for elements of the `graph` data type by including *view descriptions* in your loaded **.mdv** file(s). When you select **Display Program Visualizations** from a right-click shortcut menu in the Debugger source pane, a graph will be displayed for each view that is defined in your loaded **.mdv** files. These individual graphs will appear on separate tabs of a single window.

Every view starts at a default root and contains one or two panes. A single pane view displays the data specified by a single profile. A dual-pane view displays a profile in the initial, or primary, pane. When a node in this primary pane is clicked, that node is used as the root variable for a graph that is then displayed in the second pane, using a different profile.

A view description contains specifications about the contents and layout of a single view. The syntax for defining views has the following form. (Lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are part of the syntax.)

```
view_unique_name {  
    initial_pane = "pane_name"  
    default_root = "def_root_view"  
    [tab_name = "tab"]  
}
```

```

panes {
  pane_name1 {
    profile_name = "prof_name"
    [child_pane = "child_pane_name"]
  }
  [pane_name2 {
    profile_name = "prof_name2"
  }]
}

```

The fields in this syntax are described in the table below.

<i>view_unique_name</i>
Specifies a unique identifier, <i>view_unique_name</i>, for the view. This name can be used with the dataview command to open a Graph View window on this view (see “Invoking Customized Data Visualizations” on page 746). This field is required.
<i>initial_pane = "pane_name"</i>
Specifies which pane is the primary pane for this view. In single pane views, this will be the name of the only pane. In dual pane views, this will be the name of the pane on the left. In dual pane views, clicking a node in the primary pane will cause the child pane to display a new graph with the selected object as its root. This field is required.
<i>default_root = "def_root_view"</i>
Specifies <i>def_root_view</i> as the root variable for the view. This field is required.
<i>tab_name = "tab"</i>
Specifies the string to be displayed on the tab that displays this view. This field is optional.
<i>pane_name</i>
Specifies the name (<i>pane_name1</i>, <i>pane_name2</i>) of the pane(s). This is used to refer to the pane within this view. This field is required for each pane in the view.

```
profile_name = "prof_name"
```

Specifies *prof_name* as the profile to use as the active profile when displaying this pane. (See “Profile Descriptions” on page 766.)

This field is required for each pane in the view.

```
child_pane = "child_pane_name"
```

Specifies *child_pane_name* as the child pane. When a node is selected in the pane being defined, the selected node will be displayed in this child pane.

This field is required if the pane has a child pane.

Using Expressions in MULTI Data Visualization (.mdv) Files

Many of the fields defined in **.mdv** files take an expression as their value. An expression is specified in quotation marks and can contain any statement or series of statements that can be evaluated from the MULTI command pane. The value of the expression is whatever value is found in the `return` statement. If no `return` statement is specified, the expression returns `void` and is probably incorrect.

Expressions can contain loops, MULTI special variables, and command line procedure calls. Expressions can also refer to program variables and return pointers, but they cannot return objects.

Expressions access the *current* structure through the `self` keyword. The current structure is usually the structure that the data description's signature matches. (There are some exceptions, such as when the current structure is the iterator while walking through a container.)

.mdv File Examples

Consider the following data types:

```
class Device {  
public:  
    Device(const char *name, const int id);  
    void addInputBus(Bus* input);  
    void addOutputBus(Bus* output);  
};
```

```
protected:
    const char* deviceName;
    int deviceId;

    std::list<Bus*> inputs;
    std::list<Bus*> outputs;
};

class Bus {
public:
    Bus(const char* name, const int id);
    void addInputDevice(Device* input);
    void addOutputDevice(Device* output);

protected:
    const char* busName;
    int busId;

    std::list<Device*> inputs;
    std::list<Device*> outputs;
};
```

A sample **.mdv** file used to display the preceding types is listed below.

```
demo_profile {
    profile_name = "Device_Bus_Profile"

    data_descriptions {
        device {
            signature = {"Device"}
            type = "graph"
            node_text = "mprintf(\"%s\nDevice ID: %d\", self.deviceName, self.deviceId)"
            children {
                outputs {
                    value = "return self.outputs"
                    is_container = true;
                }
            }
            parents {
                inputs {
                    value = "return self.inputs"
                    is_container = true;
                }
            }
        }
    }
}
```

```
bus {
    signature = {"Bus"}
    type = "graph"
    node_text = "mprintf(\"%s\nBus ID: %d\nInputs: %d\nOutputs: %d\", \"
    node_text+= \"self.busName, self.busId, self.inputs.size(), self.outputs.size())\"
    children {
        outputs {
            value = "return self.outputs"
            is_container = true;
        }
    }
    parents {
        inputs {
            value = "return self.inputs"
            is_container = true;
        }
    }
}

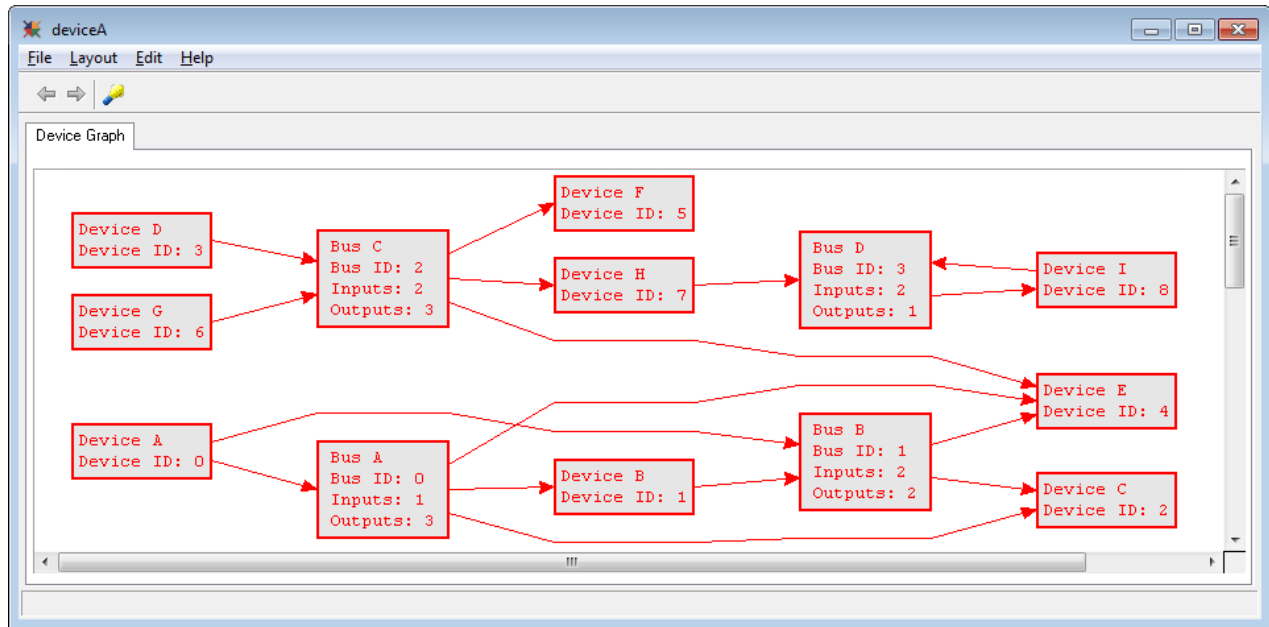
my_list {
    signature = {"std::list<*>"}
    type = "container"

    size = "return self._Mysize;"
    begin_iter = "return self._Myhead._Next;"
    next_iter = "return self._Next;"
    value_from_iter = "return self._Myval;"
}

}

demo_view {
    initial_pane = "only_pane"
    default_root = "deviceA"
    tab_name = "Device Graph"
    panes {
        only_pane {
            profile_name = "Device_Bus_Profile"
        }
    }
}
```

The preceding **.mdv** file might yield a **Graph View** like the following.



Appendix F

Register Definition and Configuration Reference

Contents

The GRD Register Definition Format	776
--	-----

This appendix describes the GRD register definition format.

The GRD Register Definition Format

A target's register information is defined in a GRD file, which will contain some or all of the following sections:

- The `general` section
- The `enum` section
- The `structure` section
- The `bitfield` section
- The `register` section
- The `group` section

Each section is enclosed in a block (delimited by `{}`) with a tag name identifying the section. The definition for each section can be separated into multiple parts. For example, you can define one `register` section for all special purpose registers, then a `group` section for the special purpose registers, and then another `register` section.

Preprocessor directives can be used in GRD files. The syntax and usage of the GRD file preprocessor is the same as in C++ except that the initial character is `%` instead of `#` (an initial `#` indicates a comment line in GRD format).

A preprocessor symbol can be substituted in a string with syntax `${DefinedSymbol}`. For example:

```
%define PPC400_DCR_BASE    5000

CPC0_SR {address="${PPC400_DCR_BASE}+0x000000b0"}
```

In the value of `CPC0_SR`'s `address` attribute, the Debugger substitutes `5000` for `${PPC400_DCR_BASE}` and then evaluates the resulting expression `5000+0x000000b0`.

Pre-defined preprocessor symbols for the debugging environment are available for use in GRD files. For more information, see the comment at the top of the file

os_constants_internal.grd, which is located at *compiler_install_dir/defaults/registers*.

Blocks can be represented in two formats:

- Delimited by curly braces ({ }). For example:

```
general {  
    register_offset = "16";  
}
```

- Delimited with a period (.). For example:

```
general.register_offset = "16";  
  
or  
  
"general"."register_offset" = "16";
```

The following sections describe each element's syntax.

The general Section

The `general` section defines the following settings for the target:

```
general {  
    version = number  
    pad_bitfield = bool  
    register_width = int  
    register_offset = string  
    byte_endian = string  
    bit_0_is_msb = bool  
    memory_space = number  
    pvr_address[index] = identifier  
    pvr_info[index] {register definition block}  
    pvr_mask[index] = mask  
    bcr_address[index] = identifier  
    bcr_info[index] {register definition block}  
    bcr_mask[index] = mask  
    multi_command = commands  
}
```



Note

The only required attribute is `version`. All other attributes are optional.

```
version = number
```

Defines the version number of the Green Hills Software Register Definition language used in the GRD file. The version number should be 3.

Example: `version = 3`

```
pad_bitfield = bool
```

Where `bool` can be:

- `true` or `false`
- An integer (where 0 means false, and a non-zero integer means true)
- A string expression starting with `%EVAL` that can be dynamically evaluated and result in a Boolean value

When a register with bitfields is defined, some bits or bit ranges may be omitted. This option tells the Debugger whether to automatically generate fields for the omitted bits or bit ranges when creating symbol information for the bitfield. If this attribute is true, when a register with the corresponding bitfield is displayed, the automatically generated padding fields will be displayed.

The default value for this attribute is `true`.

Example: `pad_bitfield = true`

```
register_width = int
```

Where `int` defines the default register width in bits. It can be an integer or a string expression starting with `%EVAL`. The expression enclosed in `%EVAL { }`, will be dynamically evaluated into an integer.

The default value for this attribute is 32.

The default register width will be applied to all registers except those registers whose definitions override the default width by using the `width` setting.

Example: `register_width = 16`

```
register_offset = string
```

Each register has an identifier in the Debugger, but it can have a different identifier (register offset) in the communication message between the Debugger and the debug server. This setting defines the default register offset that will be applied to all registers. A register definition can override the default offset expression by using the `offset` setting.

Example: `register_offset = "return 2*__regadr"`

`__regadr` is a reserved symbol whose value is the register's identifier in the Debugger.

```
byte_endian = string
```

This setting defines the default byte order for a register. A register definition can override the default byte order by using the `byte_endian` setting.

Here are the supported values:

- `default`— Always the same as the target (this is the default)
- `big` — Always big endian
- `little` — Always little endian

In big endian registers, the most significant byte is at the lowest address. In little endian registers, the least significant byte is at the lowest address.

Example: `byte_endian = "big"`

```
bit_0_is_msb = bool
```

This setting defines the default bit order for a register. A register definition can override the default bit order by using the `bit_0_is_msb` setting.

This setting indicates how a register's bit numbers are ordered. If the value for `bit_0_is_msb` is true, bit 0 will refer to the register's most significant bit (msb), and the largest bit number will refer to the register's least significant bit (lsb). Otherwise, bit 0 will refer to the register's least significant bit (lsb), and the largest bit number will refer to the register's most significant bit (msb).

The default value for the attribute is `false`.

Example: `bit_0_is_msb = true`

```
memory_space = number
```

This setting defines the default memory space for memory-mapped registers.

When defining a custom memory space, you should use a value larger than 65536. The Debugger reserves values less than 65536 for its own use.

The Debugger also supports the following memory spaces, which are defined in the **`os_constants_internal.grd`** file located at *`compiler_install_dir/defaults/registers`*.

- `MSPACE_TEXT_PHYSICAL` — The physical memory space for text.
- `MSPACE_DATA_PHYSICAL` — [default] The physical memory space for data.
- `MSPACE_TEXT_DEFAULT` — The default memory space (recognized by the debug server) for text.
- `MSPACE_DATA_DEFAULT` — The default memory space (recognized by the debug server) for data.

See also the `memory_space` attribute in “The register Section” on page 791.

Example: `memory_space = 95`

```
pvr_address[index] = identifier
```

If a processor has more than one processor version register (PVR), *index* can be appended to *pvr_address*. The two settings *pvr_address* and *pvr_address0* represent the same processor version register.

identifier can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Processor Version Register Information” on page 783.

Example: *pvr_address10* = 1061

```
pvr_info[index] {register definition block}
```

This setting defines a memory-mapped processor version register (PVR). To support memory-mapped PVRs, the syntax for the PVR definition is extended to support attributes from the register definition block. For information about the register definition block, see “The register Section” on page 791.

The *access* string must be a regular, special, io, or memory-mapped string, or it must be a string expression that can be resolved into such a string. For more information, see the *access* attribute in “The register Section” on page 791.

See also “Processor Version Register Information” on page 783.

When this PVR specification is used, the version number should be 2 or higher. See the *version* description earlier in this table.

Example:

```
pvr_info1 {
    access = "memorymapped";
    address = 0x123456;
    width = 16;
    read_length = 16;
    byte_endian="little";
}
```

```
pvr_mask[index] = mask
```

index has the same meaning as in the processor version register address.

mask can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Processor Version Register Information” on page 783.

Example: *pvr_mask10* = 0xffff0000

```
bcr_address[index] = identifier
```

If a target has more than one board configuration register (BCR), *index* can be appended to *bcr_address*. The two settings *bcr_address* and *bcr_address0* represent the same BCR. See also “Board Configuration Register Information” on page 783.

Example: *bcr_address10* = 258

```
bcr_info[index] {register definition block}
```

This setting defines a memory-mapped board configuration register (BCR). To support memory-mapped BCRs, the syntax for the BCR definition is extended to support attributes from the register definition block. For information about the register definition block, see “The register Section” on page 791.

The *access* string must be a regular, special, io, or memory-mapped string, or it must be an expression that can be resolved into such a string. For more information, see the *access* attribute in “The register Section” on page 791.

See also “Board Configuration Register Information” on page 783.

When this BCR specification is used, the version number should be 2 or higher. See the *version* description earlier in this table.

Example:

```
bcr_info1 {
    access = "memorymapped";
    address = 0x123456;
    width = 16;
    read_length = 16;
    byte_endian="little";
}
```

```
bcr_mask[index] = mask
```

index has the same meaning as in the board configuration register address.

mask can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Board Configuration Register Information” on page 783.

Example: *bcr_mask10* = 0xffff0000

```
multi_command = multi_commands
multi_command += multi_commands
```

This setting defines Debugger commands in a string that will be executed when the register definition is constructed inside the Debugger.

This setting is especially useful to support memory-mapped registers whose addresses are not based on another register's value but which could depend on other factors, such as the following:

- A debug server could move a register base around when it connects to the target (via its setup script, for example).
- A program such as the INTEGRITY kernel could move a register base when it runs.
- A program's build process might move a register base.

You can define the address of these registers with an expression using one or more Debugger local variables. For example, on a PowerPC target supporting QUICC, a Debugger variable is defined for the base of the QUICC registers:

```
$MULTI_PPC_QUICC_SIU_IMMR_BASE
```

which can be used to define the addresses of QUICC registers as described below:

```
bcr_address = "$MULTI_PPC_QUICC_SIU_IMMR_BASE + 0x10024"
```

The special variable can be initialized to the default base with a Debugger command:

```
multi_command +=
    " if (!$MULTI_PPC_QUICC_SIU_IMMR_BASE_IS_SET)
    {eval $MULTI_PPC_QUICC_SIU_IMMR_BASE = 0x0f000000;}"
```

where `eval` is used to prevent the Debugger from printing the new value of `$MULTI_PPC_QUICC_SIU_IMMR_BASE` after the assignment.

The default GRD files for some targets already use this mechanism. If the default IMMR base does not match your target setting, you can either change the GRD file directly or add statements such as the following into your program resource file (**.rc**) or target setup script (**.mbs**) to set the correct IMMR base and disable the default setting:

```
eval $MULTI_PPC_QUICC_SIU_IMMR_BASE_IS_SET = 1
eval $MULTI_PPC_QUICC_SIU_IMMR_BASE = 0xfc000000
```

Processor Version Register Information

Some targets have a processor version register or registers that can be used to identify target processors and their variants. If such information about the processor version is defined, the value can be used in GRD file preprocessor expressions with the following syntax:

```
pvr_value[index]
```

where *index* is the index (starting from 0) for the processor version register. A processor version register's basic information contains:

- An identifier, which is used by the Debugger to communicate with the underlying debug server
- A value mask

Board Configuration Register Information

As with processor version registers, a mechanism is provided for users to define board configuration register(s). If such information for the board configuration registers is defined, the value can be used in GRD file preprocessor expressions with the following syntax:

```
bcr_value[index]
```

where *index* is the index (starting from 0) for the board configuration register.

A board configuration register's basic information contains:

- An identifier, which is used by the Debugger to communicate with the underlying debug server
- A value mask

The enum Section

An enumeration definition must be in an enumeration section, and must be defined before it is referenced. An enumeration can have the following attributes:

```
enum {  
    enum_name {  
        description = string  
        help_key = string  
        auto_value_desc = bool  
  
        enum_entry_name {  
            description = string  
            help_key = string  
            long_name = string  
            value = int  
        }  
        ...  
    }  
}
```



Note

All attributes are optional.

```
description = string  
description += string  
desc = string  
desc += string
```

The description string can be an expression starting with %EVAL, which will be dynamically evaluated.

An enumeration can have a long description combined from multiple description settings (with +=). Newline characters (\n) must be specified if you want a newline displayed in the Register Information window's help pane.

Example:

```
desc = "Indicates whether interrupts are enabled:\n";  
desc += "    0 - disabled\n";
```

```
help_key = string
```

```
hk = string
```

The help key string can be an expression starting with %EVAL, which will be dynamically evaluated.

Example:

```
hk = "register.msr"
```

```
auto_value_desc = bool
```

The default setting for this option is true.

When this option is set to false, the Debugger will not automatically generate the specification for the enumeration's values in the help pane of the Register Information window for those bitfields whose type is the enumeration.

If you include a detailed list of enumeration values in the enumeration description, set this option to false to avoid duplicating the list of enumeration values in the help pane.

Example:

```
auto_value_desc = false
```

An enumeration must have at least one value entry, which can have the following attributes. If none of the attributes are specified, the braces should still be present.

```
description = string
```

```
description += string
```

```
desc = string
```

```
desc += string
```

The description string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
help_key = string
```

```
hk = string
```

The help key will be used to show online help. The help key string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
long_name = string
```

```
ln = string
```

The long name string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
value = int
value = {int1, ..., intn}
```

The enumeration entry will have the value or values specified. If an enumeration entry does not have a value explicitly specified, its value is assigned as follows:

- If no previous enumeration entry exists, the enumeration entry will be assigned the value 0.
- If a previous enumeration entry exists, the enumeration entry will be assigned a value one greater than that of that previous enumeration entry.
- If a previous enumeration entry contains a list of values, the enumeration entry will be assigned a value one greater than maximum value in the previous enumeration entry.

The structure Section

Any structure definition must be in a structure section, and it must be defined before it is referenced. Structure definitions are rarely used in register definition files. A structure can have the following components:

```
structure {
    structure_name {
        description = string
        help_key = string

        field_name {
            description = string
            help_key = string
            long_name = string
            type = string
        }
        ...
    }
}
```



Note

All attributes are optional.

```
description = string
```

```
desc = string
```

This is the structure's description. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
help_key = string
```

```
hk = string
```

This is the structure's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.

A structure's field can have the following attributes:

```
description = string
```

```
desc = string
```

This is the structure field's description. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
help_key = string
```

```
hk = string
```

This is the structure field's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
long_name = string
```

```
ln = string
```

This is the structure field's long name. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
type = string
```

The type string must be one of the following:

- A basic data type, including
 - `int`
 - `unsigned int`
 - `char`
 - `unsigned char`
 - `short`
 - `unsigned short`
 - `long`
 - `unsigned long`
 - `long long`
 - `unsigned long long`
 - `float`
 - `double`
 - `code`
- A pointer to one of the above basic data types (for example, `int *`)

The bitfield Section

Any bitfield definition must be in a bitfield section, and must be defined before it is referenced.

A bitfield can have the following attributes:

```
bitfield {  
    bitfield_name {  
        description = string  
        help_key = string  
  
        field_name {  
            description = string  
            help_key = string  
            short_name = string  
            long_name = string  
            auto_value_desc = bool
```

```

        loc = "begin_bit...end_bit"
        type = string
    }
    ...
}

```



Note

The only required attribute is `loc`. All other attributes are optional.

```
description = string
```

```
desc = string
```

The bitfield description is displayed in the Register Information window's help pane.

```
help_key = string
```

```
hk = string
```

This is the bitfield's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.

Every field in a bitfield must have a unique *field_name*.

```
description = string
```

```
description += string
```

```
desc = string
```

```
desc += string
```

A long description can be defined with multiple `+=` statements.

The field description is displayed in the Register Information window's help pane.

```
help_key = string
```

```
hk = string
```

This is the field's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
short_name = string
```

```
sn = string
```

This is the field's GUI name shown in **Register View** window and Register Information window. Multiple fields can have the same GUI name, like `Reserved` for those reserved fields. But each field's tag name must be unique in a bitfield.

If the attribute is absent, the field's tag name will be used as GUI name.

```
long_name = string
```

```
ln = string
```

This is the field's long name. It is displayed in the following places in the Register Information window:

- A tooltip in the concise display pane
- The **Description** in the detailed display pane
- The help pane

```
auto_value_desc = bool
```

This option is only relevant for fields with an enumeration type. The default setting for this option is `true`.

When this option is set to `false`, the Debugger will not automatically generate the specification for the field's values in the help pane of the Register Information window.

If you include a detailed list of enumeration values in the field description, set this option to `false` to avoid duplicating the list of enumeration values in the help pane.

```
loc = "begin_bit..end_bit"
```

```
loc = "bit_index"
```

The second form of this setting is for a field with only one bit.

```
type = type_string
```

The type must be one of the following basic types:

- hex
- binary
- unsigned
- signed
- A defined enumeration type

The register Section

A register definition must be in a register section. A register definition can have the following elements:

```
register {
    register_name {
        description = string
        help_key = string
        short_name = string
        alias = {string_list}
        long_name = string
        gui_tab = string
        cache_value = bool
        access = string
        address = number_or_string
        offset = string
        address_port = string
        address_mask = number_or_string
        data_port = string
        width = number_or_string
        byte_endian = string
        bit_0_is_msb = bool
        permission = string
        type = string
        read_length = number_or_string
        write_length = number_or_string
        hide = bool_or_string
        memory_space = number
    }
}
```



Note

The address attribute is always required. Other attributes may be required depending on the access type.

In the Debugger, each register has an identifier, which is usually defined by the address attribute. When the Debugger communicates with a debug server, a different register identifier may be used to refer to the register, in which case, the identifier is defined by the offset setting.

Some register attributes only apply to certain register types, as described in the following table.

<pre>description = string description += string desc = string desc += string</pre> This is the register's description. It is displayed in Register Information window.
<pre>help_key = string hk = string</pre> This is the register's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.
<pre>short_name = string sn = string</pre> If a register's short name is defined, the short name instead of the tag name will be displayed in Register Explorer windows.
<pre>alias = {string_list}</pre> A register's aliases. They will be displayed in the Register View window, the Register Information window, and the Register Search window.
<pre>long_name = string ln = string</pre> This is the register's long name. It is displayed in the following places: • As a tooltip in the Register View window • In the help pane of the Register Information window
<pre>gui_tab = string</pre> This setting allows a register to be associated with a Register View window tab. The Debugger will create the specified tab and display the register in the tab.
<pre>cache_value = bool</pre> This setting indicates whether the Debugger can cache a register's value to improve performance while the target is halted. The default value is <code>true</code>, which is okay for most registers. For some special registers, such as the timer, the setting should be defined as <code>false</code> so that when you manually refresh values from the Register View window or print values in the Debugger window, the values can be fetched from the target even though the corresponding process is stopped on the target.

```
access = string
```

The string must be one of the following strings or an expression that can be resolved into one of the following strings:

- `regular` — [default] A physical register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `fpv` — A floating point register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `special` — A special coprocessor register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `io` — An input/output register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `synonym` — A synonym for another defined register. The address syntax is `name|expr` and resolves to the name of the register for which this register is a synonym.
- `memorymapped` — The contents of the register correspond to a location in memory. The address syntax is `num|name|expr`.

If the address field is a number, it indicates the address in memory of the start of the register's contents.

If the address field is not a number, then it is assumed to be string containing a Debugger expression that will evaluate to an address. This expression is evaluated dynamically each time the register is accessed to produce an address, and may depend on other registers. This expression is *not* surrounded by `%EVAL{ }`.

An expression value surrounded by `%EVAL{ }` is evaluated when the file is loaded, producing either a numerical address or a string that contains a Debugger command to be evaluated at run time to produce an address.

- `dynamic` — The contents of the register are obtained by evaluating a Debugger expression. The address syntax is `name|expr`.

The value is a string that contains an expression in the language being debugged and which is evaluated to produce the register's value each time the register is accessed. This expression is *not* surrounded by `%EVAL{ }`.

An expression value surrounded by `%EVAL{ }` is evaluated when the file is loaded, producing a string that contains a Debugger command to be evaluated at run time to produce a value.

- `indirect` — The register's value is obtained by writing its address into an address port and reading its value from a data port. The address syntax is `num|expr` and resolves to the control value to be written to the address port so the register value will be made available on the data port.

<code>address = number_or_string</code> For a register of type <code>regular</code>, <code>special</code>, <code>io</code>, or <code>fpu</code>, the address must be a number, or a string that can be immediately resolved into a number. For a <code>synonym</code> register, the address should be another register name or a string that can be immediately resolved into another register name. The register name used here should not contain the <code>\$</code> prefix. For a <code>memorymapped</code> register, the address must be a number, or an expression that can be resolved into an address immediately or at the time it is accessed. For a <code>dynamic</code> register, the address should be an expression that can be resolved into a number immediately or at the time it is accessed. For an <code>indirect</code> register, the address can be a number or an expression that can be resolved into a number immediately or at the time it is accessed.
<code>offset = string</code> The value must be a string or an expression that can be resolved into a number when the register is accessed. The resolved number is the register's ID used by the Debugger to communicate with the debug server. If the setting is absent and <code>register_offset</code> is defined in the <code>general</code> section, then the string value of <code>register_offset</code> from the <code>general</code> section will be used as the register's offset.
<code>address_port = number_or_string</code> This setting defines an indirect register's address port. The address port value must be a number, a register name or an expression that can be resolved into a number or register name. If it is an expression that can be resolved into a register name, it should be resolvable immediately.
<code>address_mask = number_or_string</code> This setting defines the mask for the control value to be written to an indirect register's address port. Its value must be a number or an expression that can be resolved into a number immediately.
<code>data_port = number_or_string</code> This setting defines an indirect register's data port, from which the indirect register's value is read. The data port value must be a number, a register name or an expression that can be resolved into a register name or a number. If it is an expression that can be resolved into a register name, it should be resolvable immediately. An indirect register's data port and address port must both be register names or both be addresses.

```
width = number_or_string
```

This setting specifies a register's width in bits. If the attribute is not explicitly specified, the default register width will be used.

The value must be a number or a string expression that can be immediately resolved into a number.

```
byte_endian = string
```

This setting defines a register's byte order, and overrides the byte order defined in the `general` section. For the available values and other details, see `byte_endian` attribute in “The general Section” on page 777.

If the setting is absent, the byte order defined in the `general` section will be used for the register.

```
bit_0_is_msb = bool
```

This setting defines a register's bit order.

If the value for `bit_0_is_msb` is true, bit 0 will refer to the register's most significant bit (msb), and the largest bit number will refer to the register's least significant bit (lsb). Otherwise, bit 0 will refer to the register's least significant bit (lsb), and the largest bit number will refer to the register's most significant bit (msb).

If the attribute is not explicitly specified, the default bit order defined in the `general` section will be applied.

`permission = string`

The value must be a string in the syntax specified below or an expression which can be resolved into such a string immediately:

`action/permission[/user_mode] [;action/permission[/user_mode]]`

where:

- *action* can be one of the following values:
 - read
 - write
 - both [default]
- *permission* can be one of the following values:
 - none
 - once
 - full [default]
- *user_mode* can be one of the following values:
 - user
 - supervisor
 - both [default]

`type = string`

The value must be one of the following strings or an expression that can be immediately resolved into one of the strings:

- unsigned
- signed
- A defined bitfield name
- A defined enumeration name
- A defined structure name
- A pointer to one of the above types or a pointer to void, such as `unsigned *` or `void *`

`read_length = number_or_string`

The value must be a number or an expression that can be resolved into a number immediately.

The attribute is only valid for `memorymapped` registers or `indirect` registers located in memory. If no read length is specified for such registers, the default read block length is the length of the whole register.

```
write_length = number_or_string
```

The value must be a number or an expression that can be resolved into a number immediately.

The attribute is only valid for `memorymapped` registers or `indirect` registers located in memory. If no write length is specified for such registers, the default write block length is the length of the whole register.

```
hide = bool_or_string
```

The value must be a Boolean value or a string expression that can be resolved into a Boolean immediately.

This setting indicates whether a register should be hidden or displayed.

```
memory_space = number
```

This setting defines the memory space for memory-mapped registers.

For more information, see the `memory_space` attribute in “The general Section” on page 777.

The group Section

Registers appear in groups. A register must be included in at least one group, but a register can appear in multiple groups, if desired. If a register is not explicitly included by any group, it will be included in a default group named `Ungrouped`.

A group can contain other groups as well as registers. A group can be included by at most one other group. If a group is not included by any other group, it is a top-level group. A group must contain at least one register or group; otherwise, it will be automatically hidden.

Groups and registers compose the hierarchy in the Debugger's **Register View** window.

A group definition can have the following elements:

```
group {  
    group_name {  
        description = string  
        help_key = string  
        short_name = string  
        long_name = string  
        top_level_index = number_or_string  
        collapse = bool_or_string  
        register = {registers}  
        group = {groups}  
    }  
    ...  
}
```



Note

All attributes are optional.

```
description = string
```

```
description += string
```

```
desc = string
```

```
desc += string
```

This is the group's description. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
help_key = string
```

```
hk = string
```

This is the group's help key. MULTI 6 does not use this value, which is currently being reserved for a future release.

```
short_name = string
```

```
sn = string
```

The value must be a string or an expression that can be immediately resolved into a string.

If group's short name is defined, it will be used to represent the group in the Debugger's **Register View** window, otherwise, the group's tag name is used.

```
long_name = string
```

```
ln = string
```

The value must be a string or an expression that can be immediately resolved into a string.

A group's long name is displayed in a tooltip in the **Register View** window.

```
top_level_index = number_or_string
```

The value must be a number or a string expression that can be immediately resolved into a number.

The attribute will be used to determine the group's position at top level if it is a top-level group, or inside a group if it is included by another group.

```
collapse = bool_or_string
```

The value must be a Boolean or a string expression that can be immediately resolved into a Boolean.

If this attribute is `true`, the Debugger collapses the group. If `false`, the Debugger expands the group. The default value for groups created on the fly is `false`. Otherwise, the default value is `true`.

Note: In GRD files with a `version` of 1 or 2, the Debugger initially collapses all nested groups and expands all top-level groups, regardless of the specification in the GRD file. In GRD files with a `version` of 3, the Debugger follows the specification in the GRD file.

Note: The **Register View** window remembers the value of this attribute across debugging sessions for targets of the same type. This information overrides the specification in the GRD file.

```
register = {registers}
```

```
register += {registers}
```

The strings used in the value to represent registers must be register tag names. The registers listed in the value must be defined.

```
group = {groups}
```

```
group += {groups}
```

The strings used in the value to represent groups must be group tag names. The groups listed in the value must be defined.

Appendix G

Integrate Views

Contents

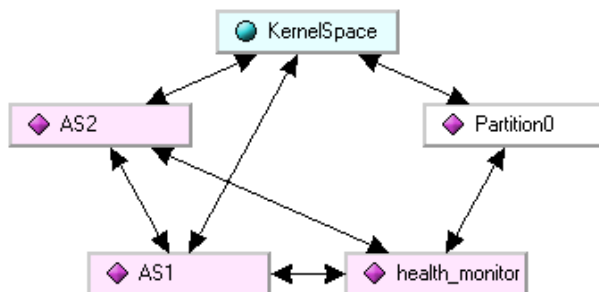
Integrate Security View	802
Integrate Relationship View	802

The Integrate GUI provides an easy way to create a customized configuration file based on the specific requirements of your INTEGRITY application. In MULTI 6, the stand-alone Integrate application adds two new views and the ability to group objects into collections. This appendix briefly describes each of these views and explains collections. Note that the lightweight version of the Integrate utility that appears in the **Integrate** tab of the MULTI Project Manager does not enable these new features.

If you are using INTEGRITY 12 or newer, see the *Integrate User's Guide* for up-to-date, complete information.

Integrate Security View

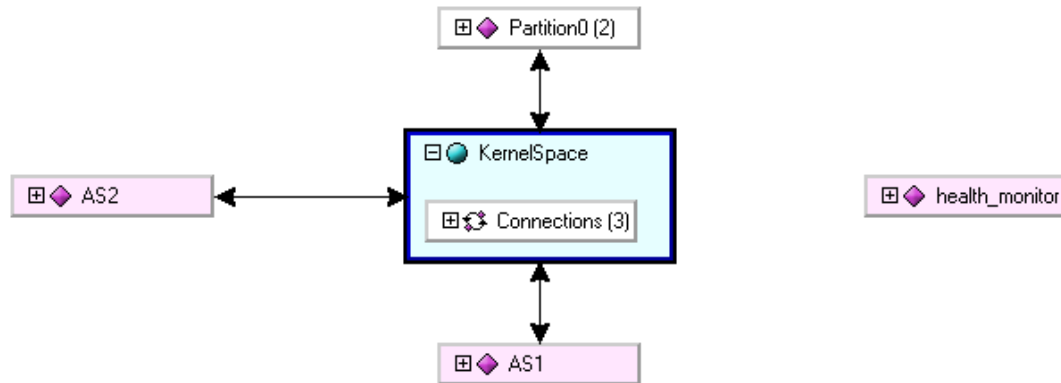
Security view displays relationships between AddressSpaces and AddressSpace collections defined in the Integrate file, thereby helping you to find unintended avenues of communication between AddressSpaces. An example is shown next.



Lines between AddressSpaces or AddressSpace collections indicate a relationship. A double-sided arrow is displayed between any two AddressSpaces or collections that contain underlying linked objects. Linked objects include connection pairs, links to objects, and mapped MemoryRegions.

Integrate Relationship View

Relationship view displays relationships between a central AddressSpace or AddressSpace collection and other AddressSpaces or AddressSpace collections. This view allows you to edit individual AddressSpaces, collections, and underlying objects while minimizing the number of lines drawn on the screen. An example is shown next.



Collections of AddressSpaces may be featured in both security view and in relationship view, but in relationship view, other objects are also automatically grouped by type into collections. A collection is a GUI-only, hierarchical grouping of objects that are defined in your Integrate file. You can also create your own collection of objects within an AddressSpace.

Collections ease viewing by giving you the ability to contract everything but the one or two objects that require closer examination. This is especially helpful when your file contains many objects. To expand or contract a hierarchy, click the +/- box.

Index

Symbols

- > (greater than sign)
 - in C++ classes, 195
- # (number sign) variable search designator, 293, 298
- \$ (dollar sign) variable search designator, 298
- * (asterisk) wildcard character, 303
- (minus sign-double) command line option, 722
- >* (minus sign, right angle bracket, asterisk) operator, 296
- . (period) variable search designator, 299
- .* (period, asterisk) operator, 296
- /* */ (slash, asterisk-double, slash) comments, 295
- // (slash-double) comments, 295
- :
- : (colon) variable search designator, 298
- :: (colon-double) variable search designator, 298
- < (left angle bracket) command
 - menu equivalent, 690
- = (equal sign) operator, 294
- == (equal sign-double) operator, 294
- > (right angle bracket) command
 - menu equivalent, 689, 690
- >> (right angle bracket-double) command
 - menu equivalent, 689, 690
- ? (question mark)
 - wildcard character, 303
- @ (at sign) wildcard character, 303

A

- Abort Trace Retrieval menu item, 683
- About banner
 - disabling with -nosplash, 725
- About dialog box, 691
- About MULTI menu item, 691
- active group, 177, 180
- addhook command, 94
- address bus walking test, 525
- addresses
 - source, expressing in Debugger, 293
 - viewing program at, 163, 677
- AddressSpaces

- freeze-mode (OSA), 16
- profiling, 355, 359
- run-mode, 581
- in Trace List, 436
- viewing individually in PathAnalyzer, 434
- Advanced Event Editor
 - complex events, specifying, 443, 497, 504
 - count expressions, creating, 505
 - event type descriptions, 501
 - exception type descriptions, 502
 - resources, creating, 498, 499
 - state machine expressions, inserting, 506
 - state machine resources, creating, 502
- All Types menu item, 679
- alternate directories
 - ignoring with -D, 723
 - specifying for search with -I, 724
- annotations (see Debugger Notes)
- ANSICMODE system variable, 310
- any-task breakpoints
 - freeze-mode, 420, 618
 - run-mode, setting, 587
- arguments
 - passing with -- (minus sign-double), 722
- Arguments dialog box, 668
- arrays in Data Explorer, 193
- arrows in source pane, 22, 123
- _ASMCACHE system variable, 312
- assem command, 677
- Assembly button, 694
- assembly code
 - infinite scrolling, 156
 - interlaced with source code, 23
 - viewing in Debugger, 23
- Assembly Only menu item, 677
- associating executable and connection
 - methods for, 105
 - on multi-core system, 626, 627
- asterisk (*) wildcard character, 303
- at sign (@) wildcard character, 303
- Attach on Fork/Thread menu item, 670
- Attach on Task Creation menu item, 672
- Attach to Process menu item, 662
- attaching to a process, 662
 - with -P, 725
- attaching to a task, 582
- Auto Check Coherency menu item, 671
- Auto Update Views menu item, 676
- _AUTO_CHECK_COHERENCY system variable, 312
- _AUTO_CHECK_NUM_ADDRS system variable, 312

B

- bc command, 416
- bcU command, 416
- beeping
 - in incremental Debugger search, 157
- bitfield definitions
 - changing, 276
- block detailed profiling report, 370
- board setup scripts
 - commands for, 94
 - created by Project Wizard, 90
 - creating custom, 90
 - early, 101
 - editing, 91
 - example, 96
 - MULTI (.mbs) vs. legacy (.dbs), 91, 98
 - multi-core system, 623
 - need for, 90
 - running manually, 101
 - running on connection, 99, 100
 - testing, 94
- bookmarking trace data, 432, 444, 445, 446, 448
- Bookmarks menu item, 691
- \$bp_adr() special operator, 318
- bprev command, 416
- bpview command, 128
- _BREAK system variable, 314
- breakdots, 22, 125
 - shortcut menu for, 707
- breakpoints, 124
 - (see also Breakpoints window)
 - any-task, freeze-mode, 420, 618
 - any-task, run-mode, 587
 - deleting, 129
 - disabling, 127
 - enabling, 127
 - freeze-mode, 420, 618
 - hardware, 133
 - (see also hardware breakpoints)
 - information about, 128
 - jump, 707
 - listing, 678
 - markers, 22, 125
 - moving (see software breakpoints)
 - overview, 124
 - restoring deleted, 137
 - run-mode, 587
 - run-mode, group, 588, 589
 - Separate Session TimeMachine, 422
 - setting basic, 22, 129
 - shared between TimeMachine and live target, 417
 - shared object, 136
 - software, 128
 - (see also software breakpoints)
 - task groups, using when setting, 589
 - task-specific, 420, 587, 618
 - toggling status of, 127
- Breakpoints button, 128, 695
- Breakpoints menu item
 - in View menu, 128, 673
 - in ViewList submenu, 678
- Breakpoints Restore window, 152
- Breakpoints window, 128
 - (see also breakpoints)
 - hardware breakpoints, 133, 135, 146, 148, 149, 150
 - opening, 128, 673
 - shared object breakpoints, 136, 146, 148, 149, 150
 - shortcut menus, 150
 - software breakpoints, 131, 146, 148, 149, 150
 - tracepoints, 571, 572, 573
- browse command, 377
- Browse menu, 679
- Browse window
 - cross references, browsing, 236, 238
 - data types, browsing, 233, 234, 235
 - filtering content in, 216, 217, 218, 219
 - global variables, browsing, 228, 230
 - headings, 220, 227, 233, 235
 - menus, 215
 - overview, 214
 - procedure ambiguities, 227
 - procedures, browsing, 166, 220, 222, 224, 227
 - source files, browsing, 165, 230, 231, 232, 233
- browsing
 - calls, classes (see Tree Browser)
 - classes, includes (see Graph View window)
 - cross references, data types, global variables, procedures, source files (see Browse window)
 - trace data, 431, 451, 452, 454, 456, 457
- bs command, 417
- build command line option, 722
- buttons, 691
 - (see also toolbar)
 - adding to toolbar, 698
 - commands for, 716
 - configuring, 698
 - in Debugger, 692
 - removing from toolbar, 698
 - reprogramming, 700

C

- c command, 94
- c command line option, 722
- C command line option, 722
- C++
 - application, TimeMachine API, 473
 - classes in Data Explorer, 195, 196
 - command line procedure calls, caveats for, 306
 - expressions and language dependencies, viewing, 303
 - expressions, caveats for, 296
 - unsupported operators, casts, destructors in expressions, 296, 306
- Cache Find window, 396
- Cache Memory Reads menu item, 671
- _CACHE system variable, 313
- Cache View window, 392
- caches
 - searching, 392, 396
 - viewing, 392, 396
- Caches menu item, 675
- call count profiling data, 354, 355, 360, 475
- call graph
 - profiling data, 354, 355, 475
 - profiling report, 367
- Call Stack button, 695
- Call Stack menu item, 673
- Call Stack window
 - buttons, 388
 - call stack pane, 389
 - call stack, viewing, 388
 - command line procedure calls, 389
 - functions, listing, 695
 - interrupt/exception handlers, 390
 - opening, 388, 673
 - procedure prologues and epilogues, 390
- calls
 - browsing, 213, 244, 245, 246
- casts, in C++
 - not supported in expressions, 296
- caveats
 - for command line procedure calls, 306
 - for expressions, 294
 - for profiling, 360
- .cfg files, ignoring with -nocfg, 724
- classes
 - browsing, 213, 243
- Classes menu item, 679
- Clear Data menu item, 683
- Clear User Default Configuration menu item, 687
- clearhooks command, 94
- clearing
 - profiling data, 384
 - trace, 410
- Close All Views menu item, 676
- Close Debugger button, 697
- Close Debugger Window menu item, 663
- Close Entry menu item, 662
- closing multiple windows, 676
- cmd command line option, 722
- Cmd pane, 28
- coherency checking, 536
- collecting
 - OS trace data, 406
 - profiling data, 356, 357, 358, 359
 - trace, 405
- collections, Integrate, 803
- colon (:) variable search designator, 298
- colon-double (::) variable search designator, 298
- command groups
 - register commands, 282
- command line
 - starting MULTI from, 6, 722
- command line options
 - (minus sign-double), 722
 - build, 722
 - c, 722
 - C, 722
 - cmd, 722
 - config, 722
 - configure, 722
 - connect, 7, 723
 - connectfile, 723
 - D, 723
 - data, 723
 - display, 7, 723
 - e, 723
 - E, 724
 - h, 7, 724
 - H, 7, 724
 - help, 7, 724
 - I, 724
 - listing with -h, 7, 724
 - m, 724, 728
 - minus sign-double (--), 722
 - nocfg, 724
 - nodisplay, 724
 - norc, 7, 725
 - noshared, 725
 - nosplash, 725
 - osa, 607, 725
 - p, 7, 725

- P, 725
- passing in a specification file, 727
 - (see also -m command line option)
- preload, 725
- r, 725
- R, 726
- rc, 726
- RO, 726
- run, 726
- servvertimeout, 726
- socket, 726
- text, 726
- top, 727
- tv, 727
- usage, 727
- V, 727
- command line procedure calls
 - call stack for, 389
 - caveats for, 306
 - in Debugger, 304
 - unsupported class members, operators in C++, 306
- command pane
 - Debugger, 28, 172
 - shortcut menus, 712
 - shortcuts, 29, 718
- command script, reading on startup, 726
- commands
 - < (left angle bracket)
 - menu equivalent, 690
 - > (right angle bracket)
 - menu equivalent, 689, 690
 - >> (right angle bracket-double)
 - menu equivalent, 689, 690
 - addhook, 94
 - assem, 677
 - b, 130
 - bc, 416
 - bcU, 416
 - board setup script, 94
 - bprev, 416
 - bpview, 128
 - browse, 377
 - bs, 417
 - c, 94
 - clearhooks, 94
 - connect -restart_runmode, 71
 - dataview, 746, 747, 767
 - Debugger button equivalents, 716
 - dialogsearch, 712
 - dvclear, 745
 - dvload, 745
 - dvprofile, 744, 766
 - e, 163
 - edithwbp, 134
 - editswbp, 130
 - eval, 94
 - flash gui, 541
 - h, 689
 - halt, 95
 - hook, 101, 623
 - left angle bracket (<)
 - menu equivalent, 690
 - legacy debug scripts, 91
 - memread, 95
 - memtest, 535
 - memwrite, 95
 - passive, 574, 575
 - profdump, 377, 382
 - profile, 357, 359
 - profilemode, 373, 374, 375, 376, 377, 378, 381, 382, 383, 384
 - profilereport, 363, 364, 372
 - recording to playback files, 725, 726
 - reset, 95
 - restore, 689
 - right angle bracket (>)
 - menu equivalent, 689, 690
 - right angle bracket-double (>>)
 - menu equivalent, 689, 690
 - save, 689
 - sb, 587, 619
 - serialconnect, 686
 - serialdisconnect, 686
 - set_runmode_partner, 71
 - target, 95
 - TimeMachine run-control, 416
 - timemachine, 414, 415, 418, 422
 - tpdel, 564
 - tpenable, 565
 - tplist, 564
 - tpprint, 566
 - tppurge, 567
 - tpreset, 565
 - tpset, 562, 568, 570, 571
 - trace abort, 408
 - trace clear, 411
 - trace disable, 406
 - trace enable, 406
 - trace retrieve, 408
 - traceload, 450
 - tracemevsys, 475
 - tracepro, 358

- tracesave, 449
- update, 189
- view, 184
- viewdel, 185
- viewlist, 192
- wait, 95
- comments
 - in expressions, 295
- Compare menu item, 682
- compare test, CRC, 533
- compilers
 - generating debugging information, 5
- complex events, specifying with Advanced Event Editor, 443, 497, 504
- compressed target list display, 17
- compute test, CRC, 532
- concise display pane in Register Information window, 272
- conditionally not-executed instructions, 123
- config command line option, 722
- Config menu, 687
- configuration file, 722
- configuration options
 - Initial Position (XxY), 200
 - Maximum initial size (WxH), 200
 - Minimum initial size (WxH), 200
 - osaSwitchToUserTaskAutomatically, 614
 - osaTaskAutoAttachLimit, 614
 - ProcRelativeLines, 24
- configure command line option, 722
- Configure Trace Interval window, 492
- configuring
 - buttons, 698
 - line numbers in source pane, 24
 - MULTI with -c, 722
 - toolbar, 698
- connect -restart_runmode command, 71
- connect command line option, 7, 723
- Connect menu item, 680
- connected targets
 - managing from Connection Organizer, 54
- connectfile command line option, 723
- connecting
 - from Connection Organizer, 54
 - to debug server with -connect, 7, 723
 - to hardware (freeze mode) and OS (run mode), 69
 - troubleshooting, 73
 - to OS (run mode), 578
 - to process with -P, 725
 - to serial port (see serial connections)
 - to simulator, 41
 - to target
 - Custom Connection Method, 47
 - Standard Connection Method, 45
 - Temporary Connection Method, 49
- connecting MULTI to your target
 - through a terminal server, 68
- Connection Chooser
 - creating Custom Connection Method, 47
 - creating Standard Connection Method, 43
 - opening, 43
- Connection Editor
 - configuring Standard Connection Method, 44
 - INDRT (rtserv), 80, 81, 82, 83
 - INDRT2 (rtserv2), 62, 63, 64
 - opening, 44
- Connection Files, 52
- Connection Methods
 - connecting with, 45, 54
 - creating, 52
 - Custom, 42, 47
 - custom
 - INDRT (rtserv), 86
 - INDRT2 (rtserv2), 66, 67, 68
 - editing, 44, 52
 - INDRT (rtserv), 80, 81, 82, 83
 - INDRT2 (rtserv2), 62, 63, 64
 - examples
 - INDRT2 (rtserv2), 68, 69
 - overview, 40
 - Serial, 644, 646, 647, 649
 - Standard, 42, 43, 44
 - Temporary, 49
- Connection Organizer
 - connected targets in, 54, 57
 - connecting to target from, 54
 - Connection Files in, 52, 55
 - Connection Methods in, 52, 56
 - opening, 50
- CONTINUECOUNT system variable, 310
- CONTINUING status bar message, 36
- conventions
 - typographical, xxvi
- converting debugging information (see translating, debugging information)
- Copy menu item, 681
- copying text
 - in main Debugger window, 161
- core file
 - debugging, 117
 - specifying with -C, 722
- cores, multiple (see multi-core systems)
- coverage analysis profiling data, 354, 355, 360, 475

- coverage profiling report, 369
- CPU % target list column, 20, 603
- CRC compare test, 533
- CRC compute test, 532
- cross references
 - browsing, 213, 236, 238
- current line pointer, 22
- Current PC menu item, 163, 676
- _CURRENT_TASKID system variable, 314
- Custom Connection Methods, 42, 47
- Customize Menus menu item, 687
- Customize Toolbar menu item, 687
- Customize Toolbar window, 698
- cutting text in Debugger windows, 160

D

- D command line option, 723
- data bus walking test, 527
- data command line option, 723
- data descriptions
 - MULTI data visualization (.mdv) files, 743, 747
- Data Explorer
 - activating variables, 186, 189
 - adding variables to, 187
 - arrays, viewing, 193
 - buttons, 186
 - C++ classes, viewing, 195
 - closing, 184
 - custom, 742, 746
 - dimensions, 198, 200
 - edit bar, 187
 - freezing variables, 186, 189
 - global options for, 199
 - limiting data complexity, 199
 - linked lists, viewing, 191
 - messages, 200
 - modifying variables in, 187, 198
 - opening, 184, 188
 - pointers, viewing, 193, 196
 - structures, viewing, 190
 - toolbar, 186
 - types, changing, 195
 - updating, 189
 - viewing multiple items in, 188
 - window, 185
- ... Data Explorer message, 200
- data offsets, position-independent data (PID)
 - specifying with -data, 723
 - specifying with _DATA, 313
 - specifying with Data Offset, 112

- data pattern test, 529
- _DATA system variable, 313
- data types
 - browsing, 213, 214, 233, 234, 235
 - profiling, 354
- data visualization (see MULTI data visualization)
- data visualization profile, 744, 766
- dataview command, 746, 747, 767
- Dead Data Explorer message, 200
- Debug menu, 665
- Debug Program as New Entry menu item, 661
- Debug Program menu item, 661
- debug servers
 - connecting to with -connect, 7, 723
 - debugging interfaces supported by, 90
 - setting default timeout with -servvertimeout, 726
 - starting, 47, 49
 - use, 42
- Debug Settings menu item, 668
- Debug Translator
 - translating debugging information, 730
- DebugBrk target list status, 19
- Debugger, 4
 - (see also debugging)
 - command line options (see command line options)
 - connecting to your target, 40
 - graphical (GUI) mode, 6, 7, 723
 - menus (see Debugger menus)
 - non-GUI mode, 8, 724
 - notes in code (see Debugger Notes)
 - overview, 4
 - passive mode, 574, 575
 - searching source, 167, 712
 - special operators, 317
 - starting, 5
 - system variables, 310, 312
 - version information, 727
 - window (see Debugger window)
- Debugger Help menu item, 691
- Debugger menus
 - Browse, 679
 - Config, 687
 - Debug, 665
 - File, 661
 - Help, 691
 - Target, 680
 - TimeMachine, 683
 - Tools, 685
 - View, 673
 - Windows, 690
- Debugger Notes

- ul style="list-style-type: none;">
- browsing, 178
- creating, 174
- editing, 174, 176
- grouping, 177, 178
- markers, 22
- overview, 174
- properties, 174
- Debugger Notes menu item, 675
- Debugger window
 - assembly code, viewing, 23
 - breakdots, 22, 125
 - breakpoint markers, 126, 129
 - button-command equivalents, 716
 - buttons, 691, 692
 - (see also buttons)
 - navigation buttons, 167
 - closing, 663
 - command pane, 28, 172
 - controlling process from, 122
 - copying text, 161
 - current line pointer, 22
 - cutting text, 160
 - Debugger Note marker, 22
 - File Locator, 164
 - I/O pane, 30
 - information pane, 13, 26
 - interlaced assembly code, viewing, 23
 - line numbers, 21, 24
 - menu bar, 13, 660
 - (see also Debugger menus)
 - navigation bar, 13, 25
 - navigation buttons, 167
 - opening, 5
 - opening multiple, 14
 - output pane, 13, 26
 - overview, 12
 - pasting text, 160, 161
 - PC pointer, 22, 123
 - Procedure Locator, 165
 - python pane, 32
 - reusing, 14
 - scroll bar with diamond, 156
 - selecting text, 160, 161
 - serial terminal pane, 31
 - shortcut menus, 704, 705, 707, 708, 709, 710, 711, 712
 - shortcuts, 716
 - source code, viewing, 23
 - source pane, 13, 21, 162
 - (see also source pane)
 - status bar, 14, 36
 - target list, 15
 - (see also target list)
 - target pane, 30
 - toolbar (see toolbar)
 - tracepoint markers, 126
 - traffic pane, 33
- debugging
 - core files, 117
 - disabling for shared libraries with -noshared, 725
 - in freeze mode, 606
 - (see also freeze-mode debugging)
 - multi-core systems, 620
 - (see also multi-core systems)
 - multiple files with -E, 724
 - multiple-language applications, 294
 - multitasking applications, 606
 - native processes, 390
 - non-intrusively with tracepoints, 560
 - operating systems (see freeze-mode debugging) (see run-mode debugging)
 - in passive mode, 574, 575
 - preliminary steps, 5
 - programs compiled with third-party tools, 730
 - programs with specification files, 727
 - (see also -m command line option)
 - real-time operating systems
 - INDRT (rtserv), 78
 - INDRT2 (rtserv2), 60
 - ROM programs, 554
 - in run mode, 578
 - (see also run-mode debugging)
 - run-mode
 - INDRT (rtserv), 78
 - INDRT2 (rtserv2), 60
 - tasks, 583
- debugging information, 5
 - generated by third-party tools, 730
- debugging interfaces
 - debug servers associated with, 90
- DEBUGSHARED system variable, 311
- default.con file, 42
- Defines menu item, 678
- Delete all view windows button, 700
- DEREFPOINTER system variable, 311
- destructors in C++, 296
- Detach from Process menu item, 662
- detailed display pane in Register Information window, 273
- Dialog Boxes menu item, 678
- dialog boxes, listing, 678
- dialogsearch command, 712
- diamond on Debugger scroll bar, 156
- Direct hardware access target list message, 18

- directories, alternate
 - ignoring with -D, 723
 - specifying for search with -I, 724
- Disable Trace menu item, 683
- disabling
 - profiling data collection, 360
 - TimeMachine, 415
 - trace collection, 406
- Disassemble From Host menu item, 671
- discarding trace, 410
- Disconnect from Serial menu item, 686
- Disconnect from Target menu item, 58, 680
- disconnecting
 - from target, 58
- DISNAMELEN system variable, 311
- display MULTI command line option, 7, 723
- DisplayMode menu item, 673
- _DISPMODE system variable, 313
- document set, xxiv, xxv
- dollar sign (\$) variable search designator, 298
- dots in source pane, 22, 125
- Download Program button, 701
- Download to RAM option, 114
- downloading
 - modules with -preload, 725
 - separate executables on multi-core system, 627
 - single executable on multi-core system, 626
 - to RAM, 114
- Downstack button, 695, 717
- DownStack menu item, 163, 676
- Dump and Show Events button, 696
- dump, core file, 117
- dumping profiling data, 381
- dvclear command, 745
- dvload command, 745
- dvprofile command, 744, 766
- DWARF debugging information
 - converting, 730
- DYING status bar message, 36
- Dynamic Calls menu item, 679

E

- e command, 163
- e command line option, 723
- E command line option, 724
- early MULTI board setup scripts, 101
- edit bar
 - in Data Explorer, 187
- Edit button, 696
- Editor menu item, 685

- Editor, MULTI, 170
- Empty Data Explorer message, 200
- Enable Trace menu item, 683
- enabling
 - OS trace collection, 406
 - TimeMachine, 414
 - trace collection, 405
- \$entadr() special operator, 318
- entry label, specifying with -e, 723
- _ENTRYPOINT system variable, 314
- epilogue code
 - debugging, 390
- equal sign (=) operator, 294
- equal sign-double (==) operator, 294
- error checking, run-time (see run-time error checking)
- eval command, 94
- evaluating
 - expressions, 292
- EventAnalyzer (see MULTI EventAnalyzer)
- EventAnalyzer menu item, 684
- events
 - complex
 - specifying with Advanced Event Editor, 443, 497, 504
 - trace
 - viewing in EventAnalyzer, 474
- examining data (see viewing)
- examples
 - Connection Methods, custom
 - INDRT2 (rtserv2), 68, 69
- exception handlers
 - call stack for, 390
- exception type descriptions, 502
- EXEC'ING status bar message, 36
- _EXEC_NAME system variable, 314
- Excing target list status, 19
- \$exists() special operator, 318
- Exit All menu item, 663
- Exited target list status, 19
- expressions
 - caveats for, 294
 - in Debugger commands, 292
 - evaluating, 292
 - formats for, 300
 - language-independent, 294
 - unsupported operators, casts, destructors in C++, 296
 - viewing with wildcards, 303

F

- F1 key, for help, 691
- Fast Source Step menu item, 672

- Fast-Find, PathAnalyzer, 430
- FASTSTEP system variable, 311
- field debugging (see non-intrusive debugging)
- File Calls menu item, 679
- file chooser dialog box (Linux/Solaris), 713
- file interface, TimeMachine, 467, 469, 470, 472
- File Locator on navigation bar, 164
- File menu, 661
- _FILE system variable, 314
- file-relative line numbers
 - displaying in source pane, 21, 24
- files
 - debugging multiple with -E, 724
 - listing, 678
 - recently opened from File menu, 662
 - searching, 159
- Files menu item
 - in Browse menu, 679
 - in ViewList submenu, 678
- Fill menu item, 681
- filtering content in
 - Browse window, 216, 217, 218, 219
- Find Address in Cache menu item, 675
- Find menu item, 682
- find start/end ranges test, 533
- flash gui command, 541
- flash memory
 - base address, setting, 541
 - erasing, 544
 - file, selecting, 542
 - flash bank, specifying, 541
 - MULTI Fast Flash Programmer, 540
 - operations, 541
 - option in Prepare Target dialog, 114
 - prerequisites to working with, 541
 - programming, 541, 543
 - troubleshooting problems, 547
 - verifying download to, 544
 - write offset, specifying, 542
- Flash menu item, 681
- flashing to ROM, 114
- formats for expressions, 300
- Freeze button
 - in Data Explorer, 186, 189
- freeze-mode AddressSpaces, 16
- freeze-mode breakpoints, 420, 618
- freeze-mode connections
 - concurrent with run-mode connections, 69
 - troubleshooting, 73
- freeze-mode debugging, 614
 - (see also OSA Explorer)

- AddressSpaces, 16
- breakpoints, 420, 618
- configuration file, 632
- I/O, 620
- in Debugger windows, 614, 615
- kernel, 613, 614
- Master Debugger mode, 614
- multi-core systems, 620
 - (see also multi-core systems)
- overview, 606
- starting, 607
- target environments supported, 606
- Task Debugger mode, 615
- task-specific single-stepping, 616
- tasks, 613, 614, 615
 - with TimeMachine, 418, 419
- freeze-mode tasks, 615
 - in TimeMachine mode, 418, 419
- functions (see procedures)

G

- generating profiling data, 356
- global variables
 - browsing, 214, 228, 230
 - listing, 678
- Globals menu item
 - in Browse menu, 679
 - in ViewList submenu, 678
- Go Back button, 692
- Go on Selected Items button, 693, 716
- Go on Selected Items menu item, 665
- Go to next selected location in trace data button, 701
- Go to previous selected location in trace data button, 701
- Goto Location menu item, 163, 677
- Graph View window
 - controlling layout in, 248
 - custom, 742, 746
 - history buttons, 251
 - includes, browsing, 247
 - Layout menu, 248
 - navigating, 248
 - Search for Objects window, 250
 - shortcut menus, 249
 - window, 247
- graphical (GUI) mode
 - starting Debugger in, 6, 7, 723
- GRD register definition files
 - format, 776
 - location of, 286
 - saving, 276

- greater than sign (>)
 - in C++ classes, 195
- Green Hills Monitors
 - setup scripts unnecessary for, 91
- grep utility, 160
 - (see also searching)
 - licensing, 160
- group breakpoints, run-mode, 588, 589
- GrpHalt target list status, 19

H

- h command, 689
- h command line option, 7, 724
- H command line option, 7, 724
- halt command, 95
- Halt on Attach menu item, 672
- Halt on Selected Items button, 693
- Halt on Selected Items menu item, 665
- Halt System menu item, 666
- Halted target list status, 19
- halting tasks on attach, 583
- hardware breakpoints
 - editing, 133
 - managing, 135, 146, 148, 149, 150
 - managing deleted, 152
 - overview, 133
 - properties of, 134, 139
 - setting, 133
 - viewing, 135, 146, 148, 149, 150
 - viewing deleted, 152
- hardware, target (see targets)
- headings
 - in Browse window, 220, 227, 233, 235
- help command line option, 7, 724
- Help menu, 691
- hierarchical target list display, 18
- hooks
 - in early MULTI board setup scripts, 101
 - in multi-core setup scripts, 623
- HostIO target list status, 19

I

- I command line option, 724
- I/O
 - program, with INTEGRITY, 620
 - redirection, 665
- I/O pane, 30
- ignoring
 - alternate directories with -D, 723
 - .cfg files with -nocfg, 724

- .rc files with -norc, 7, 725
- \$in() special operator, 318
- Included Files menu item, 678
- includes
 - browsing, 247
- Includes menu item, 679
- incremental searching
 - beeping in, 157
 - for strings, 156
- INDRT, 78
 - connecting MULTI to (see Connection Methods)
- INDRT2, 60
 - connecting MULTI to (see Connection Methods)
- infinite scrolling, 156
- information pane, 13, 26, 597
- _INIT_SP system variable, 313
- Initial Position (XxY) configuration option, 200
- input/output pane, 30
- input/output redirection, 665
- installing target hardware, 90
- instruction pane in Trace List, 436
- instructions
 - conditionally not executed, 123
 - dimmed, 123
 - stepping through, 666
- instrumented profiling, 356
- Integrate, 802
- INTEGRITY
 - freeze-mode debugging, 606
 - program I/O in freeze mode, 620
 - run-mode tasks, 422
 - Separate Session TimeMachine with, 422
 - TimeMachine with, 418, 419
 - trace data, 406, 436
- INTEGRITY Object Viewer button, 697
- interfaces, TimeMachine API, 467, 469, 470, 471, 472
- _INTERLACE system variable, 314
- interlaced assembly code
 - viewing in Debugger, 23
- Interlaced Assembly menu item, 677
- interrupt handlers
 - call stack for, 390
- interrupts
 - disabling during target setup, 93

J

- jump breakpoints, setting, 707

K

- kernel

debugging in freeze mode, 613, 614

keyboard shortcuts (see shortcuts)

keywords, language, 294

Kill Selected Items menu item, 665

L

language dependencies, 303

language keywords, 294

_LANGUAGE system variable, 313

language-independent expressions, 294

_LAST_COMMAND_STATUS system variable, 314

_LAST_CONNECT_CMD_LINE system variable, 314

Launch Utility Programs menu item, 685

Launcher menu item, 685

left angle bracket (<) command

menu equivalent, 690

legacy debug server setup scripts (.dbs) (see board setup scripts)

lifetime information for variables, 299

line numbers

displaying in source pane, 21, 24

jumping to with line pointer, 22

memory address of, 293

program counter, 22, 123

line pointer, 22

_LINE system variable, 314

_LINES system variable, 313

linked lists in Data Explorer, 191

linker directives files

created by Project Wizard, 90

editing, 98

need for, 90

Linux/Solaris

file chooser dialog box, 713

printing, 663

List menu item, 675

listing

breakpoints, 678

command line options with -h, 7, 724

dialog boxes, 678

files, 678

global variables, 678

included source files, 678

local variable addresses, 678

local variables, 678

macros, 678

mangled procedures, 678

procedure variables, 678

procedures, 678

processes, 678

register synonyms, 678

registers, 678

signals, 678

source paths, 678

special variables, 678

static variables, 678

lists

selecting items from, 161

live TimeMachine interface, 467, 471

Load Configuration menu item, 688

Load Module menu item, 680

Load Trace Session menu item, 683

loading (see downloading)

Local Addresses menu item, 678

local variables, 673, 678

Local Variables menu item, 673

Locals (\$locals\$) window, 169

Locals button, 695

Locals menu item, 678

-log option

for rtserve2, 67, 68

logging

INDRT (rtserve), 83

INDRT2 (rtserve2), 64, 67, 68

M

-m command line option, 724, 728

\$M_called_from() special operator, 318

\$M_can_read_address() special operator, 318

\$M_file_exists() special operator, 318

\$M_get_temp_memory() special operator, 318

\$M_offsetof() special operator, 319

\$M_sec_begin() special operator, 319

\$M_sec_end() special operator, 319

\$M_sec_exists() special operator, 319

\$M_sec_size() special operator, 319

\$M_strcaseprefix() special operator, 319

\$M_strcmp() special operator, 319

\$M_strcpy() special operator, 319

\$M_strprefix() special operator, 319

\$M_strstr() special operator, 319

\$M_sym_exists() special operator, 318

\$M_typeof() special operator, 320

\$M_var_is_valid() special operator, 320

macros

calling, 307

evaluating, 307

listing, 678

shortcut menu, 711

Make Serial Connection menu item, 686

- Manage Bookmarks menu item, 691
- Mangled Procedures menu item, 678
- mangled procedures, listing, 678
- Manuals menu item, 691
- Master Debugger mode, 614
- master process, OSA, 614
 - in TimeMachine mode, 419
- Maximum initial size (WxH) configuration option, 200
- .mdv files (see MULTI data visualization (.mdv) files)
- memory
 - accessing from Memory View window, 328
 - allocation errors (see memory allocation)
 - coherency, 536
 - configuring via setup script, 93
 - editing from Memory View window, 329
 - flash (see flash memory)
 - leaks, finding, 347
 - manipulating in Memory Allocations window, 348
 - testing (see memory testing)
 - testing access to, 97
 - viewing, 324
 - (see also Memory View window)
 - from Data Explorer, 184
- memory allocation
 - tracking, 350
 - viewing, 343, 347
 - window (see Memory Allocations window)
- Memory Allocations menu item, 674
- Memory Allocations window
 - leaks, viewing, 347
 - menus, 342
 - opening, 340, 674
 - overview, 340
 - procedures, manipulating, 348
 - refreshing, 349
 - tabs, 341
 - tracking leaks and allocations, 350
 - updating, 346
 - visualization, 343
- Memory button, 695
- Memory Dump menu item, 682
- Memory Load menu item, 682
- Memory Manipulation menu item, 681
- Memory menu item, 674
- Memory Sensitive menu item, 671
- memory sensitive mode, 115
- Memory Test menu item, 681
- Memory Test Results window, 523
- Memory Test Wizard, 512
- memory testing
 - address bus walking test, 525
 - advanced, 515
 - coherency errors, 536
 - command line, 535
 - continuous, 524
 - CRC compare test, 533
 - CRC compute test, 532
 - data bus walking test, 527
 - data pattern test, 529
 - find start/end ranges test, 533
 - hints for efficient testing, 535
 - memory read test, 532
 - Memory Test Results window, 523
 - Memory Test Wizard, 512
 - overview, 512
 - Perform Memory Test window, 515, 516
 - (see also Perform Memory Test window)
 - quick, 512
 - results, viewing, 523
 - running, 522
 - selecting, 518
 - with target agent, 524
 - test area, specifying, 516
 - test method, specifying, 521
 - test options, setting, 520
 - types, 525
- Memory View window
 - active location, setting, 325
 - columns, 334
 - editing memory from, 329
 - formatting columns, 326
 - freezing, 328
 - infinite scrolling, 156
 - memory pane, 326
 - opening, 324, 674
 - overview, 324
 - updating, 328
- memread command, 95
- memtest command, 535
- memwrite command, 95
- menu bar, 13, 660
 - (see also Debugger menus)
- menus
 - Debugger (see Debugger menus)
- messages
 - Data Explorer, 200
 - status bar, 36
- Minimum initial size (WxH) configuration option, 200
- minus sign, right angle bracket, asterisk (->*) operator, 296
- minus sign-double (--) command line option, 722
- Modify Register Definition dialog, 276, 277, 278
- modules, downloading with -preload, 725

- Move splitter down button, 15
 - Move splitter left button, 15
 - Move splitter right button, 15
 - Move splitter up button, 15
 - Move target list to left button, 15
 - moving software breakpoints, 132
 - mterminal command, 654
 - MTerminal serial terminal emulator, 644
 - (see also serial connections)
 - configuring, 646, 647, 649
 - console mode, 653
 - creating Connection Methods, 646, 647, 649
 - overview, 644
 - as stand-alone application, 654
 - starting, 644
 - window features, 650
 - MULTI (see MULTI Integrated Development Environment (IDE))
 - MULTI board setup scripts (.mbs) (see board setup scripts)
 - MULTI data visualization
 - with Data Explorer, 191
 - invoking, 746
 - MULTI data visualization (.mdv) files (see MULTI data visualization (.mdv) files)
 - overview, 742
 - MULTI data visualization (.mdv) files
 - clearing, 745
 - data descriptions, 743, 747
 - expressions in, 770
 - file format, 747
 - loading, 745
 - overview, 742, 743
 - profile descriptions, 744, 766
 - type-specific fields, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763
 - view descriptions, 744, 768
 - MULTI Debugger (see Debugger)
 - MULTI Editor, 170
 - MULTI EventAnalyzer
 - pane in PathAnalyzer, 432
 - viewing trace events in, 474
 - MULTI EventAnalyzer button, 697
 - MULTI EventAnalyzer menu item, 685
 - MULTI Fast Flash Programmer
 - base address, setting, 541
 - erasing flash memory, 544
 - file, selecting, 542
 - flash bank, specifying, 541
 - opening, 540
 - operations, 541
 - prerequisites to working with flash, 541
 - programming flash memory, 541, 543
 - troubleshooting problems, 547
 - verifying flash download, 544
 - write offset, specifying, 542
 - MULTI Integrated Development Environment (IDE)
 - document set, xxv
 - exiting, 663
 - starting for Object Structure Awareness (OSA), 725
 - starting from command line, 6, 722
 - viewing information about, 691
 - MULTI ResourceAnalyzer button, 697
 - MULTI ResourceAnalyzer menu item, 685
 - MULTI Variables menu item, 678
 - multi-core systems
 - display in target list, 620
 - display in Task Manager, 621
 - hook commands in setup script, 623
 - limitations, 631
 - running separate executables on, 627
 - running single executable on, 626
 - synchronous run control, 629
 - _MULTI_DIR system variable, 314
 - _MULTI_MAJOR_VERSION system variable, 314
 - _MULTI_MICRO_VERSION system variable, 315
 - _MULTI_MINOR_VERSION system variable, 315
 - multiple files, debugging with -E, 724
 - multiple run-mode connections, 578
 - multiple-language applications, debugging, 294
 - multitasking applications, debugging, 606
- ## N
- NaN Data Explorer message, 200
 - native processes
 - viewing, 390
 - navigating
 - buttons for, 167
 - MULTI windows, 156
 - navigation bar, 13, 25
 - Navigation menu item, 673
 - Next (over Functions) on Selected Items button, 693, 717
 - Next (over Functions) on Selected Items menu item, 666
 - No process Data Explorer message, 200
 - NO PROCESS status bar message, 36
 - No Stack Trace menu item, 671
 - no stack trace mode, 116
 - No symbols for this procedure Data Explorer message, 201
 - No TimeMachine Data target list status, 19
 - nocfg command line option, 724
 - node operations in Tree Browser, 242
 - nodisplay MULTI command line option, 724

non-GUI mode
 starting Debugger in, 8, 724
non-intrusive debugging, 560
 (see also tracepoints)
 example, 567
 limitations, 570
 overview, 560
 with passive mode, 574, 575
_NONGUIMODE system variable, 315
-norc command line option, 7, 725
-noshared command line option, 725
-nosplash command line option, 725
Not Initialized Data Explorer message, 201
Not loaded target list status, 19
Note Browser, 178
Number of Addresses to Check menu item, 672
number sign (#) variable search designator, 293, 298

O

Object Structure Awareness (OSA), 607
 (see also OSA Explorer)
 starting MULTI for, 607, 725
offsetof() special operator, 319
offsets
 position-independent code (PIC), 112, 313, 726
 position-independent data (PID), 112, 313, 723
online help
 opening with -help, 7, 724
 opening with F1, 691
_OPCODE system variable, 313
Open project manager for current project button, 701
operating systems
 collecting trace for, 406
 debugging (see freeze-mode debugging) (see run-mode debugging)
 navigating trace for, 436
 tasks, launching TimeMachine on, 418
operators
 in count expressions, 506
 in state expressions, 507
 language-independent, 294
 not supported in expressions, 296, 306
 overloaded in C++, 304
 special (see special operators)
Optimized away Data Explorer message, 201
options (see command line options)
 rtserv2
 -log, 67, 68
 -serial, 68
Options menu item, 687

Original procedure not on stack Data Explorer message, 201
OS-awareness, 602, 606, 607, 632
 (see also freeze-mode debugging)
 (see also OSA Explorer)
 (see also OSA Object Viewer)
_OS_DIR system variable, 315
OSA AddressSpaces in target list, 16
-osa command line option, 607, 725
OSA Explorer, 609
 (see also Object Structure Awareness (OSA))
 displaying, 609
 menus, 611
 object list, 612
 opening, 674
 overview, 609
 shortcut menus, 612, 638
 toolbar, 611
OSA Explorer button, 696
OSA Explorer menu item, 674
OSA master process, 614
 in TimeMachine mode, 419
OSA Object Viewer, 602
OSA tasks, 615
 in TimeMachine mode, 418, 419
osaSwitchToUserTaskAutomatically configuration option, 614
osaTaskAutoAttachLimit configuration option, 614
output
 recording to playback files, 726
output pane, 13, 26
output/input redirection, 665

P

-p command line option, 7, 725
-P command line option, 725
pane-switching tabs, 13, 26
partner, run-mode (see run-mode partner)
passing arguments with -- (minus sign-double), 722
passive command, 574, 575
passive mode, 574, 575
pasting text
 in Debugger windows, 160
 in main Debugger window, 161
PathAnalyzer, 424
 (see also trace)
 analyzing OS data, 432
 bookmarks in, 432
 browsing references, 431
 EventAnalyzer pane in, 432
 Fast-Find, 430

- ul style="list-style-type: none;">
- navigating, 428, 429
- opening, 424
- searching, 430
- thumbnail pane, 427
- viewing a single AddressSpace, 434
- window, 425
- PathAnalyzer menu item, 684
- PC (program counter), 22
 - dimmed, 123
 - location in TimeMachine mode, 416
 - samples, 354, 355, 356, 360, 475
- Pended target list status, 19
- Perform Memory Test window, 515
 - running tests from, 522
 - test area, setting, 516
 - test method, specifying, 521
 - test options, setting, 520
 - test type, selecting, 518
- period (.) variable search designator, 299
- period, asterisk (.*) operator, 296
- _PID system variable, 315
- Playback Commands menu item, 690
- playback files, 7, 725, 726
- pointers in Data Explorer, 193, 196
- position-independent code (PIC)
 - offset, specifying with -text, 726
 - offset, specifying with _TEXT, 313
 - offset, specifying with Text Offset, 112
- position-independent data (PID)
 - offset, specifying with -data, 723
 - offset, specifying with _DATA, 313
 - offset, specifying with Data Offset, 112
- preload command line option, 725
- Prepare Target button, 694
- Prepare Target dialog box, 110
- Prepare Target menu item, 667
- preparing a target
 - basics, 108
 - by downloading, 114
 - by flashing, 114
 - by verifying, 114
 - multi-core, 626, 627
 - settings related to, 115, 116
- Previous button, 692
- Print Expression menu item, 675
- Print menu item, 661
- Print Setup dialog box, 663
- Print Window menu item, 661
- printing
 - on Linux/Solaris, 663
- Procedure Locator on navigation bar, 165
- _PROCEDURE system variable, 315
- procedure-relative line numbers
 - displaying in source pane, 21, 24
- procedures
 - ambiguities, Browse dialog box for, 227
 - browsing (see Browse window)
 - calling from command line, 306
 - calling from Debugger, 304
 - epilogues, 390
 - listing, 678
 - mangled, listing, 678
 - manipulating in Memory Allocations window, 348
 - in Procedure Locator, 165
 - prologues, 390
 - shortcut menu for, 708
 - stepping into, out of, over, 666
- Procedures in Files menu item, 679
- Procedures menu item
 - in Browse menu, 679
 - in ViewList submenu, 678
- Process running Data Explorer message, 201
- _PROCESS system variable, 315
- Process Viewer
 - opening, 674, 727
 - window, 390
- process-specific breakpoints (see task-specific breakpoints)
- processes
 - attaching to, 662, 725
 - controlling from Debugger, 122
 - halting, 665
 - killing current, 665
 - listing, 678
 - native, viewing, 390
 - sending signal to, 666
 - state of, 36
 - stopping, 122
- Processes menu item, 678
- processing profiling data, 383
- ProcRelativeLines configuration option, 24
- profdump command, 377, 382
- profile command, 357, 359
- profile descriptions
 - MULTI data visualization (.mdv) files, 744, 766
- Profile menu item, 674, 684
- Profile window, 361
 - (see also profiling reports)
 - menus, 372, 373
 - overview, 361
 - toolbar, 375
 - updates in, 363

- profilemode command, 373, 374, 375, 376, 377, 378, 381, 382, 383, 384
 - profilereport command, 363, 364, 372
 - profiling, 355
 - (see also profiling data)
 - AddressSpaces, 355, 359
 - caveats, 360
 - instrumented, 356
 - methods overview, 355
 - PC sampling, 355, 356, 603
 - range, 379
 - stand-alone program, 355, 359
 - tasks
 - all in system, 355, 357, 603
 - single, 355, 356, 359, 360
 - trace-enabled target, 355, 358, 475
 - profiling data, 354
 - (see also profiling)
 - (see also profiling reports)
 - adding to, 380
 - call count, 354, 355, 360, 475
 - call graph, 354, 355, 475
 - clearing, 384
 - collecting, 356, 357, 358, 359
 - converted from trace, 475
 - coverage analysis, 354, 355, 360, 475
 - disabling collection of, 360
 - dumping, 381
 - instrumenting code to generate, 356
 - overwriting, 380
 - PC (program counter) samples, 354, 355, 356, 360, 475
 - processing manually, 383
 - types of, 354
 - viewing, 361, 363, 377
 - profiling reports
 - block detailed, 370
 - call graph, 367
 - coverage, 369
 - overview, 363
 - source, 371
 - standard calls, 365
 - status, 365
 - Program already present on target. Verify option, 114
 - program counter (PC), 22
 - dimmed, 123
 - location in TimeMachine mode, 416, 419
 - samples, 354, 355, 356, 360, 475
 - Program Counter button, 694
 - Program Flash ROM option, 114
 - programs
 - execution of, 122
 - profiling stand-alone, 355, 359
 - RAM download, 111
 - ROM copy, 113
 - ROM run, 112
 - in Separate Session TimeMachine mode, 422
 - starting, 122
 - stepping through, 123
 - in TimeMachine mode, 17, 413
 - unknown, 113
 - Project Manager
 - opening, 685, 701
 - Project Manager menu item, 685
 - prologue code
 - debugging, 390
 - Py pane, 32
 - python pane, 32
 - Python scripts, TimeMachine API, 471, 472
- Q**
- question mark (?) wildcard character, 303
- R**
- r command line option, 725
 - R command line option, 726
 - RAM download programs, 111
 - range analysis, 379
 - rc command line option, 726
 - .rc files
 - customizing registers with, 285
 - ignoring with -norc, 7, 725
 - read-only system variables, 314
 - reading command scripts on startup, 726
 - real-time operating systems (RTOS)
 - debugging
 - INDRT (rtserv), 78
 - INDRT2 (rtserv2), 60
 - Rebuild menu item, 685
 - Reconstructed Registers window, 476
 - Record Command+Output menu item, 689
 - Record Commands menu item, 689
 - recording commands
 - menu items for, 689, 690
 - to playback files, 725, 726
 - recording output
 - to playback files, 726
 - redirecting input/output, 665
 - Refresh Views menu item, 675
 - register definition files
 - customizing, 285, 288, 289
 - default, 285, 288, 289

- GRD format, 276, 776
- location of, 286
- overloading, 288
- overview, 776
- searching for, 286
- Register Explorer, 254
 - (see also register definition files)
 - (see also Register Information window)
 - (see also Register View window)
 - overview, 254
 - startup, 286
- Register Information window
 - buttons, 275
 - opening, 271
 - overview, 270, 271
 - panes, 272, 273, 274
 - register values, changing, 275
 - shortcut menus, 272, 274
- Register Search window, 280
- Register Setup dialog, 279, 280
- Register Synonyms menu item, 678
- Register View window
 - configuration files, 269
 - copying register values, 267
 - customizing, 263, 264, 265, 266
 - editing register contents, 267
 - File menu, 256
 - Format menu, 258
 - opening, 254, 673
 - overview, 254, 255
 - printing window contents, 269
 - refreshing, 268
 - register tree, 260, 266, 267
 - shortcut menus, 262
 - tabs, 260, 264, 265, 266
 - toolbar, 259
 - View menu, 257
- registers
 - bitfield definitions, 276
 - changing values of, 275
 - commands, 282
 - copying values, 267
 - creating, 279, 280
 - customizing, 285
 - defining (see register definition files)
 - editing contents, 267
 - listing, 678
 - printing list of, 269
 - searching for, 280
 - testing access to, 96
 - viewing (see Register Information window) (see Register View window)
- Registers button, 695
- Registers menu item
 - in View menu, 673
 - in ViewList submenu, 678
- relationship view, Integrate, 802
- Reload button, 694
- _REMOTE_CONNECTED system variable, 315
- Remove All Breakpoints menu item, 667
- reports, profiling (see profiling reports)
- reprogramming
 - buttons, 700
- rerooting Tree Browser, 243
- Reset button, 694
- reset command, 95
- Restart button, 693, 716
- Restart menu item, 665
- restore command, 689
- Restore State menu item, 689
- restoring
 - deleted breakpoints, 137
- \$result special predefined variable, 317
- \$retadr() special operator, 320
- Retrieve Trace menu item, 683
- retrieving trace, 408
- Return on Selected Items button, 693, 717
- Return on Selected Items menu item, 666
- reusing windows, 14
- Rhapsody, integration with MULTI, 732, 733, 734, 735, 736
- right angle bracket (>) command
 - menu equivalent, 689, 690
- right angle bracket-double (>>) command
 - menu equivalent, 689, 690
- right-click
 - in command pane, 719
- RO command line option, 726
- ROM
 - attaching to a process running in, 552
 - configuring a project to run in, 551
 - creating executables for, 550
 - debugging a process running in, 554
 - differences in programs built to run from, 550
 - executing a program in, 552, 553
 - loading programs from ROM to RAM (see ROM-to-RAM programs)
 - working with when using MULTI, 550
- ROM copy programs, 113
- ROM run programs, 112
- ROM-to-RAM programs
 - configuring, 555

- creating, 555
- debugging, 557
- differences from RAM programs, 554
- executing, 556
- routines (see procedures)
- rtserv debug server
 - logging, 83
 - serial port access, exclusive, 82
 - TFTP directory, 81
- _RTSERV_VER system variable, 315
- rtserv2 debug server
 - connection examples, custom, 68, 69
 - logging, 64, 67, 68
- run command line option, 726
- run control, synchronous
 - on multi-core systems, 629
- Run on Detach menu item, 673
- Run System menu item, 666
- Run to Cursor menu item, 666
- run-control buttons, TimeMachine, 414, 416
- run-mode AddressSpaces, 581
- run-mode breakpoints
 - any-task, setting, 587
 - group, 588, 589
 - task-specific, 587
- run-mode connections
 - automatically established (see run-mode partner)
 - concurrent with freeze-mode connections, 69
 - troubleshooting, 73
 - multiple, 578
 - overview, 578
- run-mode debugging, 60, 78, 580
 - (see also Task Manager)
 - AddressSpaces, 581
 - attaching to tasks, 582
 - breakpoints, 587
 - overview, 578
 - with Separate Session TimeMachine, 422
 - with TimeMachine, 418
- run-mode partner
 - disabling, 73
 - overview, 69
 - setting, 70, 71
 - troubleshooting, 73
- run-mode tasks
 - attaching to, 582
 - in TimeMachine mode, 418
- run-time error checking
 - generating information for, 5
 - memory allocation, 340
- RUNNING status bar message, 36

Running target list status, 19

S

- save command, 689
- Save Configuration As menu item, 688
- Save Configuration as User Default menu item, 687
- Save State menu item, 689
- Save Trace Session menu item, 683
- sb command, 587, 619
- scripts
 - command, reading on startup, 726
 - syntax checking, 320
- scroll bars
 - diamond in, 156
- scrolling
 - infinite, 156
- Search for Objects window, 250
- Search in Files menu item, 686
- Search menu item, 686
- searching
 - beeping in incremental, 157
 - caches, 396
 - files, 159
 - incrementally, 156
 - MULTI windows, 156
 - shortcuts for, 157, 713
 - source pane, 167, 712
 - with variable search designators, 297, 298
 - with wildcards, 303
- security view, Integrate, 802
- selecting
 - items from lists, 161
 - text in Debugger windows, 160
 - text in main Debugger window, 161
- _SELECTION system variable, 315
- Send Signal menu item, 666
- Separate Session TimeMachine, 422
- Serial Connection Chooser, 646
- Serial Connection Settings dialog, 647, 649
- serial connections, 644
 - (see also MTerminal serial terminal emulator)
 - configuring, 646, 647, 649
 - creating, 646, 647, 649
 - overview, 644
 - starting, 644
- serial option
 - for rtserv2, 68
- serial port access, exclusive
 - INDRT (rtserv), 82
- Serial Terminal menu item, 685

- ul style="list-style-type: none;">
- serial terminal pane, 31
- serialconnect command, 686
- serialdisconnect command, 686
- ServerPollinterval option, 604
- servertimeout command line option, 726
- SERVERTIMEOUT system variable, 311
- Session, Separate TimeMachine, 422
- Set And Edit menu item, 706
- Set Any Task Breakpoint menu item, 707
- Set Breakpoint menu item, 706
- Set INTEGRITY Distribution menu item, 688
- Set Jump Breakpoint menu item, 707
- Set Program Arguments menu item, 665
- Set Run-Mode Partner dialog box, 70, 71
- Set Run-Mode Partner menu item, 71, 73, 680
- Set Triggers menu item, 683
- Set Triggers window
 - events, 494, 495
 - opening, 493
 - triggers, setting, 497
- Set u-velOSity Distribution menu item, 688
- Set Watchpoint menu item, 667
- set_runmode_partner command, 71
- setup scripts, board (see board setup scripts)
- _SETUP_SCRIPT system variable, 315
- _SETUP_SCRIPT_DIR system variable, 315
- shared libraries
 - disable debugging for with -noshared, 725
- shared object breakpoints, 136
 - managing, 136, 146, 148, 149, 150
 - managing deleted, 152
 - viewing, 136, 146, 148, 149, 150
 - viewing deleted, 152
- sharing breakpoints between TimeMachine and live target, 417
- shortcuts
 - command pane, 29, 718
 - Debugger window, main, 716
 - search, 157, 713
 - source pane, 717
- Show Command History menu item, 689
- signals
 - listing, 678
 - sending to current process, 666
- Signals menu item, 678
- simulators
 - connecting to, 41
- single-stepping
 - process, 123
 - task-specific, in freeze mode, 616
- sizeof() special operator, 320, 500
- slash, asterisk-double, slash (/* */) comments, 295
- slash-double (//) comments, 295
- socket command line option, 726
- software breakpoints
 - editing, 130
 - managing, 131, 146, 148, 149, 150
 - managing deleted, 152
 - moving, 132
 - overview, 128
 - properties of, 139
 - setting, 130
 - viewing, 131, 146, 148, 149, 150
 - viewing deleted, 152
- Solaris/Linux
 - file chooser dialog box, 713
 - printing, 663
- source addresses
 - expressing in Debugger, 293
- source code
 - interlaced with assembly code, 23
 - stepping through, 666
 - viewing in Debugger, 23
- source files
 - browsing, 165, 214, 230, 231, 232, 233
 - listing, 678
- source lines
 - shortcut menu for, 705
- Source Only menu item, 677
- source pane
 - browsing, 162
 - content displayed in, 16
 - display modes, 23
 - navigating, 162
 - overview, 13, 21
 - searching, 162, 167, 712
 - shortcut menus, 704
 - shortcuts, 704, 717
- Source Path menu item, 675
- source paths
 - listing, 678
- Source Paths menu item, 678
- source profiling report, 371
- special operators
 - \$bp_adr(), 318
 - \$entadr(), 318
 - \$exists(), 318
 - \$in(), 318
 - \$M_called_from(), 318
 - \$M_can_read_address(), 318
 - \$M_file_exists(), 318
 - \$M_get_temp_memory(), 318

- \$M_offsetof(), 319
 - \$M_sec_begin(), 319
 - \$M_sec_end(), 319
 - \$M_sec_exists(), 319
 - \$M_sec_size(), 319
 - \$M_strcaseprefix(), 319
 - \$M_strcmp(), 319
 - \$M_strcpy(), 319
 - \$M_strprefix(), 319
 - \$M_strstr(), 319
 - \$M_sym_exists(), 318
 - \$M_typeof(), 320
 - \$M_var_is_valid(), 320
 - offsetof(), 319
 - \$retadr(), 320
 - sizeof(), 320, 500
 - special variables
 - listing, 678
 - \$result, 317
 - specification files
 - specifying with -m, 724
 - using, 727
 - splash screen
 - disabling with -nosplash, 725
 - Srch status bar message, 36, 156
 - Srl pane, 31
 - Stabs debugging information
 - converting, 730
 - stand-alone program
 - profiling, 355, 359
 - stand-alone windows, 168
 - standard calls profiling report, 365
 - Standard Connection Methods, 42, 43, 44
 - Start collecting trace data button, 701
 - starting
 - Debugger, 5
 - in graphical (GUI) mode, 6, 7, 723
 - in non-GUI mode, 8, 724
 - MULTI for Object Structure Awareness (OSA), 725
 - MULTI from command line, 6, 722
 - State menu item, 688
 - state of process, 36
 - _STATE system variable, 316
 - Static Calls menu item, 679
 - static variables
 - listing, 678
 - Statics menu item, 678
 - status bar
 - in Debugger window, 14
 - status bar messages, 36
 - status profiling report, 365
 - Status target list column, 19
 - status(TimeMachine) target list status, 19
 - Step (into Functions) on Selected Items button, 692, 717
 - Step (into Functions) on Selected Items menu item, 666
 - Step Back button, 692
 - Step Up button, 692
 - Stop on Task Creation menu item, 672
 - Stop Record Commands+Output menu item, 690
 - Stop Recording Commands menu item, 690
 - stop sign in source pane, 129
 - STOPPED arrow (red) in Debugger, 22
 - dimmed, 123
 - STOPPED INSIDE status bar message, 37
 - STOPPED status bar message, 36
 - Stopped target list status, 19
 - strings
 - searching incrementally for, 156
 - structures in Data Explorer, 190
 - subroutines
 - stepping, 666
 - synchronous run control, multi-core systems, 629
 - syntax checking
 - commands, 320
 - scripts, 320
 - SysHalt target list status, 19
 - system variables
 - in Debugger, 310, 312
 - listing, 678
 - read-only, 314
- ## T
- target command, 95
 - target list
 - auto-sizing of, 15
 - compressed display, 17
 - CPU % column, 20, 603
 - grouping of items in, 17
 - hiding, 15
 - hierarchical display, 17
 - identifying items in, 16
 - multiple cores in, 620
 - opening multiple Debugger windows from, 14
 - reusing Debugger window with, 14
 - shortcut menu, 702
 - showing, 15
 - Status column, 19
 - terminology, 18
 - Target menu, 680
 - target pane, 30
 - Target Settings menu item, 668

- `_TARGET` system variable, 316
- `_TARGET_COPROCESSOR` system variable, 316
- `_TARGET_FAMILY` system variable, 316
- `_TARGET_IS_BIGENDIAN` system variable, 316
- `_TARGET_MINOR_OS` system variable, 316
- `_TARGET_OS` system variable, 316
- `_TARGET_SERIES` system variable, 316
- targets
 - concurrent freeze-/run-mode connections to, 69
 - troubleshooting, 73
 - configuring, 90
 - (see also board setup scripts)
 - connecting to, 40
 - disconnecting from, 58
 - installing hardware, 90
 - multi-core (see multi-core systems)
 - preparing (see preparing a target)
 - simulated, 41
 - testing initialization of, 96, 97
 - trace-enabled, profiling, 355, 358, 475
- Task Debugger mode, 615
- task groups, 584
 - (see also Task Manager)
 - (see also tasks)
 - breakpoints for, 588, 589
 - configuration file for, 600
 - creating, 584
 - default, 585
 - synchronous operations, relating to, 589
- Task Manager, 580
 - (see also task groups)
 - (see also tasks)
 - attaching to tasks, 582
 - debugging tasks, 583
 - freezing task list, 584
 - halting tasks, 582, 583
 - information pane, 597
 - killing tasks, 582
 - menus, 592, 593, 594
 - mouse operations, 598
 - multiple cores in, 621
 - opening, 580, 674
 - overview, 580, 592
 - running tasks, 582
 - shortcut menus, 598
 - task groups in, 584, 585
 - task list pane, 596
 - toolbar, 595
- `_TASK_EXIT_CODE` system variable, 316
- task-specific breakpoints, 587
 - freeze-mode, 420, 618
 - task-specific single-stepping, freeze-mode, 616
- tasks, 578
 - (see also task groups)
 - (see also Task Manager)
 - breakpoints for, 587, 618
 - displaying in the Debugger source pane, 583
 - freeze mode, debugging, 613, 614, 615
 - halting, 582
 - halting on attach, 583
 - killing, 582
 - profiling
 - all, 355, 357, 603
 - single, 355, 356, 359, 360
 - run-mode, attaching to, 582
 - running, 582
 - in Separate Session TimeMachine mode, 422
 - in target list, 614
 - terminology conventions, 578, 607
 - in TimeMachine mode, 17, 418, 419
 - in Trace List, 436
- Tasks menu item, 674
- TASKWIND system variable, 311
- Temporary Connection Methods, 49
- terminal server connections, 68
- tests, memory (see memory testing)
- text command line option, 726
- text offsets, position-independent code (PIC)
 - specifying with -text, 726
 - specifying with `_TEXT`, 313
 - specifying with Text Offset, 112
- `_TEXT` system variable, 313
- text, selecting, cutting, and pasting, 160
- Tfc pane, 33
- TFTP directory
 - INDRT (rtserv), 81
- third-party tools
 - compilers, 730
 - Rhapsody, 732
 - running from Debugger, 731
- Thread ID,
 - (see also ThreadX)
- threads, 607
- ThreadX
 - freeze-mode debugging, 606
 - Thread ID,
 - `_tx_thread_created_ptr` OS-specific symbol, 607
- thumbnail pane
 - in PathAnalyzer, 427
 - in Trace List, 439
- time analysis, trace, 439
- TimeMachine API

- C/C++ example, 473
- file interface, 467, 469, 470, 472
- live interface, 467, 471
- overview, 466
- Python examples, 471, 472
- timemachine command, 414, 415, 418, 422
- TimeMachine Debugger, 413
 - (see also TimeMachine tools)
 - (see also trace)
 - commands, 416
 - disabling, 415
 - enabling, 414
 - limitations, 413
 - location of program counter in, 416, 419
 - OS trace in, 418, 419
 - overview, 413
 - run-control buttons, 414, 416
 - Separate Session, 422
- TimeMachine Debugger button, 696
- TimeMachine Debugger menu item, 684
- TimeMachine menu, 683
- TimeMachine mode
 - program or task in, 17, 413
- TimeMachine tools, 404
 - (see also MULTI EventAnalyzer)
 - (see also PathAnalyzer)
 - (see also Profile window)
 - (see also TimeMachine API)
 - (see also TimeMachine Debugger)
 - (see also Trace Browsers)
 - (see also Trace List)
 - (see also Trace Statistics window)
 - accessing from Trace List, 444
 - overview, 404
- timeout
 - setting for debug servers, 726
- Toggle All Breakpoints menu item, 667
- Toggle IO Buffering menu item, 681
- toggling
 - breakpoint status, 127
 - tracepoint status, 127
- toolbar
 - adding buttons to, 698
 - button descriptions, 691
 - removing buttons from, 698
- Tools menu, 685
- top command line option, 727
- Top Project
 - contents, 90
- _TOP_PROJECT system variable, 317
- _TOP_PROJECT_DIR system variable, 317
- tpset command, 562, 568, 570, 571
- trace
 - accessing with TimeMachine API, 466
 - analyzing
 - with TimeMachine, 404
 - with TimeMachine API, 466
 - bookmarking, 444, 445, 446, 448
 - browsing, 431, 451, 452, 454, 456, 457
 - clearing, 410
 - collecting, 405
 - configuring collection of, 489, 491, 492, 493, 497
 - converting to
 - EventAnalyzer information, 474
 - profiling data, 475
 - disabling collection of, 406
 - discarding, 410
 - events supported, by architecture, 494
 - events, defining, 504
 - events, modifying with Set Triggers window, 495
 - events, specifying with
 - Advanced Event Editor, 443, 497
 - count expressions, 505
 - state machine expressions, 506
 - state machine resources, 502
 - events, viewing in the EventAnalyzer, 474
 - filtering, 442, 443
 - loading, 448
 - managing, 405
 - navigating, 428, 436, 440, 441, 442
 - operating system
 - collecting, 406
 - navigating, 436
 - options, setting, 480
 - overview, 402
 - quick start, 403
 - reconstructed register values, viewing, 476
 - retrieving, 408
 - saving, 448
 - searching, 430, 446, 447
 - statistics, viewing, 460, 462, 463, 464, 465
 - targets
 - configuring, 488
 - profiling, 355, 358, 475
 - supported, 402
 - viewing information about, 489
 - time analysis, 439
 - triggers, 489, 491, 493, 497
 - viewing with Trace List, 434, 435, 436
 - (see also Trace List)
- trace abort command, 408
- Trace Branch Browser, 454

- Trace Browsers, 451
 - Trace Call Browser, 431, 457
 - trace clear command, 411
 - trace disable command, 406
 - trace enable command, 406
 - Trace Function Interval menu item, 492
 - Trace Instruction Browser, 456
 - Trace List, 434
 - (see also trace)
 - AddressSpaces in, 436
 - columns, 436
 - display formats, 435
 - filtering trace with, 442
 - instruction pane, 436
 - navigating with, 436, 440, 441, 442
 - OS trace data in, 436
 - overview, 434
 - tasks in, 436
 - thumbnail pane, 439
 - time analysis, 439
 - TimeMachine functions, accessing, 444
 - tree pane, 435
 - Trace List menu item, 684
 - Trace Memory Browser, 452
 - Trace Options menu item, 684
 - Trace Options window, 480
 - trace retrieve command, 408
 - trace session
 - saving and loading, 448
 - Trace Statistics menu item, 684
 - Trace Statistics window
 - AddressSpaces tab, 462
 - Branches tab, 464
 - Functions tab, 465
 - Memory tab, 463
 - opening, 460
 - overview, 460
 - Summary tab, 460
 - traceload command, 450
 - tracemevsys command, 475
 - tracepoints
 - buffer
 - overview, 560
 - purging, 567
 - viewing, 566
 - deleting, 564
 - disabling, 127, 565
 - editing, 562, 563
 - enabling, 127, 565
 - example, 567
 - limitations, 570
 - listing, 564
 - markers, 125, 561
 - overview, 124, 560, 561
 - properties, 562
 - resetting, 565
 - setting, 561
 - timeout feature, 563
 - toggling status of, 127
 - viewing information about, 128
 - tracepro command, 358
 - tracesave command, 449
 - traffic pane, 33
 - translating
 - debugging information, 730
 - Tree Browser, 239
 - calls, browsing, 244, 245, 246
 - classes, browsing, 243
 - node operations, 242
 - opening, 239
 - rerooting, 243
 - tree pane in Trace List, 435
 - Trg pane, 30
 - triggers, trace, 489, 491, 493, 497
 - troubleshooting
 - concurrent freeze-/run-mode connections, 73
 - flash memory, 547
 - tv command line option, 727
 - _tx_thread_created_ptr OS-specific symbol, 607
 - Type menu item, 679
 - types
 - in Data Explorer, 195
 - data, browsing, 213, 214, 233, 234, 235
 - shortcut menu for, 710, 711
 - typographical conventions, xxvi
- ## U
- u-velOSity (see operating systems)
 - Unconnected Executables target list message, 18
 - unknown programs, 113
 - Unload Module menu item, 680
 - Unreadable memory Data Explorer message, 201
 - Unreadable/Unknown Data Explorer message, 201
 - update command, 189
 - Upstack button, 694, 717
 - UpStack menu item, 162, 676
 - UpStack To Source menu item, 163, 677
 - usage command line option, 727
 - Use Connection menu item, 667
 - Use Which Connection/CPU? dialog box, 105
 - using a connection, 105

Utility Program Launcher, 685

V

-V command line option, 727

variable lifetime information, 299

variables

 global, listing, 678

 local, listing, 673, 678

 in procedure, listing, 678

 \$result, 317

 search designators, 297, 298

 shortcut menu for, 709

 static, listing, 678

 system, 310, 312

 viewing value of, 300

 viewing values of, 297

Variables In Procedure menu item, 678

velOSity (see operating systems)

VERIFYHALT system variable, 311

verifying presence of executable, 114

view command, 184

view descriptions

 MULTI data visualization (.mdv) files, 744, 768

View Expression menu item, 675

View menu, 673

viewdel command, 185

viewing

 assembly code, 23

 caches, 392, 396

 Debugger version information, 727

 information about MULTI, 691

 interlaced assembly code, 23

 line numbers, 21, 24

 local variables, 673

 memory, 324

 (see also Memory View window)

 native processes, 390

 profiling data, 361, 363, 377

 program at addresses, 163, 677

 source code, 23

 variables, 297, 300

viewlist command, 192

views, Integrate, 802

visualizations

 custom (see MULTI data visualization)

W

wait command, 95

wildcards, 303

windows

 closing multiple, 676

 in MULTI, 156

 opening multiple, 14

 reusing, 14

 stand-alone, 168

Windows menu, 690

Write to file menu item, 662

Z

ZOMBIE status bar message, 37

Zombied target list status, 19