ARM® DSTREAM™ and RVI™

Version 4.4

Using the Debug Hardware Configuration Utilities

ARM®

ARM DUI 0498E (ID091611)

ARM DSTREAM and RVI
Using the Debug Hardware Configuration Utilities

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
May 2010	A	Non-Confidential	First release.
November 2010	B	Non-Confidential	Second Release
30 April 2011	C	Non-Confidential	DSTREAM and RVI v4.2.1 Release
29 July 2011	D	Non-Confidential	Update 1 for DSTREAM and RVI v4.2.1 Release
30 September 2011	E	Non-Confidential	DSTREAM and RVI v4.4 Release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Conformance Notices

This section contains conformance notices.

Federal Communications Commission Notice

This device is test equipment and consequently is exempt from part 15 of the FCC Rules under section 15.103 (c).

Class A

Important: This is a Class A device. In residential areas, this device may cause radio interference. The user should take the necessary precautions, if appropriate.

CE Declaration of Conformity

The system should be powered down when not in use.

It is recommended that ESD precautions be taken when handling DSTREAM, RVI, and RVT equipment.

The DSTREAM, RVI, and RVT modules generate, use, and can radiate radio frequency energy and may cause harmful interference to radio communications. There is no guarantee that interference will not occur in a particular installation. If this equipment causes harmful interference to radio or television reception, which can be determined by turning the equipment off or on, you are encouraged to try to correct the interference by one or more of the following measures:

- ensure attached cables do not lie across the target board
- reorient the receiving antenna
- increase the distance between the equipment and the receiver
- connect the equipment into an outlet on a circuit different from that to which the receiver is connected
- consult the dealer or an experienced radio/TV technician for help

Note

It is recommended that wherever possible shielded interface cables be used.

Contents
ARM DSTREAM and RVI Using the Debug Hardware Configuration Utilities

Chapter 1
Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM® processor signal names.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0498E
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center, http://infocenter.arm.com/help/index.jsp
- ARM Technical Support Knowledge Articles, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html
- ARM Support and Maintenance, http://www.arm.com/support/services/support-maintenance.php
- ARM Glossary, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Chapter 2
Getting started with the debug hardware configuration utilities

The following topics describe how to get started with using the debug hardware configuration utilities:

2.1 About the debug hardware configuration utilities

The debug hardware configuration utilities enable you to connect to the debug hardware unit that provides the interface between your development platform and your PC. The following utilities are provided:

RVI Config IP utility

Used to configure the IP address on a debug hardware unit. This enables you to access the unit over Ethernet.

Debug Hardware Config utility

Used to configure a debug hardware unit. This enables you to:

- Identify the target devices on your development platform. These devices can be one or more processors, and optional trace devices or CoreSight™ devices.
- Configure debug hardware and target-related features that are appropriate to correctly debug your development platform.
- Save the configuration to a device configuration file. The device configuration file is used by your debugger to connect to each target processor on your development platform.

RVI Update utility

Used to update the firmware and devices on a debug hardware unit and probe.

Note

This document applies to the ARM® DSTREAM™ debug and trace unit and the ARM RVI™ debug unit. The term trace hardware refers to:

- the built-in trace hardware of a DSTREAM unit
- an ARM RVT or ARM RVT2™ trace capture unit for RVI.

Differences in debug and trace hardware features between the units are explicitly stated.

2.1.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Scanning for available debug hardware units* on page 2-4
- *Connecting to a debug hardware unit* on page 2-7
- Chapter 3 *Configuring network settings for your debug hardware unit*
- Chapter 4 *Managing the firmware on your debug hardware unit*
- Chapter 5 *Creating debug hardware target configurations*.

2.2 Starting the debug hardware configuration utilities

How you start the required debug hardware configuration utility depends on:

- your PC operating system
- whether you installed the host software with an ARM® software development product
- whether you are running a debugger session.

Note

For information about debugging with your own debugger, see your debugger documentation.

2.2.1 Starting the Debug Hardware Config utility on Windows

To start the required debug hardware configuration utility on Windows platforms:

1. Select **Start → All Programs → ARM DS-5 → Debug Hardware**
2. Select the option for the utility you want to use:
   - select **Debug Hardware Config IP** to start the RVI Config IP utility
   - select **Debug Hardware Configuration** to start the Debug Hardware Config utility
   - select **Debug Hardware Update** to start the RVI Update utility.

   Note

   Be aware that an option for the Debug Hardware Config utility might not be available on the **Start** menu for some ARM products.

For more information, see the documentation for your ARM software development product.

2.2.2 Starting the Debug Hardware Config utility on Red Hat Linux

To start the required debug hardware configuration utility on Red Hat Linux platforms, select the appropriate shortcut. The shortcut depends on the version of Red Hat Linux and the desktop environment that you are using.

If no desktop shortcut is available, at the command-line:

1. Make sure the paths to the utilities are configured. See the *Getting Started* document provided with your ARM software development product for more details.
2. Enter the command for the utility you want to use:
   - enter **rviconfigip** to start the RVI Config IP utility
   - enter **rvconfig** to start the Debug Hardware Config utility
   - enter **rviupdate** to start the RVI Update utility.

2.2.3 Accessing the Debug Hardware Config utility from your Debugger

You can access the Debug Hardware Config utility directly from your debugger. See your debugger documentation for details on how to do this.

2.2.4 See also

Tasks

- *Scanning for available debug hardware units* on page 2-4
- *Connecting to a debug hardware unit* on page 2-7.

2.3 Scanning for available debug hardware units

To scan for available debug hardware units:

1. Start the required configuration utility, for example Debug Hardware Config.

2. Click the **Scan** button to scan for debug hardware units that are connected to your local network or to a USB port on your PC. The **Scan** button becomes animated to indicate that a scan is in progress. When the configuration utility finds a unit, it adds it to the list of available units. The following figure shows an example:

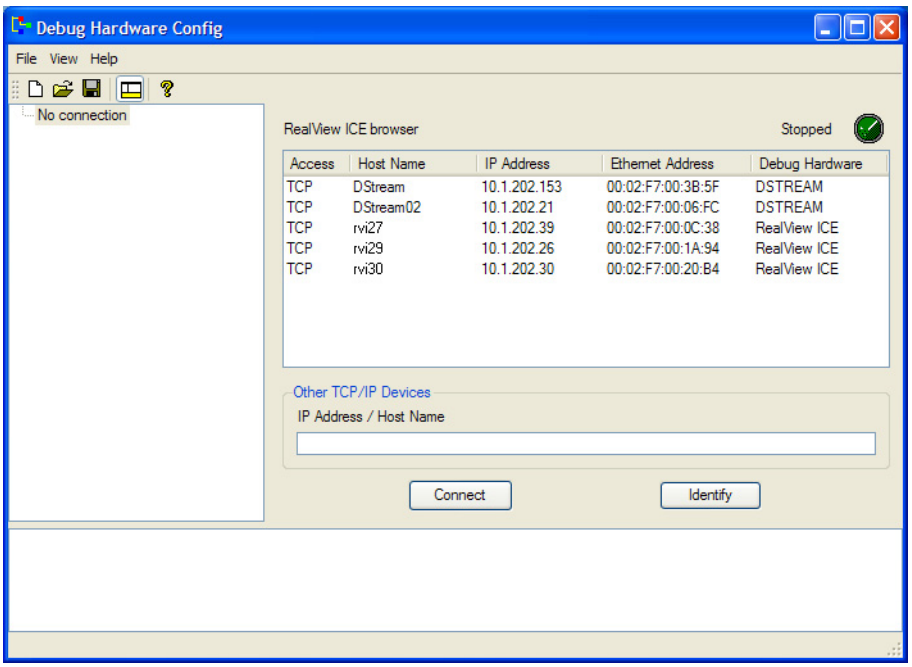


Figure 2-1 Debug Hardware Config utility

Note

Any unit shown in light gray is one that has responded to browse requests but does not have a valid IP address. You cannot connect to that unit by TCP/IP until you have configured it for use on your network.

The scan tool searches for debug hardware units that are connected to your local network or USB ports on your PC. The units found are listed in the browser on the right of the window.

Note

Units that are connected to different networks do not appear in the configuration utility. Consequently, if you want to connect to a debug hardware unit on a separate network, you must know the IP address of that unit.

If you want to stop scanning, click the **Scan** button. You can click the **Scan** button again at any time to force a rescan for available debug hardware units and update the list.

3. Select **Exit** from the **File**. This disconnects from any connected unit, and exits the configuration utility.

2.3.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Identifying a debug hardware unit* on page 2-6
- *Connecting to a debug hardware unit* on page 2-7
- Chapter 3 *Configuring network settings for your debug hardware unit*.

2.4 Identifying a debug hardware unit

If you have multiple debug hardware units on a network, you can identify the unit you want to access from the configuration utility.

To identify the debug hardware unit you want to access:

1. Select a unit from the list, or enter an IP address of a unit if it is on a different network.
2. Click **Identify**.

   The identification indicators on the selected debug hardware unit flash for 5 seconds. If you have selected the wrong unit, select another unit from the list and repeat this step.

   Note

   On RVI, all LEDs on the front panel flash during identification.

   On DSTREAM, the DSTREAM logo flashes during identification.

2.4.1 See also

Tasks

- *Scanning for available debug hardware units* on page 2-4
- *Connecting to a debug hardware unit* on page 2-7
- Chapter 3 *Configuring network settings for your debug hardware unit*.

Reference

ARM® DSTREAM™ Setting Up the Hardware:

- The DSTREAM debug and trace unit, `../com.arm.doc.dui0481e/CHDCJEFH.html`.

ARM® RVT™ and RVT™ Setting Up the Hardware:

- The RVI debug unit, `../com.arm.doc.dui0481e/CHDCJEFH.html`.

2.5 Connecting to a debug hardware unit

You must connect to your debug hardware unit to create a configuration file that contains details of your target hardware and the debug hardware. The configuration file is required by your debugger to connect to your target development platform and debug your software.

2.5.1 Prerequisites

Before you can connect to a debug hardware unit, make sure you have:

1. Set up, or have access to, the debug hardware unit that interfaces with your development platform.
2. Installed the correct version of the host software on your PC for your debug hardware unit.

2.5.2 Procedure

To connect to a debug hardware unit:

1. Start the required configuration utility, for example the Debug Hardware Config utility.
2. If you do not see any debug hardware units listed for your local network, click the **Scan** button. The **Scan** button becomes animated to indicate that a scan is in progress. When the utility finds a debug hardware unit on your local network, it is added to the list of available units. The following figure shows an example:

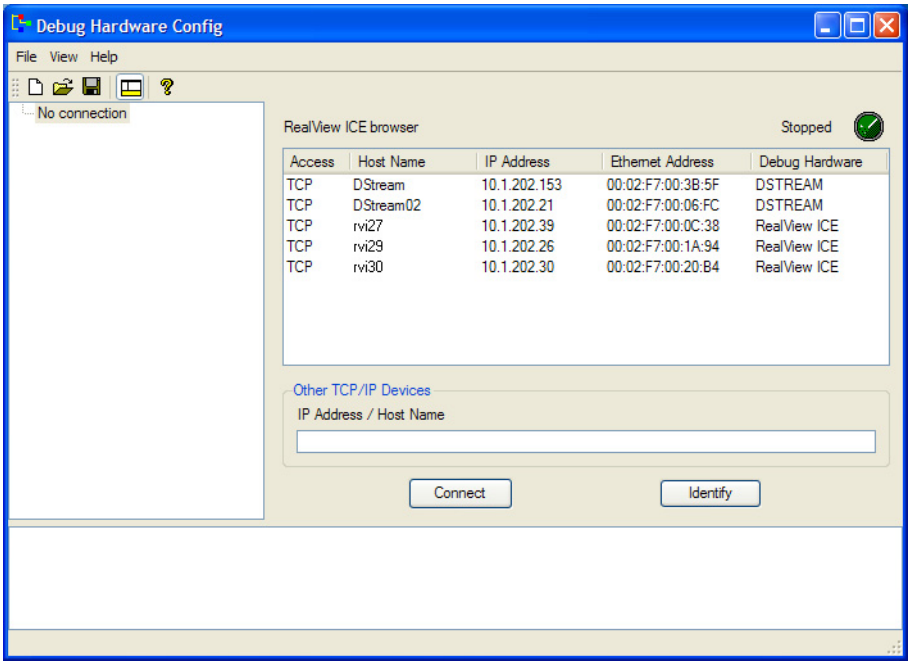


Figure 2-2 Debug Hardware Config utility showing debug hardware units

Note

The scan tool only searches for debug hardware units that are connected to your local network or USB ports on your PC. Therefore, units that are connected to a different network do not appear in the configuration utility. Consequently, if you want to connect to a debug hardware unit that is not accessible on your local network, ensure that you know the IP address of that debug hardware unit.

Any unit shown in light gray is one that has responded to browse requests but does not have a valid IP address. You cannot connect to that unit by TCP/IP until you have configured it for use on your network.

Alternatively, connect the debug hardware unit directly to your PC using a USB cable.

3. If multiple units are listed, and you are unsure about the debug hardware unit you want to use:
   a. Select a unit in the list, or enter an IP address of a unit on a different network.
   b. Click **Identify**. The identification indicators on the unit flash for five seconds.

   Note

   On RVI, all LEDs on the front panel flash during identification.
   On DSTREAM, the DSTREAM logo flashes during identification.

4. To connect to your required unit, select the unit and click **Connect**. Alternatively, do one of the following:
   - Double-click on the unit you want to connect to.
   - In the IP Address/Host Name field, enter either the IP address or host name of the device you want to connect to and click **Connect**.

   When a connection has been established, the configuration utility display changes to show the configuration features provided by that utility.

   If you have problems connecting to a debug hardware unit, you must troubleshoot the debug hardware connections.

5. Select **Exit** from the **File** menu. This disconnects from any connected unit, and exits the configuration utility.

2.5.3 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Scanning for available debug hardware units* on page 2-4
- *Scanning for available debug hardware units* on page 2-4
- Chapter 3 *Configuring network settings for your debug hardware unit*
- Chapter 10 *Troubleshooting your debug hardware unit*.

Reference

ARM® DSTREAM™ Setting Up the Hardware:

- The DSTREAM debug and trace unit, `../com.arm.doc.dui0481e/CHDCJEFH.html`.

ARM® RVI™ and RVT™ Setting Up the Hardware:

- The RVI debug unit, `../com.arm.doc.dui0481e/CHDCJEFH.html`.

Chapter 3
Configuring network settings for your debug hardware unit

The following topics describe how to configure the network settings for your debug hardware unit:

3.1 About configuring network settings

The configuration process depends on the way in which the debug hardware unit is connected to the host computer, and whether or not your network uses *Dynamic Host Configuration Protocol* (DHCP).

If you have connected your debug hardware unit to an Ethernet network or directly to the host computer using an Ethernet cross-over cable, you must configure the network settings before you can use the unit for debugging. You have only to configure the network settings once.

The following connections are possible:

- Your debug hardware unit is connected to your local network that uses DHCP. In this situation, you do not have to know the Ethernet address of the unit, but you must enable DHCP.
- Your debug hardware unit is connected to your local network that does not use DHCP. In this situation, you must assign a static IP address to the debug hardware unit.

Note

If you have connected your debug hardware unit directly to the host computer using a USB cable, and you do not intend to connect it to a network, you do not have to configure the network settings.

3.1.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Determining the correct network settings* on page 3-3
- *Configuring the network settings for a debug hardware unit* on page 3-8
- Chapter 10 *Troubleshooting your debug hardware unit*.

ARM® DSTREAM™ Setting Up the Hardware:

- *Connecting the DSTREAM debug and trace unit*, `../com.arm.doc.dui0481e/I1004916.html`

ARM® RVT™ and RVT™ Setting Up the Hardware:

- *Connecting the RVI debug unit*, `../com.arm.doc.dui0515e/I1004916.html`

Reference

- *Debug hardware unit network settings* on page 3-7.

3.2 Determining the correct network settings

Before you can configure the network settings, you must first determine the correct network settings for your debug hardware unit. To do this, you must consult with the system administrator for your network.

The information that you require depends on whether your network uses *Dynamic Host Configuration Protocol* (DHCP):

Table 3-1 Required debug hardware network settings

Information	Using DHCP	Not using DHCP
Host Name	Yes	Yes
IP Address	-	Yes
Default Gateway	-	Yes
Subnet Mask	-	Yes
Ethernet Address	Yes	Yes
Ethernet Type	Yes	Yes

3.2.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3.

Reference

- *The Configure debug_hardware device dialog box* on page 3-4
- *The Configure new debug_hardware device dialog box* on page 3-6.

3.3 The Configure *debug_hardware* device dialog box

The Configure *debug_hardware* device dialog box enables you to modify the network settings on a debug hardware unit that has previously been configured. The following figure shows an example:

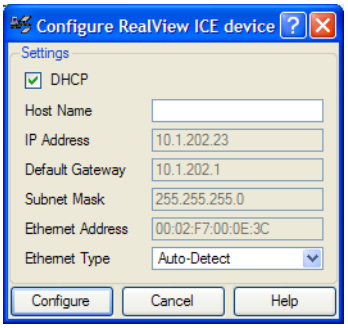


Figure 3-1 The Configure *debug_hardware* device dialog box

Note

You can modify the settings only for a debug hardware unit that is on your local network or that is connected to a USB port on your PC.

The network settings available depend on whether or not your network uses *Dynamic Host Configuration Protocol* (DHCP):

- If your network uses DHCP, you must know:
  - the hostname that you want to use for your unit (if any)
  - the Ethernet type of your network.
- If your network does not use DHCP, you must know:
  - the hostname that you want to use for your unit (if any)
  - the IP address that you want to use for your unit
  - the default gateway for your network (if it has one)
  - the subnet mask for your network.
  - the Ethernet type of your network.

Note

The Ethernet Address field is read-only.

After setting up the network settings, click **Configure** to write the values to the unit.

Click **Exit** to close the Configure *debug_hardware* device dialog box

3.3.1 See also

Tasks

- *About configuring network settings* on page 3-2
- *Determining the correct network settings* on page 3-3
- *Configuring the network settings for a debug hardware unit* on page 3-8
- *Modifying the network settings for a debug hardware unit* on page 3-10
- *Troubleshooting* on page 3-13.

Reference

- *The Configure new debug_hardware device dialog box* on page 3-6
- *Debug hardware unit network settings* on page 3-7.

3.4 The Configure new *debug_hardware* device dialog box

The Configure new *debug_hardware* device dialog box enables you to:

- configure the network settings for a debug hardware unit that has not been previously configured
- configure the network settings for a debug hardware unit that is on a different subnet.

The following figure shows an example:

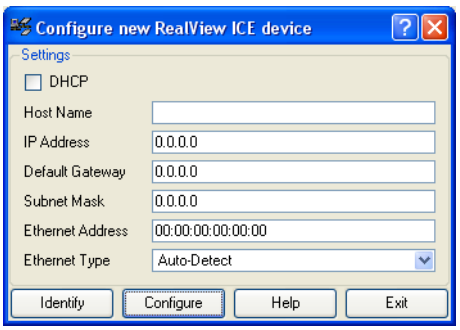


Figure 3-2 The Configure new *debug_hardware* device dialog box

The network settings available depend on whether or not your network uses *Dynamic Host Configuration Protocol* (DHCP):

- If your network uses DHCP, you must know:
  - the hostname that you want to use for your unit (if any)
  - the Ethernet address of unit
  - the Ethernet type of your network.
- If your network does not use DHCP, you must know:
  - the hostname that you want to use for your unit (if any)
  - the IP address that you want to use for your unit
  - the default gateway for your network (if it has one)
  - the subnet mask for your network.
  - the Ethernet address of unit
  - the Ethernet type of your network.

If more than one unit is listed in the browser, click **Identify** to identify your unit. The Identification LEDs on the selected unit flash for five seconds.

After setting up the network settings, click **Configure** to write the values to the unit.

Click **Exit** to close the Configure new *debug_hardware* device dialog box

3.4.1 See also

Tasks

- *About configuring network settings* on page 3-2
- *Determining the correct network settings* on page 3-3
- *Configuring the network settings for a debug hardware unit* on page 3-8.

Reference

- *Debug hardware unit network settings* on page 3-7.

3.5 Debug hardware unit network settings

The following network settings are available for a debug hardware unit:

DHCP Enables or disables *Dynamic Host Configuration Protocol* (DHCP):

- If your network uses DHCP, you must know the hostname that you want to use for your debug hardware unit (if any).

  Note

  You do not have to know the IP address for your debug hardware unit, or the default gateway and subnet mask for your network, because these settings are fetched from a DHCP server on your network.

- If your network does not use DHCP, you must know:
  - the hostname to use for your debug hardware unit (if any)
  - the IP address to use for your debug hardware unit
  - the default gateway for your network (if it has one)
  - the subnet mask for your network.

Host Name The host name for the unit. This must contain only the alphanumeric characters (A to Z, a to z, and 0 to 9) and the - character, and must be no more than 39 characters long.

IP Address The static IP address to use when DHCP is disabled.

Default Gateway

The default gateway to use when DHCP is disabled.

Subnet Mask

The subnet mask to use when DHCP is disabled.

Ethernet Address

The Ethernet address of the unit.

Ethernet Type

The type of network you are using:

- If you know the type of network, select the type. The options are:
  - **10-MBit, Half Duplex**
  - **10-MBit, Full Duplex**
  - **100-MBit, Half Duplex**
  - **100-MBit, Full Duplex**.
- Otherwise, select **Auto-Detect**.

3.5.1 See also

Tasks

- *About configuring network settings* on page 3-2
- *Configuring the network settings for a debug hardware unit* on page 3-8
- *Troubleshooting* on page 3-13.

Reference

- *The Configure debug_hardware device dialog box* on page 3-4
- *The Configure new debug_hardware device dialog box* on page 3-6.

3.6 Configuring the network settings for a debug hardware unit

If you have a debug hardware unit that does not have a valid IP address, or is on a different network, you must manually enter the Ethernet address during configuration.

3.6.1 Prerequisites

Before you can configure the network settings, you must first determine the correct network settings for your debug hardware unit:

- If you do not want to use DHCP, then you must obtain an IP Address, Default Gateway, and Subnet Mask from your network administrator.
- If you want to use DHCP, you must inform your network administrator of the Ethernet Address of the unit, so that it can be added to the DHCP server.

3.6.2 Procedure

To configure your new debug hardware unit:

1. Open the RVI Config IP utility.
2. If the debug hardware unit is on your local network or connected to a USB port on your PC, continue at step 3.

   Otherwise, continue at step 6.

3. Click the **Scan** tool to scan for debug hardware units.

   Note

   Only debug hardware units that are on your local network or connected to a USB port on your PC are listed.

4. Select the debug hardware unit you want to configure.

5. Click the **Identify** tool to verify that the identification LEDs flash on the correct debug hardware unit.

6. Click the **Config New** tool. The Configure new *debug_hardware* device dialog box appears, as shown in the following figure:

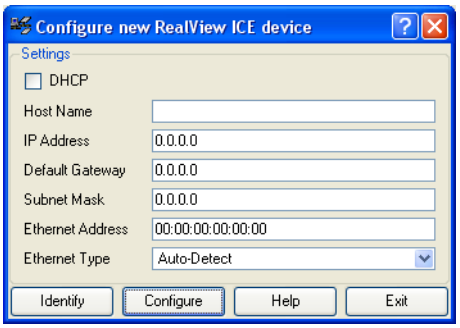


Figure 3-3 The Configure new *debug_hardware* device dialog box

7. Determine the Ethernet address of your debug hardware unit by reading the label on the side of the unit, and enter it into the Ethernet Address field.
8. If you are not using DHCP:
   a. Deselect **DHCP**.

b. Enter the required details in the following fields:
   - IP Address
   - Default Gateway
   - Subnet Mask.

c. Continue at step 9.

9. If you are using DHCP, select **DHCP**.

10. Enter the hostname in the Host Name field. This must contain only the alphanumeric characters (`A-Z`, `a-z`, and `0-9`) and the - character, and must be no more than 255 characters long.

11. Select the required Ethernet Type:
    - if you know the type of network that you are using, select that type
    - otherwise, select **Auto-Detect**.

12. Click **Configure**.

    The debug hardware unit restarts. During the restart the unit is removed from the list of units. When the restart is complete, the unit re-appears in the list of units, with the new network settings.

    Note

    If the debug hardware unit is using DHCP, the list of units might display its **IP Address** as `127.0.0.2`. This is a dummy address that the debug hardware unit uses when it fails to obtain an IP address from the DHCP server.

    The list of units shows the correct address if the DHCP server has assigned it.

3.6.3 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Scanning for available debug hardware units* on page 2-4
- *Determining the correct network settings* on page 3-3
- *Troubleshooting* on page 3-13
- Chapter 10 *Troubleshooting your debug hardware unit*.

Concepts

- *About configuring network settings* on page 3-2.

3.7 Modifying the network settings for a debug hardware unit

You can modify the network settings of a debug hardware unit only if that unit is on your local network or connected to a USB port on your PC.

Note

If the debug hardware unit is on a different network, you must use the Configure New *debug_hardware* device dialog box.

3.7.1 Prerequisites

Before you can configure the network settings, you must first determine the correct network settings for your debug hardware unit:

- If you do not want to use DHCP, then you must obtain an IP Address, Default Gateway, and Subnet Mask from your network administrator.
- If you want to use DHCP, you must inform your network administrator of the Ethernet Address of the unit, so that it can be added to the DHCP server.

3.7.2 Procedure

To configure your debug hardware unit by manually entering an Ethernet address:

1. Open the RVI Config IP utility.
2.  Click the **Scan** tool to scan for debug hardware units.
3. Select the debug hardware unit you want to modify.
4.  Click the **Identify** tool to verify that the identification LEDs flash on the correct debug hardware unit.
5. Click the **Configure** tool to display the Configure *debug_hardware* device dialog box. An example is shown in the following figure:

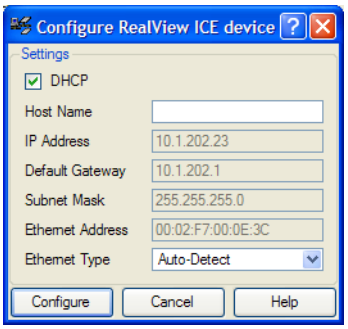


Figure 3-4 The Configure *debug_hardware* device dialog box

Note

The Ethernet Address field is read-only.

6. If you are not using DHCP:
   a. Deselect **DHCP**.
   b. Enter the required details in the following fields:
      - IP Address

- Default Gateway
- Subnet Mask.

c. Continue at step 9.

7. If you are using DHCP, select **DHCP**.

8. Enter the hostname in the Host Name field. This must contain only the alphanumeric characters (`A-Z`, `a-z`, and `0-9`) and the `-` character, and must be no more than 255 characters long.

9. Select the required Ethernet Type:
   - if you know the type of network that you are using, select that type
   - otherwise, select **Auto-Detect**.

10. Click **Configure**.

    The debug hardware unit restarts. During the restart, the unit is not present in the list. When the unit has restarted, it re-appears in the list of units with the new network settings.

    Note

    If the debug hardware unit is using DHCP, the list of units might display its **IP Address** as `127.0.0.2`. This is a dummy address that the debug hardware unit uses when it fails to obtain an IP address from the DHCP server.

    The list of units shows the correct address if the DHCP server has assigned it.

3.7.3 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Scanning for available debug hardware units* on page 2-4
- *Determining the correct network settings* on page 3-3
- *Troubleshooting* on page 3-13
- *Configuring the network settings for a debug hardware unit* on page 3-8
- Chapter 10 *Troubleshooting your debug hardware unit*.

Concepts

- *About configuring network settings* on page 3-2.

Reference

- *The Configure debug_hardware device dialog box* on page 3-4
- *The Configure new debug_hardware device dialog box* on page 3-6.

3.8 Restarting your debug hardware unit

The RVI Config IP utility restarts the networking software on the debug hardware unit whenever you change its settings. If necessary, select **Restart** from the **RVI** menu to force the networking software to restart.

You might want to restart the networking software on the debug hardware unit to get new network settings from the DHCP server. To do this, select **Restart** from the RVI menu.

3.8.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3.

3.9 Troubleshooting

If you encounter problems when configuring network settings for your debug hardware unit, see the following:

3.9.1 Why am I unable to see my DSTREAM or RVI unit on the network?

Seeing or browsing for DSTREAM or RVI units on the network relies on the local area network (LAN) allowing propagation of broadcast packets (UDP) on ports 30000 and 30001. It is common to limit the propagation of these types of packets to a localized network region to prevent congestion, but it might be possible to allow propagation of packets on these specific ports. Contact your network administrator to request this modification.

Note

This issue is seen when a DSTREAM or RVI unit is behind any network component that filters network traffic, such as a firewall.

3.9.2 When is it appropriate to assign a fixed IP address to my DSTREAM or RVI unit?

If it is not possible to browse for the DSTREAM or RVI unit on the network using the tools, you can attempt to locate the unit by specifying the host name of the unit, for example `MyDSTREAM.local.example.com`.

If the host name cannot be resolved, you can use an IP address, for example `192.168.1.16`. In this case, you might want to assign a fixed IP address to the DSTREAM or RVI unit to prevent this IP address from changing. To request a fixed IP address, contact your network administrator. When the address is assigned to the DSTREAM or RVI unit, you can confirm its correct operation by using the ping command from a DOS or UNIX prompt prior to connecting the tools.

A fixed IP address is also appropriate when an Ethernet cross-over cable is used. In this case, a private network between the host PC and the unit is created, although this might not be necessary due to the availability of a USB connection.

3.9.3 Why does my debug connection fail when I connect the Mictor cable to my target?

Some target systems have their debug signals connected to both a Mictor trace connector and a separate debug-only connector. In this scenario, if you connect the Mictor cable alongside another debug cable, there is effectively a large unterminated stub on the debug signals. This can cause the debug interface to become unstable. To solve this problem, configure the DSTREAM or RVI software to use the Mictor cable for both the debug and trace signals, and disconnect any other debug cables.

3.9.4 See also

Tasks

- *Configuring the network settings for a debug hardware unit* on page 3-8
- *Modifying the network settings for a debug hardware unit* on page 3-10.

Concepts

- *The Configure debug_hardware device dialog box* on page 3-4
- *Debug hardware unit network settings* on page 3-7.

Chapter 4
Managing the firmware on your debug hardware unit

The following topics describe how to manage and update the software that is installed on the debug hardware unit, using the RVI Update utility:

4.1 About templates and firmware files

The debug hardware unit stores templates for each supported device. Each template defines how to communicate with the device and the settings that you can configure for that device. The templates ar provided in a firmware file.

ARM periodically releases updates and patches to the firmware that is installed on a debug hardware unit. Each update or patch is released as a firmware file. The firmware filename has the following syntax:

`ARM-RVI-N.n.p-bld-type.unit`

where:

N.n.p is the version of the firmware. For example, 4.2.0 is the first release of firmware version 4.2.

build is a build number.

type is either:

- `base` the first release of the firmware for vesion *N.n*
- `patch` updates to the corresponding *N.n* release of the firmware.

unit identifies the debug hardware unit, and is one of:

- `dstream` for a DSTREAM debug and trace unit
- `rvi` for an RVI debug unit.

For example, patch 10 for DSTREAM firmware v4.2 is in the file:

`ARM-RVI-4.2.10-10-patch.dstream`

These might extend the capabilities of your debug hardware, or might fix an issue that has become apparent.

4.1.1 See also

Tasks

- *Viewing software version numbers* on page 4-4
- *Installing a firmware update or patch release* on page 4-5.

Concepts

- *Location of the firmware files in ARM products* on page 4-3.

4.2 Location of the firmware files in ARM products

You can obtain the firmware files from the ARM web site or from an installed location depending on your ARM development product:

- For DS-5, the firmware files are located in:

  install_directory\DS-5\sw\debughw\firmware

- For RVDS, the firmware files are located in:

  install_directory\ARM\RVI\Firmware*N.n**buildnumber*

See the *Getting Started* document of your ARM product for more information.

4.2.1 See also

Concepts

- *About templates and firmware files* on page 4-2.

4.3 Viewing software version numbers

To view software version numbers, select **Version Info...** from the **RVI** menu in the RVI Update utility. The Version Info dialog box displays a window giving version information. An example is shown in the following figure:

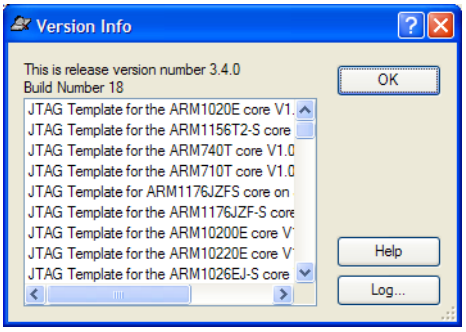


Figure 4-1 Version information

The text above the scrolling list shows the version number of the software release that is installed, in the format:

`This is release version number` *`major.minor.patch`*

where:

`major` is the major release version number

`minor` is the minor release version number

`patch` is the patch level of the *`major.minor`* version.

The scrolling list shows the version number of each component of the installed software.

The **Log...** button enables you to save the version information to a file. To do this:

1. Click **Log**. The Select Log File Name dialog box is displayed.
2. Choose the location of the log file.
3. Click **Save** to save the log file.
4. Click **OK** to close the Version Info dialog box.

4.3.1 See also

Tasks

- *Installing a firmware update or patch release* on page 4-5.

4.4 Installing a firmware update or patch release

You perform the same basic procedure for updating your firmware to a new version or applying a patch to the current version. However, if you are updating from a previous version of the firmware, for example from 4.0 to 4.1, you must upgrade to the base version first, then apply any patch related to that version. For example, if you have a DSTREAM unit:

1. Update your DSTREAM unit with `ARM-RVI-4.1.0-14-base.dstream`.
2. If a patch file is provided, such as `ARM-RVI-4.1.16-16-patch.dstream`, apply the patch to your DSTREAM unit.

If you want to restore the firmware to its original state after installing an upgrade, you can reinstall the original firmware file. This file is obtainable from the ARM website.

Note

If you have been running the Log Client utility, then make sure you stop it before performing an update. Although the Log Client utility does not prevent the update fromcompleting, it can take longer to complete.

To install an update or patch to the firmware on a debug hardware unit:

1. To open the RVI Update utility:
   - In DS-5, select:
     Start → All Programs → ARM DS-5 → Debug Hardware → Debug Hardware Update
   - In RVDS, select:
     Start → All Programs → ARM → RealView ICE vN.n → RealView ICE Update

   The RVI Update utility is displayed. An example is shown in the following figure:

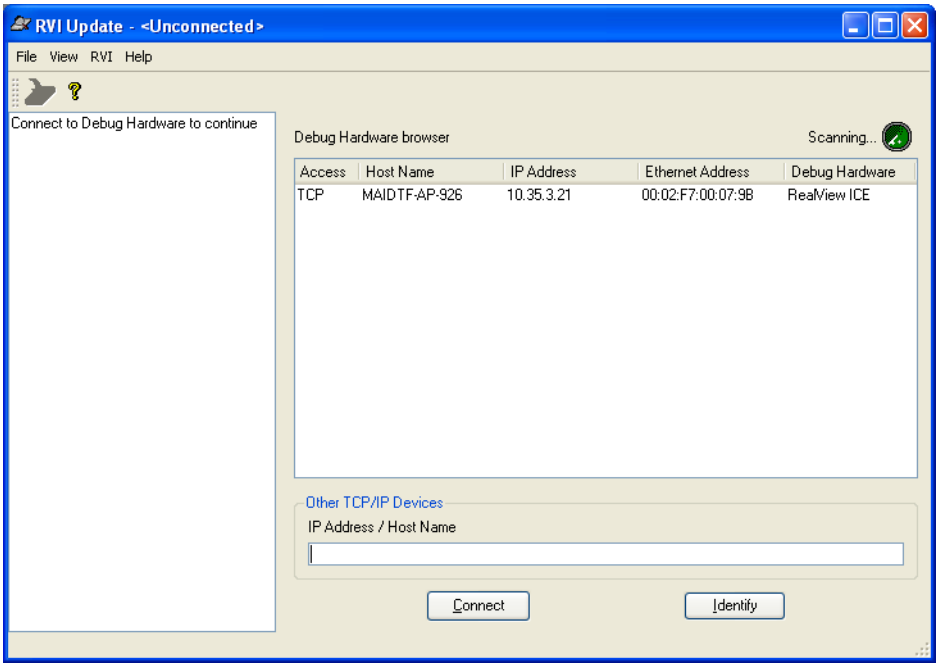


Figure 4-2 RVI Update utility

2. Either:
   - select the debug hardware unit from the list
   - enter the IP address or host name of the debug hardware unit in the Other TCP/IP Devices field.

3. Click **Connect** to connect to the debug hardware unit. Details of the existing firmware on the unit are displayed. An example is shown in the following figure:

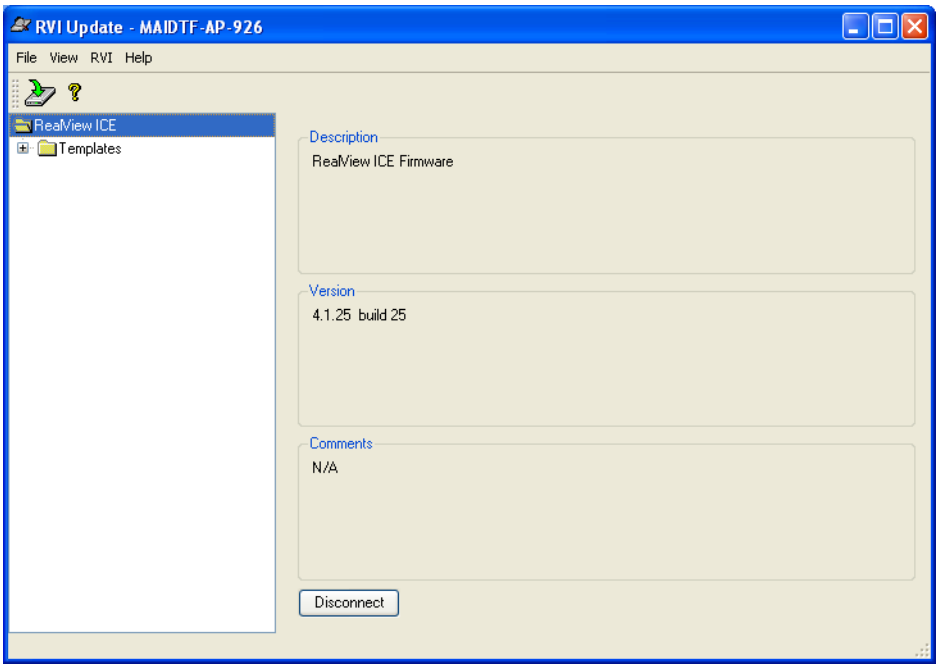


Figure 4-3 Firmware details

4. In the RVI Update utility, click the **Install Firmware** tool. The Select Firmware Update to Install dialog box is displayed. An example is shown in the following figure:

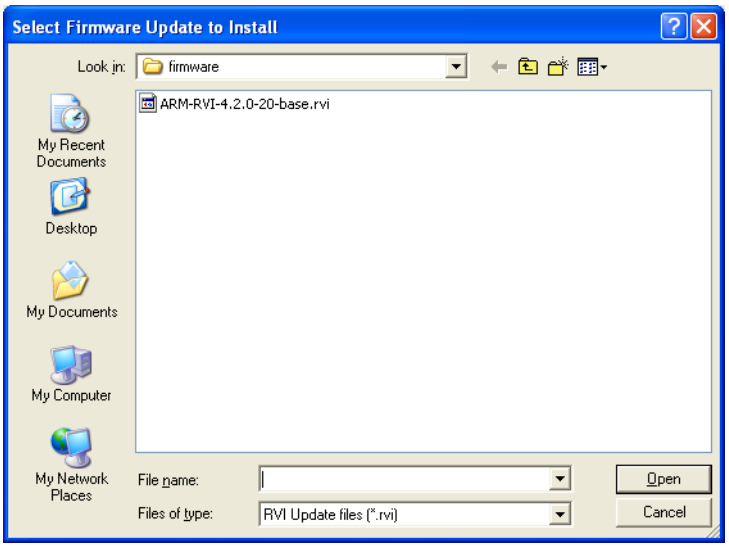


Figure 4-4 Selecting the component file to install

5. Navigate to the directory containing the component file for the update or patch that you want to install, and select the required file.

6. Click **Open**. After a short delay, a dialog box appears that describes what is in the component file, as shown in the following figure:

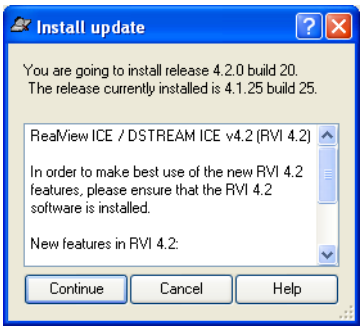


Figure 4-5 Confirming that you want to install the component file

When attempting to install a firmware update file, if you are using an older version of RVI Update, for example a pre-RVI v1.5 release, an error message appears as shown in the following figure:

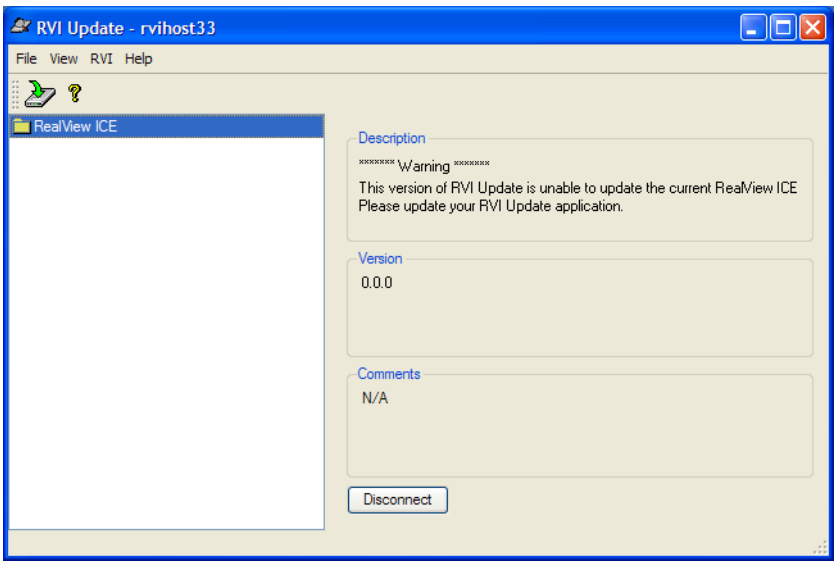


Figure 4-6 Warning message

Note

Before proceeding with your firmware update, you must upgrade your RVI Update utility to the latest software.

Similarly, if you are using a version of hardware that is incompatible with the firmware you are attempting to install, an error message similar to the one shown in the following figure:

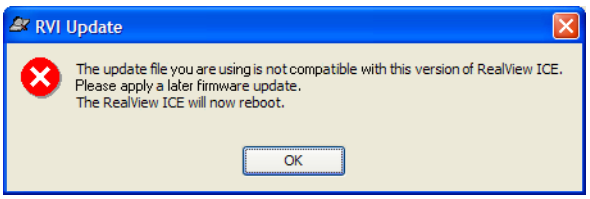


Figure 4-7 Error when using an incompatible version of hardware

7. In the Install update dialog box, click:
   - **Continue** to confirm that you want to install the components
   - **Cancel** to make no change to the debug hardware unit.

   When you click **Continue**, the RVI Update utility uploads the component file to the debug hardware unit. The debug hardware unit unpacks the component file, and installs the update or patch that it contains. The progress of the installation is displayed as shown in the following figure:

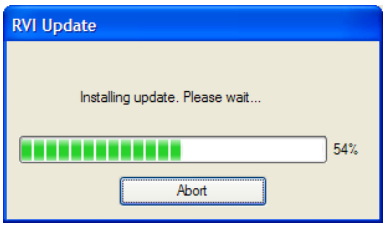


Figure 4-8 Progress during an installation

The debug hardware unit might automatically reboot itself as part of this procedure, depending on the patch or update that you are installing. The progress of the reboot is displayed as shown in the following figure:

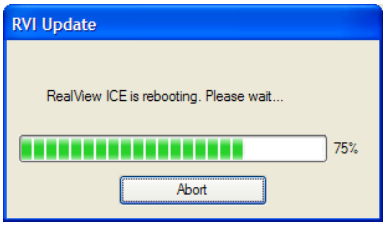


Figure 4-9 Progress when rebooting during an installation

During the installation, the FLASH LED lights up, showing that the unit is accessing its internal flash storage. During this time, do not disconnect power from the debug hardware unit. If a problem occurs during the installation, you must troubleshoot the firmware upgrade installation.

Note

During the installation, the **Abort** button is enabled. This means that you can safely stop the installation from proceeding by clicking this button. If the **Abort** button is not enabled, for example during rebooting, you cannot stop the reboot.

When the installation is complete, a message is displayed as shown in the following figure:

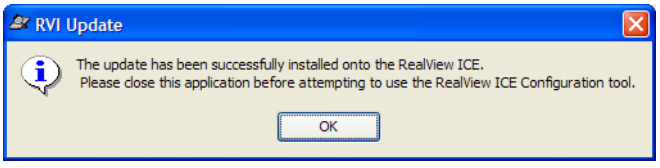


Figure 4-10 Message showing a successful installation

4.4.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Viewing software version numbers* on page 4-4

- *Upgrading an LVDS probe* on page 4-10.

Concepts

- *About templates and firmware files* on page 4-2
- *Location of the firmware files in ARM products* on page 4-3.
- *Troubleshooting firmware upgrade installations* on page 10-9.

Reference

- *Log Client Utility* on page 10-13.

Other information

- ARM web site, `http://www.arm.com`

4.5 Upgrading an LVDS probe

You can use the RVI Update utility to install an upgrade to your *Low Voltage Differential Signaling* (LVDS) probe. This upgrade procedure is necessary only if you want to make use of the *Serial Wire Debug* (SWD) feature. This is a once-only upgrade that is required if your LVDS probe was released with RVI v3.0, because this type of probe is not SWD-capable.

To upgrade your LVDS probe:

1. In the RVI Update utility, select **Upgrade LVDS Probe...** from the **RVI** menu.
2. You are prompted to confirm your option to upgrade the probe. Select **Yes**, and the RVI Update utility begins the update process, during which you are reminded not to disconnect the probe, nor to power off your debug hardware unit, until the process is completed. This is shown in the following figure:

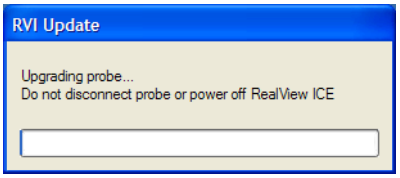


Figure 4-11 Progress during probe update

Note

To perform the upgrade you must have v3.1 firmware or later installed on your debug hardware unit.

You must also have an LVDS probe that is at least at v2.

If you have a v1 probe (board number HPI-0090x), you must replace it with a later version. If so, contact ARM for more information.

4.5.1 See also

Tasks

- *Installing a firmware update or patch release* on page 4-5.

Concepts

- *Troubleshooting firmware upgrade installations* on page 10-9.

Reference

ARM® DSTREAM™ System and Interface Design Reference:

- Serial Wire Debug , `../com.arm.doc.dui0499e/BEHIADEG.html`

ARM® RVT™ and RVT™ System and Interface Design Reference:

- Serial Wire Debug , `../com.arm.doc.dui0517e/yCHDBDBHI.html`

4.6 Restarting the debug hardware unit in RVI Update

To restart the debug hardware unit, select **Restart** from the **RVI** menu. RVI Update reboots the debug hardware unit, waits for the reboot to finish, then reconnects automatically. A message is displayed telling you that debug hardware is rebooting.

4.6.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Connecting to a debug hardware unit* on page 2-7
- *Installing a firmware update or patch release* on page 4-5
- *Upgrading an LVDS probe* on page 4-10.

Concepts

- Chapter 10 *Troubleshooting your debug hardware unit*
- *Viewing software version numbers* on page 4-4.

Reference

- Chapter 3 *Configuring network settings for your debug hardware unit*.

Chapter 5
Creating debug hardware target configurations

The following topics describe how to create a debug hardware configuration file with the Debug Hardware Config utility for use by your debugger:

5.1 About creating debug hardware target configurations

A debug hardware target configuration enables your debugger to:

- connect to the target devices on your development platform
- debug applications on your development platform.

You save your debug hardware target configuration in a configuration file. You reference this configuration file when you create target connections in your debugger.

CoreSight systems can contain many trace sources and sinks. To allow your debugger to capture trace correctly from a system, and to associate the trace information with the source that generated it, you must set up CoreSight associations for your debug hardware configuration.

5.1.1 See also

Tasks

- *Creating a debug hardware configuration file* on page 5-4
- Chapter 6 *Configuring CoreSight systems*.

Concepts

- *About CoreSight system configuration* on page 6-2
- *About trace associations* on page 6-6.

5.2 Creating a debug hardware configuration file

To create a debug hardware configuration file:

1. Start the Debug Hardware Config utility. An example is shown in the following figure:

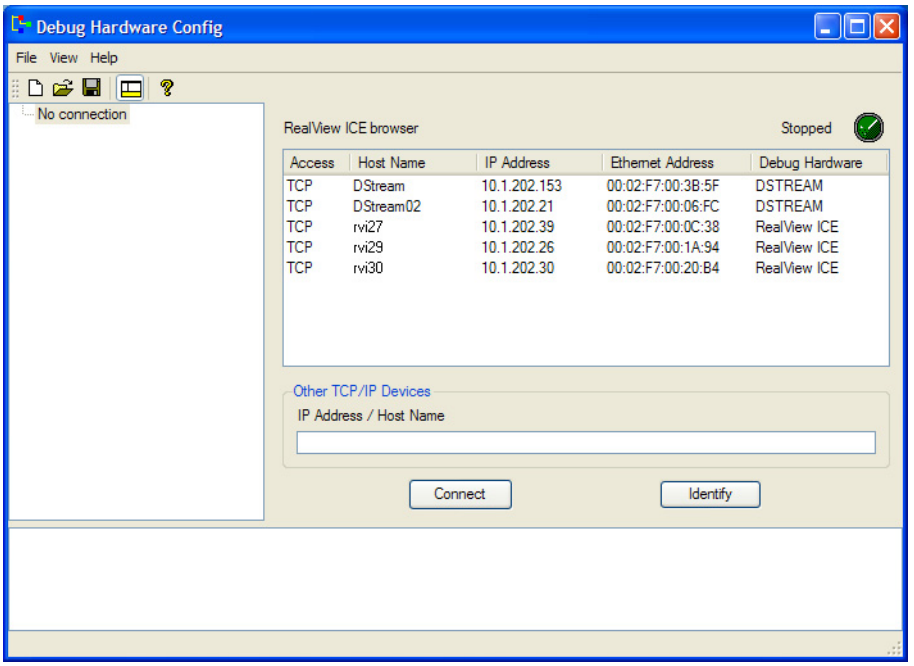


Figure 5-1 Debug HardwareConfig utility

2. Connect to a debug hardware unit.
3. Click **Auto Configure** to identify the target devices on your development platform.
4. If a platform configuration exists for your development platform, you are prompted to select that platform configuration.

   Click **OK** to use the platform configuration. Otherwise, click **Cancel**.
5. Configure the following as required:
   - device configuration settings
   - debug hardware Advanced settings
   - device properties, if appropriate
   - Trace configuration settings (DSTREAM and RVT2 only)
   - Trace Associations if you are configuring a CoreSight development platform.
6. Select **Save** from the **File** menu to save your configuration. The Choose a filename to save as dialog box is displayed.
7. Locate a directory to save your configuration file and enter an appropriate filename. For example **CoreSight_A8.rvc** for a CoreSight system containing a Cortex-A8 processor.
8. Click **Save** to save the file.
9. Select **Exit** from the **File** menu to close the Debug Hardware Config utility.

5.2.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Connecting to a debug hardware unit* on page 2-7
- *Configuring a JTAG scan chain* on page 5-7
- *Changing the properties of a device* on page 5-23
- *Setting up a CoreSight trace association file* on page 6-11
- *Configuring the debug hardware Advanced settings* on page 5-47
- *Defining CoreSight trace associations* on page 6-7.

Concepts

- *About configuring a device list* on page 5-9
- *CoreSight autodetection* on page 6-4
- *About adaptive clocking* on page 5-25
- *About CoreSight system configuration* on page 6-2
- *CoreSight autodetection* on page 6-4
- *About platform detection and selection* on page 5-40
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *Select Platform dialog box* on page 5-18
- *Device Properties dialog box* on page 5-21
- *Trace Association Editor dialog box* on page 6-9
- *Debug hardware device configuration settings* on page 5-26
- *Debug hardware Advanced configuration settings* on page 5-33
- *Debug hardware Trace configuration settings* on page 5-36
- *Trace Association Editor dialog box* on page 6-9
- *About trace associations* on page 6-6
- *Format of trace associations* on page 6-8
- Chapter 10 *Troubleshooting your debug hardware unit*.

5.3 Opening an existing debug hardware configuration file in Debug Hardware Config

To open a debug hardware configuration file, `.rvc`, in Debug Hardware Config:

1. Select **Open** from the **File** menu, and the Choose a file to open dialog box appears.
2. Locate the directory continuing your `.rvc` configuration files.
3. Select the appropriate `.rvc` file.
4. Click **Open**. The scan chain configuration is displayed, as shown in the following figure:

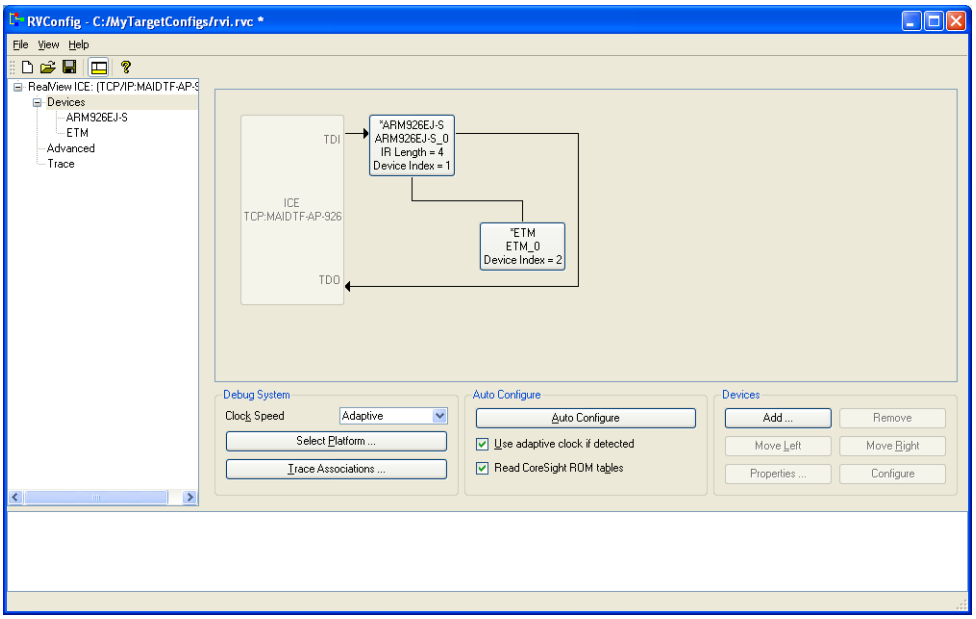


Figure 5-2 The scan chain controls

The title of the utility includes the full path to the configuration file. The path name might be different to that shown.

5. Modify your configuration as required.

5.3.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3.

Concepts

- *About creating debug hardware target configurations* on page 5-3.

5.4 Configuring a JTAG scan chain

Use the scan chain controls to configure a scan chain for the currently connected debug hardware unit:

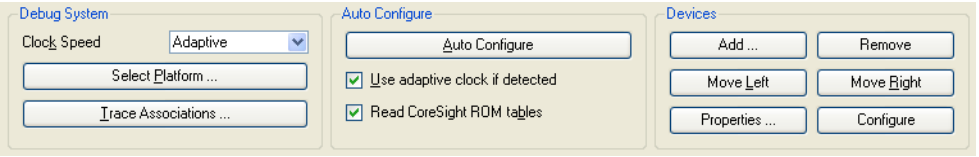


Figure 5-3 Scan chain controls

As you add devices to the JTAG scan chain, a schematic diagram of the scan chain is created, as shown in the following example:

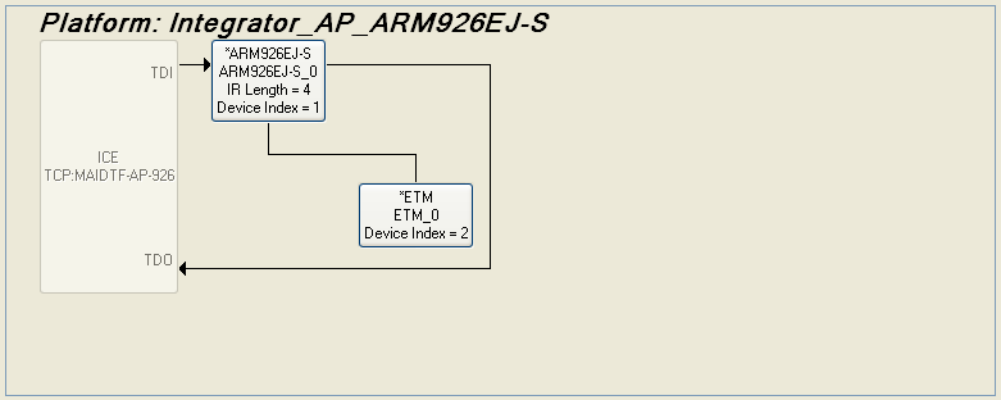


Figure 5-4 Scan chain schematic diagram

If a platform configuration file exists for your target, the Select Platform dialog box is displayed. If you want to use the platform configuration file:

1. Select the platform.
2. Click **OK**. A label identifying the platform is included at the top of the schematic diagram.

5.4.1 Managing devices

The following buttons enable you to manage the devices in your configuration:

- Click **Add...** to add a device to the scan chain.
- When a device is selected in the schematic diagram:
  - click **Remove** to remove the selected device
  - click **Properties...** to update the properties for the selected device
  - click **Configure** to display the Device configuration settings
- For a scan chain containing multiple devices, click:
  - **Move Left** to move the selected device to the left
  - **Move Right** to move the selected device to the right.

5.4.2 Device context menu controls

To display the device context menu, right-click on a device in the scan chain schematic. The following options are available for all devices:

- Select **Properties...** to change the properties of the device.
- Select **Configuration...** to configure the device.
- Select **Remove Device** to remove the chosen device.

For a CoreSight system, the **Read CoreSight ROM table** option is available for the ARMCS-DP device. This enables your debug hardware to read the CoreSight ROM table.

5.4.3 Managing a platform file

To manage a platform file for your development board:

If a platform file exists, click **Select Platform...** to choose a platform that corresponds to your development board.

When a platform is assigned to your configuration, the **Select Platform ...** button changes to **Clear Platform**. Click **Clear Platform** to remove the platform assignment from your configuration.

5.4.4 Managing trace associations

If you are configuring a CoreSight system, click the **Trace Associations ...** to manage the CoreSight trace associations.

5.4.5 See also

Tasks

- *Connecting to a debug hardware unit* on page 2-7
- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17
- *Changing the properties of a device* on page 5-23
- *Setting the clock speed* on page 5-24
- *Setting up a CoreSight trace association file* on page 6-11.

Concepts

- *About configuring a device list* on page 5-9.

Reference

- *Select Platform dialog box* on page 5-18
- *Device Properties dialog box* on page 5-21
- *Trace Association Editor dialog box* on page 6-9
- *Debug hardware device configuration settings* on page 5-26.

5.5 About configuring a device list

You can configure a device list using the following methods:

- autoconfiguration
- manual configuration.

5.5.1 Autoconfiguration

When autoconfiguring a device list, debug hardware interrogates the scan chain and automatically selects the correct templates for supported ARM target devices, then adds them to the scan chain in the correct order. This takes place at the current clock speed:

- If you are using a fixed clock speed, but debug hardware detects one or more devices that require adaptive clocking, it automatically selects adaptive clocking.
- If you are using adaptive clocking, but debug hardware does not detect any devices that support adaptive clocking, an error message is generated. Select a fixed clock speed.
- If the clock speed is too high, some devices on the scan chain might not be detected. If you suspect that this is happening, decrease the clock speed.

Warning

Autoconfiguring can be intrusive and stop your development platform from operating normally. If you want to connect to a target on your development platform without performing a reset and stop, you must manually add the devices to the scan chain.

Note

Autoconfiguration is disabled in Debug Hardware Config if the current debug hardware configuration has a platform assigned to it.

For *Serial Wire Debug* (SWD), autoconfiguring a system identifies the target devices on your development platform by reading appropriate SWD registers. The value of this register is usually set by the engineers that integrate the devices into a design. It is not set within the ARM devices themselves. For more information, see the ARM datasheet or technical reference manual for the processor that you are integrating.

Note

Before you autoconfigure a target that supports both JTAG and SWD, you must first enable **Use SWJ Switching** in the Advanced configuration settings.

5.5.2 Manual configuration

You can add devices manually to a scan chain. You must do this if your development platform includes:

- Unsupported devices. That is, devices for which no templates are provided by the debug hardware unit.
- Supported devices that are not detectable by the debug hardware unit.

5.5.3 CoreSight development platforms

If your development platform contains CoreSight devices, then autoconfiguration involves:

1. Adding the ARM CoreSight Debug Port (ARMCS-DP) device.
2. Reading the CoreSight ROM Table. This table contains a list of the CoreSight devices included in your development platform.

5.5.4 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17
- *Setting the clock speed* on page 5-24
- *Configuring a JTAG scan chain* on page 5-7
- *CoreSight autodetection* on page 6-4.

ARM® DSTREAM™ Setting up the Hardware:

- Connecting the DSTREAM debug and trace unit, `../com.arm.doc.dui0481e/I1004916.html`.

ARM® RVT™ and RVT™ Setting up the Hardware:

- Connecting the RVI debug unit, `../com.arm.doc.dui0515e/I1004916.html`.

Reference

- *Device Properties dialog box* on page 5-21.

5.6 Autoconfiguring a scan chain

To autoconfigure a scan chain:

1. Select the **Devices** node in the tree diagram.

   If a scan chain configuration is already set up and has a platform assigned to it, autoconfiguration is disabled in Debug Hardware Config. If you want to reconfigure the scan chain automatically, click **Clear Platform** before continuing.

   Note

   Before you autoconfigure a target that supports both JTAG and SWD, you must first enable **Use SWJ Switching** in the Advanced configuration settings.

2. Click on **Auto Configure**. Each detected device is added to the scan chain configuration list in the control pane, and is also added to the tree diagram. In many cases, this is all that you have to do to configure the scan chain. You must then configure the devices themselves.

3. If a scan chain configuration is already set up, the following Auto Configure Scan Chain prompt might be displayed:

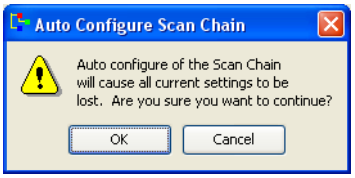


Figure 5-5 Auto Configure Scan Chain dialog box

   Caution

   If you click **OK**, your current scan chain configuration is lost.

5.6.1 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17
- *Setting the clock speed* on page 5-24
- *Troubleshooting autoconfigation of a scan chain* on page 10-11.

ARM® DSTREAM™ Setting up the Hardware:

- Connecting the DSTREAM debug and trace unit, `../com.arm.doc.dui0481e/I1004916.html`.

ARM® RVT™ and RVT™ Setting up the Hardware:

- Connecting the RVI debug unit, `../com.arm.doc.dui0515e/I1004916.html`.

Reference

- *Device Properties dialog box* on page 5-21
- *Debug hardware device configuration settings* on page 5-26
- *Debug hardware Advanced configuration settings* on page 5-33.

5.7 Adding devices to the scan chain

You can manually add devices to the scan chain, if required. You might want to do this in the following circumstances:

- You do not want to reset and stop the targets on the development platform.
- The autoconfiguration fails.
- The Read ROM Table autoconfiguration phase for a CoreSight system fails to find any devices.
- The device configuration on your development platform has changed since you created this debug hardware configuration, and the platform contains unsupported devices.

  For example, you might have added one or more devices to your development platform. In this case, you might also have to change the order of the devices, if the order on the development platform has changed.

To add a device to the scan chain:

1. Select the **Devices** node in the tree diagram.
2. Click on **Add...**. The Add Device dialog box is displayed. The following figure shows an example:

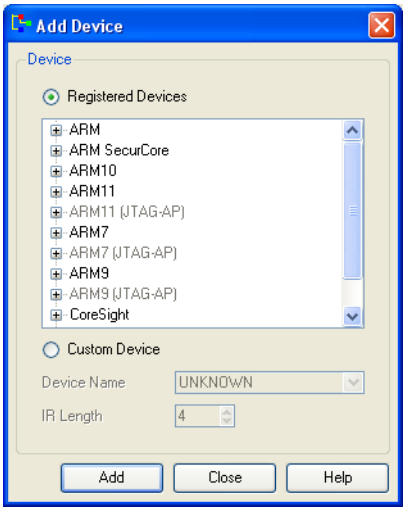


Figure 5-6 The Add Device dialog box

   The devices available depends on your firmware version.

   Note

   Device groups shown in light gray indicates that the related devices are not available. These are usually seen when you autoconfigure a CoreSight development platform.

3. If the device appears in the list of registered devices:
   a. Select **Registered Devices**. A registered device is a one that the debug hardware unit can identify and for which some level of debug support exists. That is, a template exists for the device.
   b. Expand the relevant device group.
   c. Select the device that you want to add.
   d. Click on **Add**. The device is added to the scan chain.

Note

You can also add a device to the scan chain by double-clicking the device.

e. If you have multiple devices, repeat steps 4b to 4d to add each registered device.

f. After you have added all your devices, continue at step 6.

4. You might want to add a device that is not identified by the debug hardware, because it is a device your are currently developing. To do this:

   a. Select **Custom Device**.

   b. Enter the name of the device in the Device Name field. This is used as the name of the device node in the tree view, and can have any value.

   c. Enter the JTAG *Instruction Register length* (in bits) in the IR Length field.

   Note

   If you enter an incorrect value for the IR length, any connections that you attempt to make to the device result in failure.

   d. Click on **Add**. The device is added to the scan chain.

   e. If you have multiple devices, repeat steps 5b to 5d to add each custom device.

Note

Be aware that you cannot debug a custom device. However, adding the custom device in the correct order and with the correct length enables you to debug the supported devices in the same scan chain.

5. When you have finished, click **Close**.

Note

You can remove devices, or change their order in the scan chain, without first having to close the Add Device dialog box.

The following figure shows a number of devices that have been added to the scan chain in a hierarchical manner:

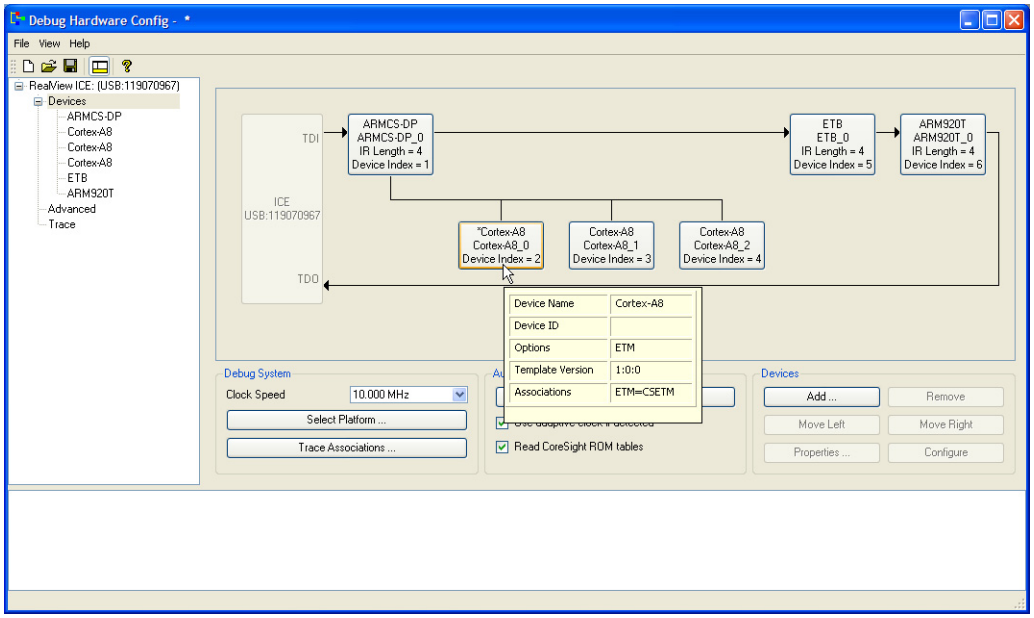


Figure 5-7 Scan chain devices with tooltip feature displayed

5.7.1 Considerations when adding devices to a scan chain

Be aware of the following:

- In a traditional JTAG configuration you must add devices in the correct order. The device nearest to **TDO** is last on the chain.

  Note

  If you add the devices in the wrong order, you can later change the order.

- In a CoreSight configuration, you must first add the ARMCS-DP device. You can then add the remaining CoreSight devices in any order.
- Hierarchies are created automatically when a device is added as a CoreSight component, and enables you to manage component interactions easily.
- The scan chain shows a unique association name for the device, if one exists, to help you to identify the correct device when you are creating associations.
- When the cursor is placed over a device, a tooltip displays details relating to that device.

  Note

  The Associations item in the tooltip only appears if an association has been set up for that device.

5.7.2 See also

Tasks

- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17
- *Setting the clock speed* on page 5-24
- *Setting up a CoreSight trace association file* on page 6-11.
- *Troubleshooting autoconfigation of a scan chain* on page 10-11.

Reference

- *Device Properties dialog box* on page 5-21.

5.8 Removing devices from the scan chain

To remove an unwanted device from the scan chain:

1. Select the **Devices** node in the tree diagram.
2. Select the device in the scan chain configuration.
3. Click **Remove**. If a device has a child component, a confirmation prompt is displayed.

Note

If a scan chain configuration is already set up and has a platform assigned to it, you cannot remove a device. If you want to remove a device from the scan chain:

1. Click **Clear Platform**.
2. Either:
   - autoconfigure the scan chain again
   - manually add only those devices you want to keep.

5.8.1 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Changing the order of devices on the scan chain* on page 5-17
- *Setting the clock speed* on page 5-24.

Reference

- *Device Properties dialog box* on page 5-21.

5.9 Changing the order of devices on the scan chain

For a traditional JTAG configuration, the devices must be in added the correct order in relation to debug hardware TDI and TDO. If you have added devices in the wrong order, then you can change the order.

To change the order of devices on the scan chain:

1. Select the **Devices** node in the tree diagram.
2. In the scan chain schematic diagram, select the device you want to move.
3. Click:
   - **Move Left** to move the device to the left
   - **Move Right** to move the device to the right.

Note

The ordering of CoreSight devices on the same *Debug Access Port* (DAP) is not important.

5.9.1 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Setting the clock speed* on page 5-24.

Reference

- *Device Properties dialog box* on page 5-21.

5.10 Select Platform dialog box

The Select Platform dialog box enables you to select a platform configuration that is suitable for your development platform. The following figure shows an example:

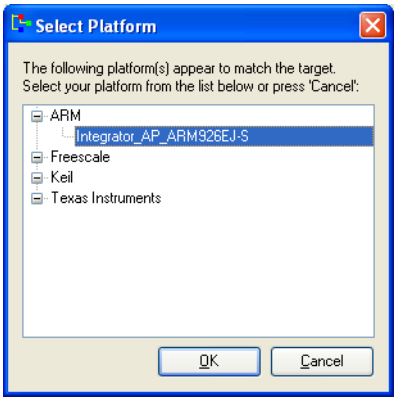


Figure 5-8 Select Platform dialog box

Click the **OK** button to select the chosen platform, in doing so the entire configuration for that platform is loaded.

Click the **Cancel** button to cancel any selected action.

Select **<None>** to put the debug hardware unit into a known state if you have incorrectly loaded a platform. This resets all settings to the default values.

5.10.1 See also

Tasks

- *Autodetecting a platform* on page 5-41
- *Manually selecting a platform* on page 5-43
- *Adding new platforms* on page 5-45
- *Adding autoconfigure support for new platforms* on page 5-46.

Concepts

- *About platform detection and selection* on page 5-40.

5.11 Export As Platform dialog box

The Export As Platform dialog box enables you to save the current configuration as a platform configuration. The following figure shows an example:

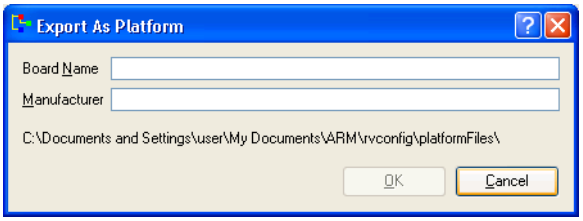


Figure 5-9 Export As Platform dialog box

5.11.1 See also

Tasks

- *Exporting a configuration to a platform file* on page 5-20
- *Autodetecting a platform* on page 5-41
- *Manually selecting a platform* on page 5-43
- *Adding new platforms* on page 5-45
- *Adding autoconfigure support for new platforms* on page 5-46.

Concepts

- *About platform detection and selection* on page 5-40.

5.12 Exporting a configuration to a platform file

To export a configuration to a platform file:

1. Select **Export platform file...** from the **File** menu to display the Export As Platform dialog box.
2. Specify a Board Name for the platform, for example **Integrator_AP_ARM926EJ-S**.
3. Specify a Manufacturer for the platform, for example **ARM**
4. Click **OK** to save the platform file and close the Export As Platform dialog box.

   The platform file is stored in:

   `C:\Documents and Settings\`*`username`*`\My Documents\ARM\rvconfig\platformFiles`

   The name of the platform file has the format:

   `Manufacturer_BoardName``.rvc`
5. Select **Save** from the **File** menu to save the configuration.

5.12.1 See also

Tasks

- *Autodetecting a platform* on page 5-41
- *Manually selecting a platform* on page 5-43
- *Adding new platforms* on page 5-45
- *Adding autoconfigure support for new platforms* on page 5-46.

Concepts

- *About platform detection and selection* on page 5-40.

Reference

- *Export As Platform dialog box* on page 5-19

5.13 Device Properties dialog box

The Device Properties dialog box enables you to change the properties of any device in the scan chain, if required. The following figure shows an example:

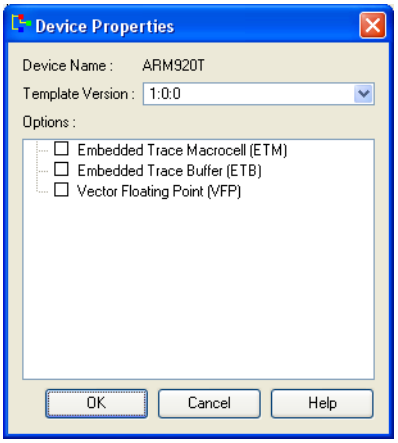


Figure 5-10 The Device Properties dialog box

You can:

- Select one or more options for the device. These options enable a debugger to determine the features that are supported by a device. For example, a Registers view might show the NEON registers when the `NEON SIMD Extensions (Neon)` option is selected.
- Set the template version for the device, if multiple versions are provided.

If no properties are available for a device, the following message is displayed in the Options list

```
No device options found in current template
```

5.13.1 Device properties

The device properties listed depends on the device being configured, and can be any of the following:

Embedded Trace Macrocell (ETM)

Select this if you want to capture trace from the ETM.

Embedded Trace Buffer (ETB)

Select this if you want to capture trace from an ETB. You must also select **Embedded Trace Macrocell (ETM)**.

Vector Floating Point (VFP)

Select this to use VFP, if supported.

Vector Floating Point v3 (VFPv3)

Select this to use VFPv3, if supported.

Vector Floating Point v3-D16 (VFPv3-D16)

Select this to use VFPv3-D16, if supported.

NEON SIMD Extensions (Neon)

Select this to use NEON, if supported.

5.13.2 See also

Tasks

- *Changing the properties of a device* on page 5-23.

5.14 Changing the properties of a device

You can change the properties of any device in the scan chain, if required. The properties available depend on the device you are configuring. For example, if you want to capture trace from an ETM, you must make sure the `Embedded Trace Macrocell (ETM)` option is selected.

To change the properties of a device:

1. Select the **Devices** node in the tree diagram.
2. Select the device in the scan chain configuration list.
3. Click **Properties...** to display the Device Properties dialog box. The following figure shows an example:

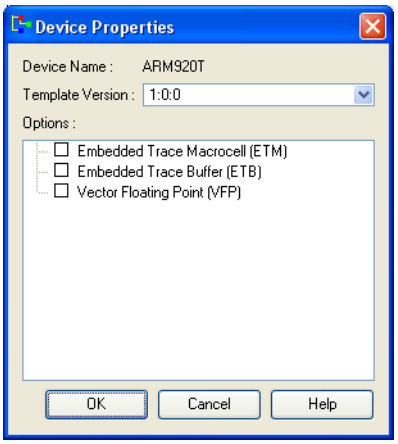


Figure 5-11 The Device Properties dialog box

Note

It is not possible to use an ETB without an ETM, so when you select ETB, ETM is selected automatically.

4. Select the options you require.

   The options available depend on the device you are configuring.
5. Click **OK**.

5.14.1 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17
- *Setting the clock speed* on page 5-24
- Chapter 4 *Managing the firmware on your debug hardware unit*.

Reference

- *Device Properties dialog box* on page 5-21.

5.15 Setting the clock speed

It is important to select the best clock speed for your system. Higher clock speeds enable faster downloads, but setting the clock speed too high can result in intermittent faults and reliability problems. If you are experiencing such problems, try manually reducing the clock speed. If you are not sure which clock speed to use, try setting the default speed, 10MHz.

Note

For RVI, for reliable operation at high clock speeds you must use the *Low Voltage Differential Signaling* (LVDS) cable.

5.15.1 Predefined clock speed

To select a predefined clock speed:

1. Select the **Devices** node in the tree diagram.
2. Select the clocking that you want to use from the Clock Speed drop-down list. The following figure shows an example:

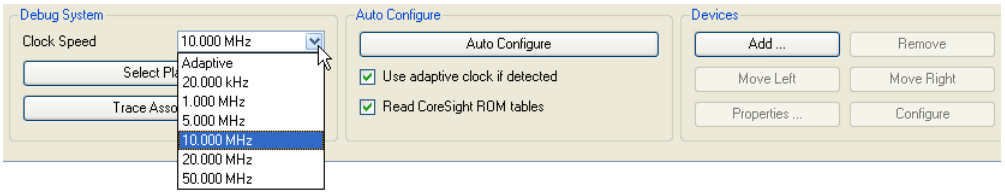


Figure 5-12 The scan chain speed controls

Note

Although the debug hardware can support JTAG clock speeds down to 13Hz, your debugging environment might become unstable at speeds lower than 1kHz.

5.15.2 Custom clock speed

If the clock speed you want to use is not available as a preset value, enter the required clock speed in the Clock Speed field with the **Hz**, **kHz**, or **MHz** suffix as required. For example, **40.0 kHz**.

5.15.3 See also

Tasks

- *Adding devices to the scan chain* on page 5-12
- *Removing devices from the scan chain* on page 5-16
- *Changing the order of devices on the scan chain* on page 5-17.

Reference

- *Device Properties dialog box* on page 5-21.

ARM® DSTREAM™ System and Interface Design Reference:

- DSTREAM System Design Guidelines, `../com.arm.doc.dui0499e/Chdbdcid.html`.

ARM® RVT™ and RVT™ System and Interface Design Reference:

- RVI and RVT System Design Guidelines, `../com.arm.doc.dui0517e/Chdbdcid.html`.

5.16 About adaptive clocking

Adaptive clocking enables your debug hardware unit to dynamically adjust the JTAG clock (**TCK**) following changes to the processor clock (**CLK**). This is useful when debugging system with low power modes and changing clocks in general.

Adaptive clocking is intended only for targets that are based on a synthesizable ARM® processor implementing **RTCK**. When a fixed JTAG clock is used, the JTAG clock must be running at most 1/6 of the processor clock. Using a fixed JTAG clock does not support changes in the processor clock speed, which can corrupt the JTAG connection.

If you use adaptive clocking, the maximum clock frequency is lower than with non-adaptive clocking, because of transmission delays, gate delays, and synchronization requirements.

5.16.1 See also

Reference

ARM® DSTREAM™ System and Interface Design Reference:

- DSTREAM System Design Guidelines, `../com.arm.doc.dui0499e/Chdbdcid.html`.

ARM® RVI™ and RVT™ System and Interface Design Reference:

- RVI and RVT System Design Guidelines, `../com.arm.doc.dui0517e/Chdbdcid.html`.

5.17 Debug hardware device configuration settings

You can configure settings that are specific to a target device on your development platform. The settings available depend on whether the device is a processor or a non-processor CoreSight device.

5.17.1 Processor device settings

Depending on the processor that you have selected, a selection of the following controls is available:

Allow execution with T-Bit Clear

The Cortex-M series processors can only execute Thumb2 code. However, their xPSR contains a bit to configure ARM/Thumb state, that is initialized from the reset vector. This enables you to test the exception handler associated with this exception type.

Bypass memory protection when in debug

This option enables the bypass of any memory protection provided by hardware (such as a memory management or protection unit) whenever the target hardware enters debug state. This means that you can access protected memory to set software breakpoints in it, or to alter its contents.

Clear breakpoint hardware on connect

This control is available if you are using the ARM11 family of processors. The debug logic of an ARM11 processor is not reset when a *Test Access Port* (TAP) reset is applied. Set this option to `True` to instruct the debug hardware unit to perform reset the debug logic each time you connect.

Code Sequence settings

Most systems store variables and the stack and heap in RAM. However, in some systems only Flash or ROM is mapped in memory at power-up and RAM must be enabled by software in the boot code.

To perform certain operations with some targets, the debug hardware unit must download a piece of code into memory and make the processor execute it. This code must be located in a writable area of memory (RAM) and must be accessed only by the JTAG tool.

You can set the address and size of this code using the **Code Sequence Address** and the **Code Sequence Size (bytes)** settings.

When you connect with a debugger to one of these targets, make sure that RAM is mapped in memory and that the cache clean code or code sequence address is mapped correctly. You can run a script from the command line to configure the target memory map straight after connecting to the target.

This area of memory must be:

- unused by the target
- readable
- writable
- non-cacheable (for cached targets)
- at least 128 bytes in size.

The debug software downloads code sequences to this area to perform various tasks, such as cleaning the cache and accessing certain system registers on targets such as ARM920T™ and ARM1136JF-S. It does not preserve the contents of this area.

Note

You must ensure that the **Code Sequence Address** and the **Code Sequence Size (bytes)** values are correctly set up before you attempt to write to any of the Cache Operations or TLB Operations in the your debugger Registers view. If you do not set these values correctly, your debugger gives one or more of the following errors:

- `Error V28305 (Vehicle): Memory operation failed`
- `Warning: Code sequence memory area size error`
- `Unable to load code sequence into defined memory area.`

The **Code Sequence Timeout (ms)** value sets a timeout for execution of the uploaded code sequence. For most targets, a 500ms timeout is sufficient.

The **Use code sequence to clean cache** option enables you to configure how the caches are cleaned if you are using ARM 920T or ARM922T processors. Set this option when using the debugger to access IO memory, for example peripheral control registers for *Universal Asynchronous Receiver/Transmitters* (UARTs), when caches are enabled.

CoreSight AP index

The AP index of the corresponding device.

Available only for processors that are part of a CoreSight system.

CoreSight base address

The base address of the CoreSight registers for the processor on the *Advanced High-performance Bus* (AHB) or *Advanced Peripheral Bus* (APB).

Available only for processors that are part of a CoreSight system.

Debug acceleration level

This controls the level of acceleration used in debug operations.

Select one of the following options:

0 - Full This is the default. It enables full use of the performance features of RVI and the target processor.

1 - Partial

This results in a lower performance than for the **Full** option.

2 - None Select the **None** option to use only basic operations. This results in the lowest performance available.

Note

In most instances, select the **Full** option unless advised otherwise by an ARM support engineer.

Default Gateway

Default gateway for the target when using virtual Ethernet. Used with the **IP Address** and **Network Mask** settings to enable access to your target from the network.

Fast Memory Download

This configuration item is available only for processors connected to a JTAG-AP port. Set this to `False` if you experience problems using fast memory downloads.

Ignore bad JTAG IDCODE

By default, debug hardware reads the device JTAG IDCODE to verify the integrity of the JTAG connection. The JTAG standard restricts the JTAG IDCODE value to be 32 bits long and requires the least significant bit to be a 1. If debug hardware reads an invalid (bad) JTAG IDCODE, it assumes that the JTAG connection is not functioning properly, and fails the attempt to connect to the processor.

You must set the **Ignore bad JTAG IDCODE** option according to whether you want to instruct debug hardware to enable connection to the processor even if it detects that the JTAG IDCODE is invalid.

Ignore debug privilege errors when starting core

When the SPIDEN line is changed from HIGH to LOW, the following errors might be seen:

- `Insufficient debug privilege to restore core state for restart.`
- `Insufficient debug privilege to write software breakpoint to memory.`
- `Set Ignore debug privilege errors when starting core to suppress these errors.`

If set to `True`, debug hardware starts the processor running even though the breakpoints/processor state is incorrect.

If set to `False` (the default), debug hardware refuses to start the processor and reports the errors.

IP Address IP address of the target when using virtual Ethernet. Used with the **Default Gateway** and **Network Mask** settings to enable access to your target from the network.

JTAG timeouts enabled

JTAG timeouts are enabled by default. You must disable these when debug hardware is connected to a processor using a low clock speed and adaptive clocking, because debug hardware cannot detect the clock speed when adaptive clocking is used, so cannot scale its internal timeouts. If a JTAG timeout occurs, the JTAG is left in an unknown state, and debug hardware cannot operate correctly without reconnecting to the processor.

Network Mask

Net Mask of the target when using virtual Ethernet. Used with the **IP Address** and **Default Gateway** settings to enable access to your target from the network.

No error if step-instr can't stop.

Controls generation of error messages if a debugger step instruction operation fails because a timeout attempts to stop the SecurCore processor after a step is complete. This can occur on the SecurCore if an instruction execution results in the processor clock being disabled using CLKEN. The processor appears to be in a running state.

If set to `True` (the default), then no error message appears if an instruction step results in the processor running.

If set to `False`, then an error dialog box is displayed in your debugger.

Post Reset State

Set to the required state for the target hardware:

Running The target hardware is running.

Stopped Controls the state of a processor after a reset. It is only available for devices that are capable of running (such as ARM processors). Each device on the scan chain does not have to be set to the same value, so it is valid to have one processor running and another stopped.

Note

If you want to connect to a running target without performing a reset and without stopping the target, you must do both of the following:

- In debug hardware, set the Post Reset State to **Running**.
- In your debugger, connect using the **No Reset/No Stop** connection mode.

Soft TAP reset on JTAG sync loss

In some situations, such as a processor entering low power mode, the synchronization between the debug unit and the TAP controller in the JTAG system can be lost. This can result in invalid values for the debug status register being read. To regain the synchronization, a soft TAP reset must be performed to get the TAP controller into a known state.

If Soft TAP reset on JTAG sync loss is checked, a soft TAP reset is performed to get the TAP controller into a known state if the debug unit reads invalid values for the debug status register.

Software breakpoint mode

If supported by your processors, this control enables you to configure how the debug hardware unit handles software breakpoints. Select the required breakpoint mode:

AUTO This is the default mode for all templates:

- If the processor being debugged supports `BKPT` instructions, debug hardware automatically uses the `BKPT` instruction for software breakpoints.
- Only one watchpoint resource is used for multiple software breakpoints. Therefore, if the processor being debugged does not support `BKPT` instructions, debug hardware uses the watchpoint unit resource when you set a software breakpoint. The debug hardware unit automatically frees the watchpoint unit resource when all software breakpoints are cleared.

NONE When this mode is selected, you cannot set software breakpoints. If you attempt to set a software breakpoint, debug hardware gives an error message telling you that there are no free resources to set the breakpoint.

WATCHPOINT

This mode instructs debug hardware to use one watchpoint unit to provide software breakpoint instructions, whether or not the processor being debugged supports `BKPT` instructions. Select this option if the processor supports `BKPT` instructions but you want to use a watchpoint unit.

BKPT This mode instructs debug hardware to use the `BKPT` instruction to provide software breakpoint instructions, whether or not the processor supports this instruction. Select this option if you want to make sure that no watchpoints are used.

Unwind vector when halt on vector catch

This control is available if you are using an ARM10 processor. It instructs the debug unit to unwind the vector if you have set a vector catch on a SWI, an Undefined instruction, a Prefetch Abort or a Data abort. Unwinding the vector sets the PC to the address of the code that caused the exception instead of leaving it at the vector address. The LR and CPSR are restored, and debugger displays the code at this address. This enables you to more easily examine the code that caused the exception. If you want to run the exception handling code, you must leave this option unchecked.

Note

This option is only activated if a vector catch occurs. If a vector catch is not set, then the exception handler is run as normal.

Unwind vector when halt on SWI

This control is available if you are using the following:

- a Cortex-A8 or Cortex-R4 processor
- ARM1136JFS-JTAG-AP, ARM1156T2FS-JTAG-AP, or ARM1176JZF-JTAG-AP device.

It instructs debug hardware to unwind the Supervisor Call (SVC) vector if you have set a vector catch on an SVC. Unwinding the vector sets the PC to the address of the code that caused the exception instead of leaving it at the vector address. The LR and CPSR are restored, and your debugger displays the code at this address. This enables you to more easily examine the code that caused the exception. If you want to run the exception handling code, you must leave this option unchecked.

Note

This option is only activated if an SVC vector catch occurs. If an SVC vector catch is not set, then the exception handler is run as normal.

Use LDM or STM for memory access

This options controls whether or not to use *Load Multiple Instructions* (LDM) or *Store Multiple Instructions* (STM) to access target memory. You might need to set this option if you have a peripheral that is not fully compatible with the AMBA 2.0 standard, as in such cases LDM and STM might not be compatible.

Write-Through L2 Cache when in debug

This option is available if you are using an ARM11 processor.

Use this option with platforms that have a level 2 cache that interferes with debug operations. By default it is set to `False`, but the platform configuration files supplied for affected platforms set it to `True`.

Note

If you set this option for other platforms, unexpected behavior might result, and cause errors.

5.17.2 Non-processor CoreSight device settings

For non-processor CoreSight devices, the following device configuration settings are available:

ARM11xx-JTAG-AP specific settings

The ARM1136JFS-JTAG-AP, ARM1156T2FS-JTAG-AP, and ARM1176JZF-JTAG-AP devices have the following CoreSight-specific controls:

JTAG-AP Port index for core

For CoreSight systems with processors connected to JTAG-AP, the port index on the JTAG-AP to which the processor is connected.

Pre-scan IR bits for Devices after the core on the JTAG-AP scanchain

The total length of the JTAG instruction registers (IRs) for devices appearing after the debugged target processor on the scan chain.

For example, if there are three devices after the processor with IR lengths of 5, 7, and 11, then set this to 23.

Post-scan IR bits for Devices before the core on the JTAG-AP scanchain

The total length of the JTAG instruction registers (IRs) for devices appearing before the debugged target processor on the scan chain.

For example, if there are two devices before the processor with IR lengths of 2 and 3, then set this value to 5.

Pre-scan DR bits for Devices after the core on the JTAG-AP scanchain

The total number of devices appearing after the debugged target processor on the scan chain.

For example, if there are three devices after the processor, then set this value to 3.

Post-scan DR bits for Devices before the core on the JTAG-AP scanchain

The total number of devices appearing before the debugged target processor on the scan chain.

For example, if there are two devices before the processor, then set this value to 2.

Fast memory download.

This control is available for those targets where the *Debug Access Port* (DAP) and the processor are running sufficiently fast enough to handle the data being sent to them by the debug unit. The debug unit does not have to check that each individual transaction with the DAP is successful.

Because the processor is behind the DAP, all processor accesses have to go through the DAP. As a guide, do not set this for FPGA-based targets, but only for real silicon.

If this option is set, then error checking is disabled and the user is not informed of any errors that occur. If problems are encountered when downloading images then uncheck this option to resolve them.

CoreSight AP index

The AP index of the device.

Available for all devices, except the ARMCS-DP, ARMJTAG-DP, and ARMSW-DP devices.

CoreSight base address

The base address of the CoreSight registers for the device on the AHB or APB.

Available for all devices, except the ARMCS-DP, ARMJTAG-DP, and ARMSW-DP devices.

Force ETM power up on connect

If the CoreSight ETM (CSETM) is powered down when your debugger attempts to connection to it, then power up the device.

Memory Access AP index

The index number of the AHB-AP memory bus on the DAP. The AHB-AP bus is used to perform memory operations within the CoreSight system.

Available for the ARMCS-DP, ARMJTAG-DP, and ARMSW-DP devices only.

The ARMCS-DP device represents the *Debug Access Port* (DAP) in a CoreSight system. This device is automatically detected when you autoconfigure a CoreSight system. However, any devices connected to the DAP are not detected. Therefore, you must read the CoreSight ROM table of the ARMCS-DP to determine the devices that are connected to the DAP.

The ARM11xx-JTAG-AP device represents the JTAG-AP in a CoreSight system. To debug CoreSight systems that have processors connected to the DAP the JTAG-AP, the debug unit must know the pre-scan and post-scan bits for JTAG operations.

Multiple devices on the JTAG scan chain are connected in series, with data flowing serially from TDI to TDO. This means that debugging a given target in the chain requires that certain pre-scan and post-scan bits are used. These bits ensure that the other devices are not affected by the data directed at the target device, and that the data is positioned correctly in the serial scan for the target device.

5.17.3 See also

Tasks

- *Configuring CoreSight processors* on page 6-22
- *Configuring ARM7, ARM9, and ARM11 processors in CoreSight systems* on page 6-24
- *Configuring a target processor for virtual Ethernet* on page 5-51.

Concepts

- *Strategies used by debug hardware to debug cached processors* on page 8-15.

5.18 Debug hardware Advanced configuration settings

Depending on your development platform configuration, a selection of the following controls is available in the Advanced settings:

Allow ICE to perform TAP reset

Set this item to `True` to allow your debug hardware unit to hold the **nSRST** line active long enough to perform any post-reset setup that might be required after a target-initiated reset. This can extend the time that the target is held in reset. If this item is set to `False`, your debug hardware unit does not assert the reset line, but post-reset setup might not be complete before the target starts to run.

Allow ICE to latch System Reset

Set this item to `True` to allow your debug hardware unit to perform **nTRST** reset while holding **nSRST**. This ensures that the *Test Access Port* (TAP) state machine and associated debug logic is properly reset.

LVDS Debug Interface mode

This can be set either to JTAG or SWD. If set to SWD, this causes RVI to connect to the target using the SWD protocol instead of JTAG.

nSRST High mode

Selects the drive strength used when the reset signal is in the high, or inactive, state. Output can be driven as a strong or weak high, or not driven (tri-state).

nTRST High mode

Selects the drive strength used when the reset signal is in the high, or inactive, state. Output can be driven as a strong or weak high, or not driven (tri-state).

nSRST Low mode

Selects the drive strength used when the reset signal is in the low, or active, state. Output can be driven as a strong or weak low, or not driven (tri-state).

nTRST Low mode

Selects the drive strength used when the reset signal is in the low, or active, state. Output can be driven as a strong or weak low, or not driven (tri-state).

nSRST Hold Time (ms)

Specifies how long the debug hardware unit holds the hardware **nSRST** system reset signal LOW.

nTRST Hold Time (ms)

Specifies how long the debug hardware unit holds the **nTRST** TAP reset signal LOW.

nSRST Post Reset Delay (ms)

Specifies how long after the hardware **nSRST** system reset before the debug hardware unit enters the Post Reset State.

nTRST Post Reset Delay (ms)

Specifies how long after the **nTRST** TAP reset before the debug hardware unit enters the Post Reset State.

Perform TAP reset on first connect

Resets the target hardware whenever you connect.

Perform SYS reset on first connect

Resets the target hardware by asserting the **nSRST** signal when connecting to the first device in a debug session.

Reset Type One of the following:

nSRST	Resets the hardware by holding the hardware **nSRST** system reset signal LOW. This is the default.
nTRST	Resets the target TAP by holding the **nTRST** TAP reset signal LOW.
nSRST+nTRST	Resets the hardware and the target TAP by holding both the hardware **nSRST** system reset signal and the **nTRST** TAP reset signal LOW.
Fake	Resets the system by entering supervisor mode, and setting the program counter to the address of the reset vector (known as a *soft reset*).
Ctrl_Reg	The Control register. This reset, in instances where processors have a reset register, enables you to reset the processor without using the external reset lines. If you set the reset type to `Ctrl_Reg`, then this control register is used.

SWO Mode Set to Manchester or UART, depending on the target mode.

If the **SWO Mode** is set to UART, the debug hardware unit is able to detect the **SWO UART Baud rate**.

This setting has no effect in Manchester mode.

SWO UART Baud rate

For the frequency of the incoming data. If you set this to 0, the system attempts to autodetect the baud rate.

Note

UART mode in the SWO context also means *Non Return to Zero* (NRZ) mode.

TAP Reset via State Transitions

If you want the JTAG logic in the target hardware to be reset by forcing transitions within its state machine. This is done in addition to holding the **nTRST** TAP reset signal LOW. Select this option if **nTRST** is not connected, or if the target hardware requires that you force a reset of the JTAG logic whenever resetting.

Target nSRST + nTRST linked

If the target hardware has its **nSRST** and **nTRST** JTAG signals linked.

Use SWJ Switching

If this is set, it causes the SWJ switching sequence to be sent before connecting to the target device. On devices that support SWJ switching, this causes the DAP to switch its interface to the selected protocol.

Note

If a target supports both JTAG and SWD, you must enable this setting before you autoconfigure that target.

Use deprecated SWJ Sequence

If this is set, it causes your debug hardware unit to use an alternative SWJ switching sequence, used on some older SWD-compatible targets. This option is normally clear, unless the processor requires the deprecated sequence.

Note

For RVI, the **LVDS Debug Interface mode**, **Use SWJ Switching**, and **Use deprecated SWJ Sequence** controls are not present in the control pane if you are not using a *Low Voltage Differential Signaling* (LVDS) probe.

User Output*n*

Used to set the state of the **USER IO** pins on the rear of the RVI unit, or on the front of the DSTREAM unit.

5.18.1 See also

Tasks

- *Configuring the debug hardware Advanced settings* on page 5-47.
- *Debugging with a reset register* on page 8-20.

Reference

ARM® DSTREAM™ System and Interface Design Reference:

- DSTREAM reset signals, `../com.arm.doc.dui0499e/CCHDEDGG.html`.

ARM® RVT™ and RVT™ System and Interface Design Reference:

- RVI reset signals, `../com.arm.doc.dui0517e/CHDDDFEJ.html`.

5.19 Debug hardware Trace configuration settings

The Trace configuration settings enable you to configure delays on the trace lines:

Delay Trace Clock, Delay Trace Signal *N*

Delay line *N* by a specified amount of time, expressed in intervals of 75 picosecond. Default delays are configured into the unit, and you are able to delay each signal by a specified amount relative to these defaults, to allow for variations in target hardware.

Invert Trace Clock

Invert the clock so that data is sampled on the falling edge, rather than on the rising edge, of the clock.

5.19.1 See also

Tasks

- *Configuring trace lines (DSTREAM and RVT2 only)* on page 7-4
- *Configuring your debugger for trace capture* on page 7-6.

Concepts

- *About using trace hardware* on page 7-2.

5.20 Debug hardware Advanced configuration reset options

The following Advanced configuration settings are available when configuring reset options in debug hardware:

Allow ICE to latch System Reset (AllowICELatchSysRst)

When enabled, this option enables the debug hardware unit to extend the time the target controller holds the target in reset. This enables the debug hardware unit to apply breakpoint settings before the processor starts execution. This is useful for debugging a target from reset, and allows the unit to stop the processor on the first instruction fetch after reset has been released by the unit.

When this option is disabled, the breakpoint settings might only take effect after the processor has already started execution, preventing debugging of the application reset handler.

The default setting is `True`.

Allow ICE to perform TAP Reset (AllowICETAPReset)

A *Test Access Port* (TAP) reset on early processors, such as the ARM7TDMI, also reset the debug registers associated with the JTAG device.

Later processor, such as the ARM1176, are not affected by a TAP reset. The main purpose of a TAP reset is to reset the JTAG state machine in the TAP controller receiving commands from the debug hardware unit.

The default setting is `True`.

5.20.1 See also

Concepts

- *Configuring SecurCore behavior if the processor clock stops when stepping instructions* on page 5-38
- *Configuring TrustZone enabled processor behavior when debug privileges are reduced* on page 5-39.

Reference

- *Debug hardware Advanced configuration settings* on page 5-33.

5.21 Configuring SecurCore behavior if the processor clock stops when stepping instructions

If a step instruction operation fails, you must consider the configuration setting **No error if step-instr can't stop (NO_ERROR_ON_STEPRUN)**.

This configuration item controls the generation of error messages if a step instruction (`stepi`) operation fails because of a timeout attempting to stop the processor after a step is complete. This can occur on the SecurCore if an instruction execution results in the processor clock being disabled through **CLKEN**. The processor appears to be in a running state. If you use the default setting of `True`, no error appears if an instruction step results in the processor running. If you set the item to `False`, an error dialog appears in your debugger.

5.21.1 See also

Tasks

- *Configuring TrustZone enabled processor behavior when debug privileges are reduced* on page 5-39.

Reference

- *Debug hardware Advanced configuration reset options* on page 5-37.

5.22 Configuring TrustZone enabled processor behavior when debug privileges are reduced

The target does not allow invasive debug, that is when the processor enters debug state, while the execution environment is in the Secure World. Any attempt to do so might result in the following errors:

- Insufficient debug privilege to restore processor state for restart.
- Insufficient debug privilege to write software breakpoint to memory.

To suppress these errors, set the **Ignore debug privilege errors when starting core (IGNORE_START_DEBUG_PRIV_FAIL)** configuration item.

This option is useful if you are trying to debug an application in Normal World, while the current connection state of the target is in Secure World.

In this setup, the debug information and breakpoints are applied to the Normal World. However, the initial connection still happens while the target is in Secure World. The debug hardware unit has no control in this state, so debug control must be delayed until the target enters Normal World. Until that time, any errors arising from debug operations must be silently ignored.

5.22.1 See also

Tasks

- *Configuring SecurCore behavior if the processor clock stops when stepping instructions* on page 5-38.

Reference

- *Debug hardware Advanced configuration reset options* on page 5-37.

5.23 About platform detection and selection

Your debug hardware unit provides support for a number of development boards.

The following methods are available for platform detection and selection:

- Autodetection
- Manual selection.

You can also create your own platform files and add them to the list of available platforms.

Note

The platform file contains information on the scan chain of a platform at the time the file was saved. However, if you create a new scan chain configuration with the devices in a different order, then the saved platform file is not auto-detected. However, you can still manually select the platform configuration from the list of available platforms.

For more information on the various platforms supported, see also the Release Notes of the host software for your debug hardware unit.

5.23.1 See also

Tasks

- *Autodetecting a platform* on page 5-41
- *Manually selecting a platform* on page 5-43
- *Adding autoconfigure support for new platforms* on page 5-46.

5.24 Autodetecting a platform

The debug hardware autodetection feature checks to see if a platform configuration file is available that matches the configuration of your development platform. If a platform configuration is available, a the Select Platform dialog box is displayed. An example is shown in the following figure:

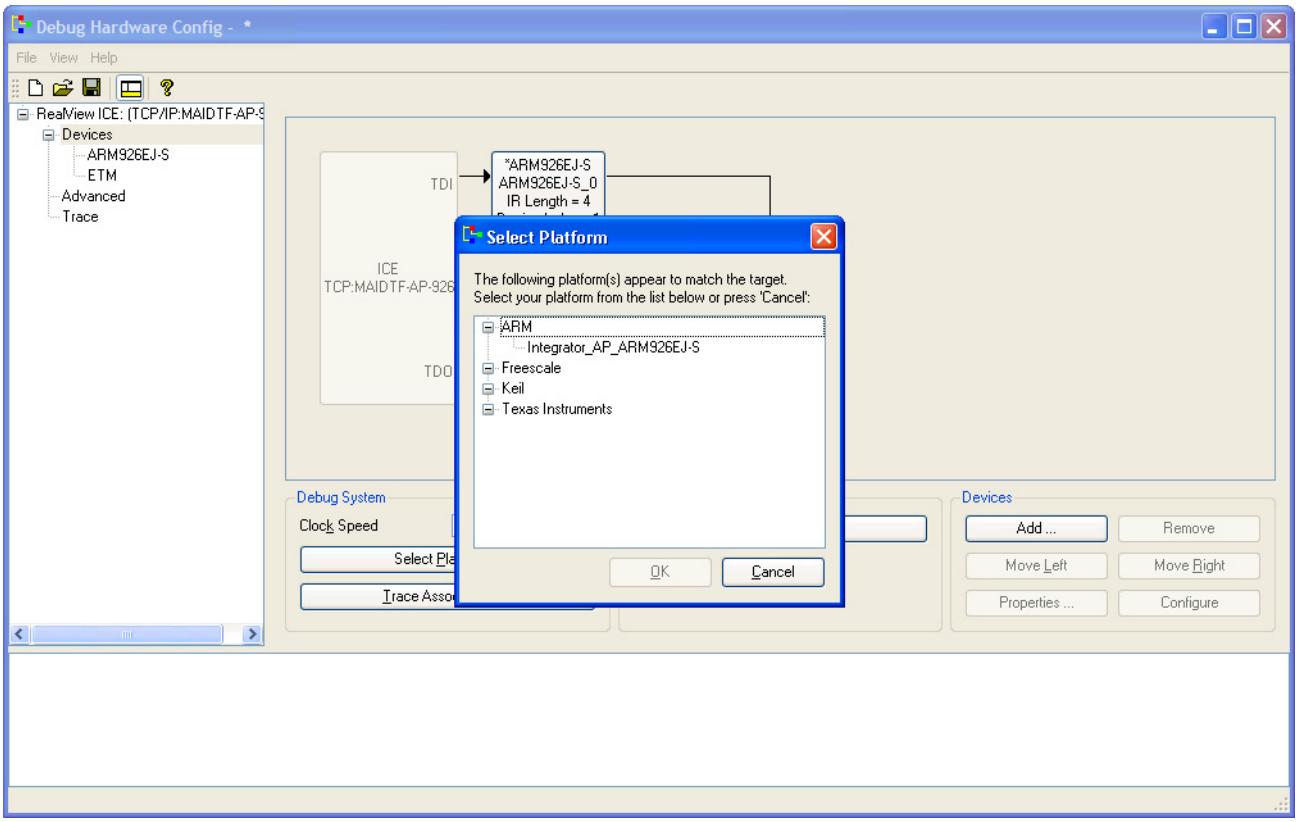


Figure 5-13 Automatic platform configuration

Select your platform from the list and click **OK**. This causes the entire platform configuration (that is, for the scan chain, Advanced settings and trace delays) to be loaded. The following figure shows an example:

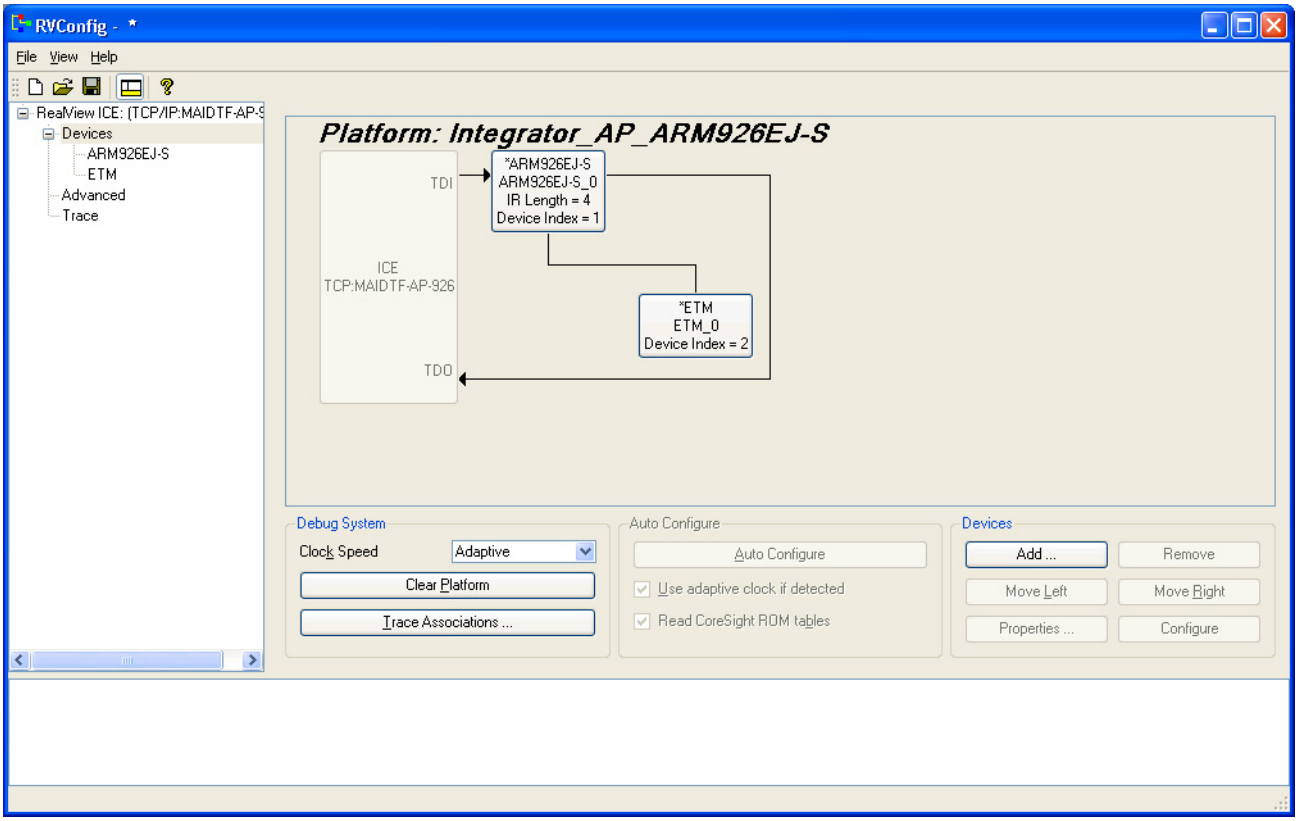


Figure 5-14 Platform configuration and identification

After the platform configuration is complete, the control pane shows the revised scan chain device/platform configuration in use and the name of the loaded platform. The tree diagram also reflects the new configuration.

Note

You cannot add, move, or remove any devices when a platform configuration is in use.

During the configuration process, the label on the **Select Platform...** button changes to read as **Clear Platform**.

5.24.1 See also

Tasks

- *Manually selecting a platform* on page 5-43
- *Clearing a platform assignment from a debug hardware configuration* on page 5-44
- *Adding new platforms* on page 5-45.

Concepts

- *About platform detection and selection* on page 5-40.

5.25 Manually selecting a platform

For platforms that cannot be detected automatically, you can perform a manual selection from a list of supported platforms. To do this, click the **Select Platform...** button from the Debug System pane of the Debug Hardware Config utility, and the Select Platform dialog box displays. The following figure shows an example:

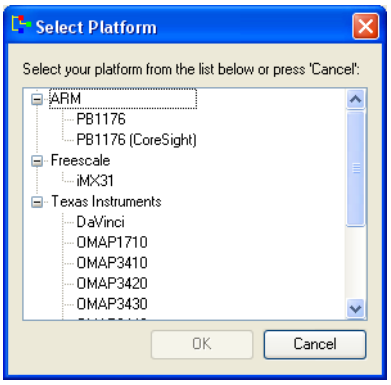


Figure 5-15 List of supported platforms

In the Select Platform dialog box, select your platform and click **OK**. This causes the entire platform configuration to be loaded, in a similar manner as when you are autodetecting a platform.

During the configuration process, the label on the **Select Platform...** button changes to read as **Clear Platform**.

You can also create your own platform files and add them to the list of available platforms.

5.25.1 See also

Tasks

- *Autodetecting a platform* on page 5-41
- *Clearing a platform assignment from a debug hardware configuration* on page 5-44
- *Adding new platforms* on page 5-45
- *Adding autoconfigure support for new platforms* on page 5-46.

Concepts

- *About platform detection and selection* on page 5-40.

5.26 Clearing a platform assignment from a debug hardware configuration

You might want to clear a platform assignment from a debug hardware configuration because you want to reconfigure the scan chain for your development platform.

Note

This also clears the scan chain configuration. As an alternative, you might want to create a new debug hardware configuration.

To clear a platform assignment:

1. Open the required debug hardware configuration file in Debug Hardware Config
2. Click **Clear Platform**.

   The scan chain configuration is deleted.
3. Either autoconfigure the scan chain or manually add devices to the scan chain as required.

5.26.1 See also

Tasks

- *Adding new platforms* on page 5-45
- *Autoconfiguring a scan chain* on page 5-11
- *Adding devices to the scan chain* on page 5-12.

5.27 Adding new platforms

Debug Hardware Config is preconfigured to support a number of platforms, but you can add support for more platforms by creating your own platform files. You can also provide a name and the manufacturer details for the new platform.

To create a new configuration file, you must:

1. Configure a scan chain for the new platform.
2. Make any changes to the device settings that are required.
3. Make any changes to the advanced settings that are required.
4. Make any changes to the trace settings that are required.
5. Specify a new platform name and its manufacturer in the Export As Platform dialog box. To do this, select **Export platform file...** from the **File** menu.
6. In the Export As Platform dialog box, enter the new name and manufacturer details of the platform in the **Board Name** and **Manufacturer** fields, respectively. The following figure shows an example:

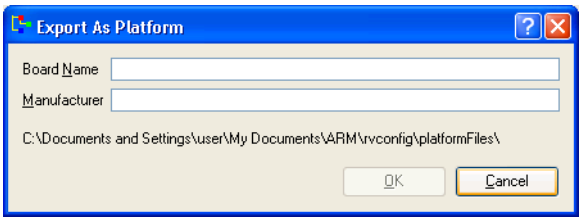


Figure 5-16 Export As platform dialog box

   The name and manufacturer details of the platform are used to name the new platform files, and are displayed in the Select Platform dialog box for the new platform.

   On Windows, the `platformFiles` directory, shown in the figure above, is located in `My Documents\ARM\rvconfig\platformFiles`. In Linux, it is located in `~/.rvconfig/platformFiles`

7. Click **OK**.

5.27.1 See also

Tasks

- *Configuring a JTAG scan chain* on page 5-7
- *About configuring a device list* on page 5-9
- *Autodetecting a platform* on page 5-41
- *Manually selecting a platform* on page 5-43
- *Adding autoconfigure support for new platforms* on page 5-46
- *Configuring the debug hardware Advanced settings* on page 5-47
- *Configuring trace lines (DSTREAM and RVT2 only)* on page 7-4.

Concepts

- *About platform detection and selection* on page 5-40.

5.28 Adding autoconfigure support for new platforms

When adding platforms to the Select Platform dialog box, a platform detection file `.det` is created automatically after you have setup the board name and manufacturer. You can, however, add more `.det` files to the `platformFiles` directory, which allow different hardware versions to be recognized when you click the **Auto Configure** button. A `.det` file can contain information for one or more platforms. The platform description consists of a JTAG ID code and mask for each device on the JTAG scan chain of the target platform, and the name of the associated debug hardware configuration file `.rvc`.

For example:

```
0x0B73B02F,0xFFFFFFFF,  0x07926001,0xFFFFFFFF,  0,0,  0x2B900F0F,0xFFFFFFFF =
mycompany_eg1.rvc
```

In the above example, the platform is expected to contain four devices on its scan chain. The first device must have a JTAG ID code of `0x0B73B02F`, the second `0x07926001`, the third device can have any ID code, and the code of the fourth device must be `0x2B900F0F`.

Several `.det` files can be supplied, and each file can contain more than one line. For example:

```
0x22193024, 0xEFFFFFFF, 0,0, 0,0 = mycompany_eg2.rvc
0x08210024, 0xFFFFFFFF, 0,0, 0,0 = mycompany_eg2.rvc
0x05310024, 0xFFFFFFFF, 0,0, 0,0 = mycompany_eg3.rvc
```

In this example, a scan chain containing three devices, where the first device has an ID code of `0x22193024`, `0x32193024` or `0x08210024` is associated with `mycompany_eg2.rvc`. The mask value of `0xEFFFFFFF` means that both `0x22193024` and `0x32193024` are identified. If there are three devices, and the first has an ID code of `0x05310024`, then it is associated with `mycompany_eg3.rvc`.

When you click the **Auto Configure** button, the detected scan chain is compared against all the platforms described by `.det` files. If the connected target matches any of these platforms, a Select Platform dialog box is displayed, and asks the user to confirm that the platform has been correctly detected.

It is possible to create platform information that associates a given scan chain with several `.rvc` files. If this happens, the Select Platform dialog box lists all the platforms that match, and you will be asked to select the platform that matches your hardware.

5.28.1 See also

Tasks

- *Adding new platforms* on page 5-45.

5.29 Configuring the debug hardware Advanced settings

The Advanced settings enable you to change the global configuration settings for the debug hardware unit. Such settings include debug connection mode settings and reset settings.

To configure the Advanced settings:

1. Open the Debug Hardware Config utility.
2. Either:
   - connect to and configure a debug hardware connection to create a new configuration file
   - open an existing debug configuration file.
3. Select the Advanced settings group in the tree control to display the settings. The following figure shows an example:

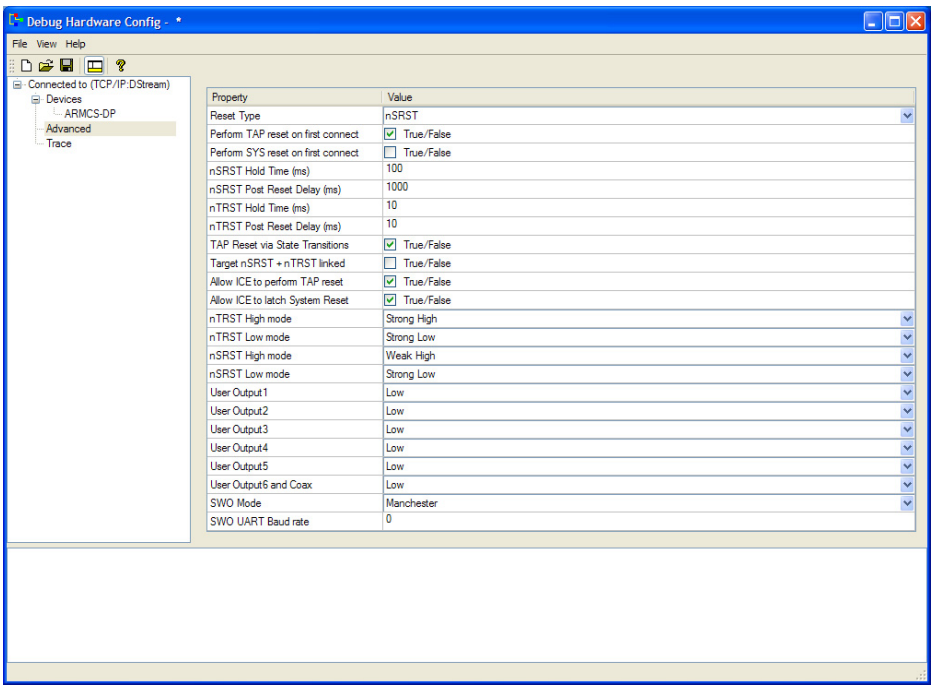


Figure 5-17 Displaying the advanced controls

4. Make the changes to the settings as required.
5. Select **Save** from the **File** menu to save your changes.
6. Select **Exit** from the **File** menu to close the Debug Hardware Config utility.

Note

These settings are sent to a debug hardware unit whenever you connect to the unit from a debugger. They are used as the default reset behavior for all target hardware that you debug with that debug hardware unit.

5.29.1 See also

Tasks

- *Debugging with a reset register* on page 8-20

- Chapter 4 *Managing the firmware on your debug hardware unit*.

Reference

- *Debug hardware Advanced configuration settings* on page 5-33.

ARM® DSTREAM™ System and Interface Design Reference:

- Reset signals, `../com.arm.doc.dui0499e/CHDHFJEH.html`
- Serial Wire Debug, `../com.arm.doc.dui0499e/BEHIADEG.html`.

ARM RVI and RVT System and Interface Design Reference:

- Reset signals, `../com.arm.doc.dui0517e/CHDHFJEH.html`
- Serial Wire Debug, `../com.arm.doc.dui0517e/BEHIADEG.html`.

5.30 Saving your changes

To save any changes that you have made to a configuration in the Debug Hardware Config utility, select **Save** from the **File** menu.

Changes are stored in a debug hardware configuration file, `*.rvc`. See your debugger documentation for the location of this file. There can be a number of `*.rvc` files in this location, and these are named with respect to the connection name.

You can change the location of the `*.rvc` file , or save multiple copies of the file for different target configurations.

5.30.1 See also

Tasks

- *Creating a debug hardware configuration file* on page 5-4
- *Opening an existing debug hardware configuration file in Debug Hardware Config* on page 5-6
- *Disconnecting from a debug hardware unit* on page 5-50.

5.31 Disconnecting from a debug hardware unit

You might want to disconnect from a debug hardware unit if you want to connect to another debug hardware unit.

To disconnect from a debug hardware unit:

1. Select the debug hardware node in the tree diagram to display the Debug Hardware Config utility. The following figure shows an example:

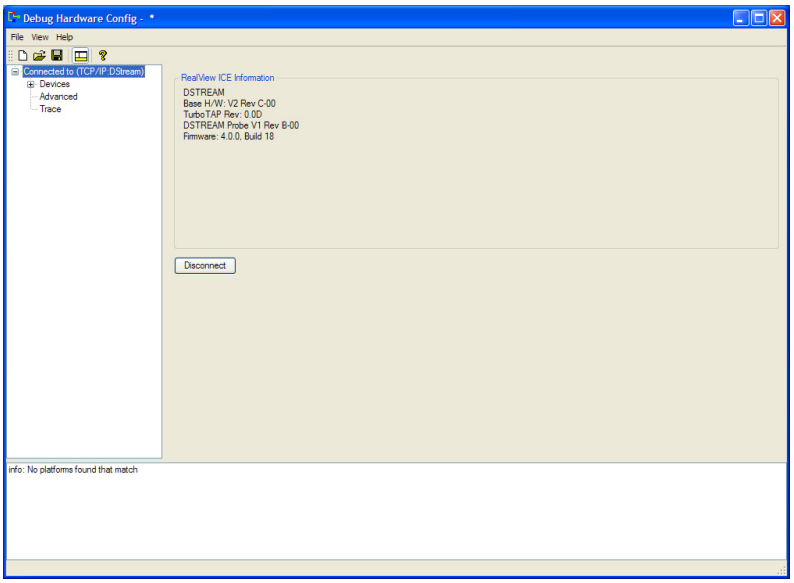


Figure 5-18 Displaying the connection controls

2. Click **Disconnect**.

   If you have unsaved configuration changes, a warning dialog box appears. The following figure shows an example:

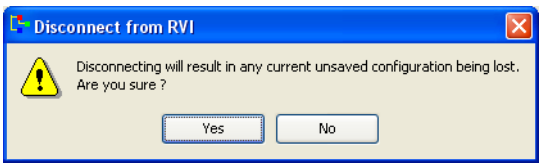


Figure 5-19 Warning when disconnecting with unsaved configuration changes

3. In this warning dialog box:
   - Click **Yes** to disconnect, losing any unsaved configuration data.
   - Click **No** to remain connected. Save your changes, then disconnect.

5.31.1 See also

Tasks

- *Connecting to a debug hardware unit* on page 2-7
- *Creating a debug hardware configuration file* on page 5-4
- *Saving your changes* on page 5-49.

5.32 Configuring a target processor for virtual Ethernet

For applications on your target to communicate over your network, you must configure virtual Ethernet on the target processor. This is required if the target does not have its own Ethernet hardware, or if its drivers have not yet been written.

Note

The network settings are supported only on ARM7, ARM9, ARM11, and ARM SecurCore processors using JTAG. They are not supported on CoreSight systems.

To configure virtual Ethernet for a target processor:

1. Open the Debug Hardware Config utility.
2. Either:
   - connect to and configure a debug hardware connection to create a new debug hardware configuration file
   - open an existing debug hardware configuration file.
3. Connect to the required debug hardware unit.
4. Select the **Devices** node in the tree diagram.
5. Select the processor that want to configured for virtual Ethernet.
6. Set the network settings as required. The following table shows an example:

Table 5-1 Items for configuring static IP addresses

Device setting	Example
IP Address	`110.35.3.21`
Network Mask	`255.255.255.0`
Default Gateway	`110.35.3.254`

7. Select **Save** from the **File** menu to save the changes.
8. Select **Exit** from the **File** menu to close the Debug Hardware Config utility.

5.32.1 See also

Tasks

- *Creating a debug hardware configuration file* on page 5-4
- *Connecting to a debug hardware unit* on page 2-7
- *Disconnecting from a debug hardware unit* on page 5-50.

Reference

- *Debug hardware device configuration settings* on page 5-26.

5.33 CoreSight device names and classes

The following table shows the name and class of all CoreSight devices that are supported by the debug hardware:

Table 5-2 CoreSight device names and classes

Device name	Description	Device class
ARMCS-DP	ARM CoreSight debug port (supports both JTAG-DP and SW-DP)	Debug port
CSETB	CoreSight Embedded Trace Buffer (ETB)	Trace sink
CSETM	CoreSight Embedded Trace Macrocell (ETM)	Trace source
CSETM11	CoreSight Embedded Trace Macrocell designed for ARM11 (ETM11)	Trace source
CSTFUNNEL	CoreSight Trace Funnel	Link
CSTPIU	CoreSight Trace Port Interface Unit	Trace sink
CSHTM32	CoreSight AHB Trace Macrocell (HTM32)	Trace source
CSHTM64	CoreSight AHB Trace Macrocell (HTM64)	Trace source
CSITM	CoreSight Instrumentation Trace Macrocell (ITM)	Trace source
CSPTM	CoreSight Program Trace Macrocell (ITM)	Trace source
CSSWO	CoreSight Serial Wire Output (SWO)	Trace sink
ARM1136JFS-JTAG-AP	ARM1136JF-S processor connected using JTAG-AP	Core
ARM1156T2FS-JTAG-AP	ARM1156T2FS processor connected using JTAG-AP	Core
ARM1176JZF-JTAG-AP	ARM1156T2FS processor connected using JTAG-AP	Core
Cortex-M0	Cortex-M0 processor	Core
Cortex-M1	Cortex-M1 processor	Core
Cortex-M3	Cortex-M3 processor	Core
Cortex-R4	Cortex-R4 processor	Core
Cortex-A5	Cortex-A5 processor	Core
Cortex-A8	Cortex-A8 processor	Core
Cortex-A9	Cortex-A9 processor	Core

The device names are the names used in debug hardware configurations and trace association files.

5.33.1 See also

Concepts

- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *Format of trace associations* on page 6-8.

Chapter 6
Configuring CoreSight systems

The following topics describe CoreSight systems how to use Debug Hardware Config to configure them:

Tasks

Concepts

Reference

6.1 About CoreSight system configuration

CoreSight systems consist of a *Debug Access Port* (DAP) that comprises the following components:

- A *debug port* (DP) that connects to the scan chain or *Serial Wire Debug* (SWD) interface, and that provides the system interface to debug hardware.
- Either *Advanced High-performance Bus Access Port* (AHB-AP for AHB access) or *ARM Peripheral Bus Access Port* (APB-AP for APB access).

Debug components are attached to the buses, and are accessed through the APs on the DAP. CoreSight debug components are configured with the index of the AP to which they are attached, and the base address on the bus.

The DAP can also contain a JTAG-AP that enables the connection of JTAG devices on internal scan chains, for example ARM11 processors. JTAG devices must be configured with the AP index of the JTAG-AP, the JTAG port on the AP, and the pre-bits and post-bits for both IR and DR scans for the particular device.

CoreSight components are associated with a DAP, so they are placed on the scan chain (or SWD connection) after the DAP with which they are associated. The DAP is represented on the scan chain by the ARMCS-DP device. CoreSight components are not located on the JTAG scan chain, so they must be placed between the DAP and the next JTAG device.

6.1.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Setting up a CoreSight trace association file* on page 6-11.

Concepts

- *Reading the CoreSight ROM table* on page 6-3
- *CoreSight autodetection* on page 6-4
- *How the debug hardware unit autodetects Serial Wire Debug* on page 6-5.

6.2 Reading the CoreSight ROM table

If the target system contains a valid CoreSight ROM table, you can use this to configure the AHB-AP and APB-AP devices behind the *Debug Access Port* (DAP).

To configure CoreSight devices using a CoreSight ROM table:

- If you are manually adding devices to the scan chain:
  1. Add the ARMCS-DP device.
  2. Right-click on the ARMCS-DP device, then select the **Read CoreSight ROM Table** option.
- If you are autoconfiguring scan chain, ensure that the **Read CoreSight ROM Tables** checkbox is selected before you auto configure. The checkbox is selected by default.

6.2.1 See also

Tasks

- *Setting up a CoreSight trace association file* on page 6-11
- *Loading a trace association file* on page 6-13

Concepts

- *About CoreSight system configuration* on page 6-2
- *CoreSight autodetection* on page 6-4
- *How the debug hardware unit autodetects Serial Wire Debug* on page 6-5.

6.3 CoreSight autodetection

Autodetecting a CoreSight system adds the ARMCS-DP device. The device represents the *Debug Access Port* (DAP) in a CoreSight system. If the system contains a valid CoreSight ROM table, the devices listed in that table are also added by default.

The **Read CoreSight ROM Table** checkbox enables you to enable or disable the automatic reading of the table during an autoconfigration. The following figure shows the location of this checkbox:

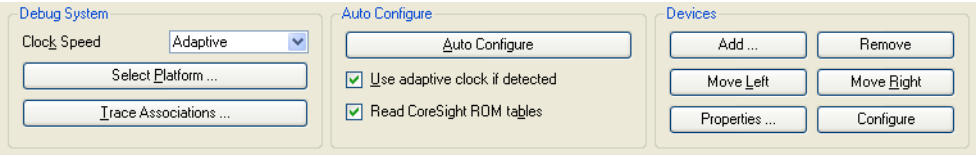


Figure 6-1 Read CoreSight ROM Table option

You can also force a read of the CoreSight ROM Table as follows:

1. Right-click on the ARMCS-DP device.
2. Select **Read CoreSight ROM tables** from the context menu.

Note

If you uncheck the **Read CoreSight ROM Table** checkbox, or the table read fails, you can manually add the CoreSight devices to the scan chain.

6.3.1 See also

Tasks

- *Setting up a CoreSight trace association file* on page 6-11
- *Reading the CoreSight ROM table* on page 6-3

Concepts

- *About configuring a device list* on page 5-9
- *About CoreSight system configuration* on page 6-2
- *How the debug hardware unit autodetects Serial Wire Debug* on page 6-5.

6.4 How the debug hardware unit autodetects Serial Wire Debug

Serial Wire Debug (SWD) does not support a scan chain in the same way that JTAG does. When autoconfiguring in SWD mode, the debug hardware unit:

1. Adds the ARMCS-DP device to the scan chain configuration.
2. Optionally reads the CoreSight ROM Table to add the remaining devices. This is done by default, but you can override this.

Note

Be aware that if your target supports both JTAG and SWD, then you must enable **Use SWJ Switching** in the Advanced configuration settings before autoconiguring the scan chain.

6.4.1 See also

Tasks

- *Setting up a CoreSight trace association file* on page 6-11
- *Reading the CoreSight ROM table* on page 6-3.

Concepts

- *About CoreSight system configuration* on page 6-2
- *CoreSight autodetection* on page 6-4.

Reference

- *Debug hardware Advanced configuration settings* on page 5-33.

6.5 About trace associations

Trace associations are stored in the debug hardware configuration file (`.rvc`). However, you can save the associations in a separate file, if you want to provide the associations to other users using a similar development platform.

Trace associations are required to describe the associations between the processors and the trace-related devices in complex CoreSight systems. Examples of such associations are:

- processor outputs into an ETM
- ETM outputs into an ETB
- ETM input from a particular processor
- CoreSight ETM outputs into a *Trace Port Interface Unit* (TPIU)
- TPIU input from a particular trace source
- ETB input from a particular trace source.

Your debug tools use the associations in the debug hardware configuration file to determine the components that must be programmed for tracing a specific source in the system

Associations have direction, and some are bi-directional. You must configure associations correctly to enable your debugger to associate trace output with the source of generated trace.

Note

Although a trace association might be created for a non-CoreSight system, do not modify it.

6.5.1 See also

Concepts

- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Trace Association Editor dialog box* on page 6-9
- *Format of trace associations* on page 6-8.

6.6 Defining CoreSight trace associations

You can define CoreSight trace associations:

By order CoreSight components are accessible through a *Debug Access Port* (DAP). The template used to access the DAP is called the *ARM CoreSight Debug Port* (ARMCS-DP). Any CoreSight components that follow the ARMCS-DP are associated with it. If there are multiple DAPs, the devices must be ordered this way:

1. ARMCS-DP
2. associated CoreSight devices
3. ARMCS-DP
4. associated devices.

Note

The order of the devices listed in the trace association file must match the order of devices as defined in the Debug Hardware Config utility.

By input and output attributes

Components have inputs and outputs, and when you define these connections you define the trace flow through a system.

6.6.1 See also

Concepts

- *About trace associations* on page 6-6
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Trace Association Editor dialog box* on page 6-9
- *Format of trace associations* on page 6-8.

6.7 Format of trace associations

Trace associations consist of a number of distinct elements separated by semi-colons. The following table lists the elements of an association:

Table 6-1 Trace association file element names and descriptions

Element name	Description
`Name`	A unique name for the component in this list.
`Type`	A type identifier for the specified component. This must match the name of the template for the component.
`Port`*n*	Used only for components that are trace sinks. Indicates that this component can get input from the component specified as connected to it using a Port. When specifying a Port, each Port element tag must be suffixed with a number starting at 0, for example "`Port0=Cortex-R4;Port1=Cortex-A8;`".
`TraceOutput`*n*	Used only for components that are trace sources. Indicates that this component can output into the component specified as a TraceOutput. Where more than one TraceOutput must be specified, each TraceOutput element tag must be suffixed with a number starting at 0, for example "`TraceOutput0=ETB;TraceOutput1=TPIU;`".
`Core`	Used to link a component to a processor.
`ETM`	Used to link a component to an ETM.

6.7.1 Example trace association file

The following example is an association file for a system containing a Cortex-R4 processor, a CoreSight TPIU, a CoreSight ETM, a CoreSight ETB, and a CoreSight DAP:

```
Name=ARMCS-DP;Type=ARMCS-DP;Name=Cortex-R4;Type=Cortex-R4;ETM=ETMR4;Name=ETMR4;Type=CSE
TM;TraceOutput0=TPIU;TraceOutput1=ETB;Core=Cortex-R4;Name=ETB;Type=CSETB;Port0=ETMR4;Na
me=TPIU;Type=CSTPIU;Port0=ETMR4;
```

Note

Although the format of an association file takes the form shown in this example, you must use the **Trace Associations...** button to create your associations.

6.7.2 See also

Concepts

- *About trace associations* on page 6-6
- *Setting up a CoreSight trace association file* on page 6-11
- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Trace Association Editor dialog box* on page 6-9.

6.8 Trace Association Editor dialog box

The Trace Association Editor dialog box enables you to set up device associations for a CoreSight system. All associations are shown expanded by default. The following figure shows an example:

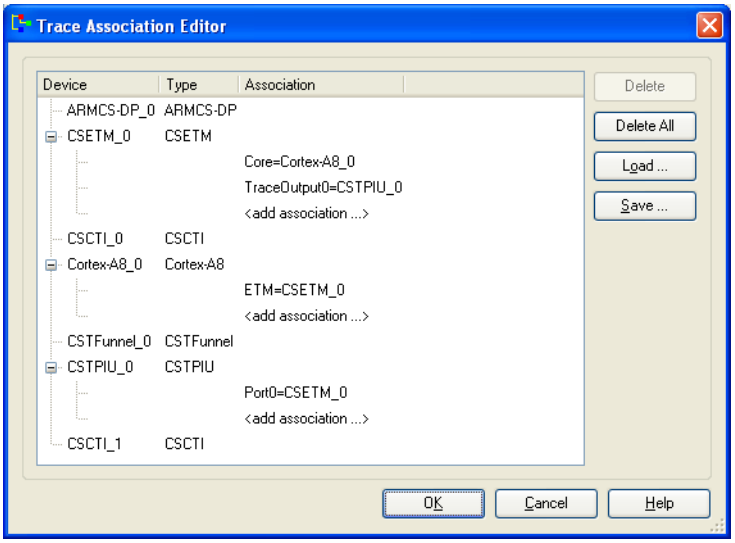


Figure 6-2 Trace Association Editor dialog box

The devices shown in the Trace Association Editor dialog box reflect the order of those in the scan chain. To expand the details for a device, either double-click on the device name, or click the + button for the device.

To assign an association to a device:

1. Double-click **<add association...>** for the device to display the Edit Association dialog box. The following figure shows an example:

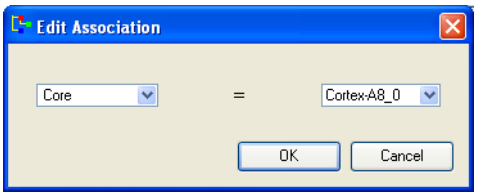


Figure 6-3 Edit Association dialog box

2. Select the required device to be used for the association from the drop-down list on the left of the dialog box.
3. Type a description of the association in the field on the right.
4. Click **OK**.

To change any typed text, double-click on an association description, and edit the text in the Edit Association dialog.

To delete an association description, select the association and click **Delete**.

To clear all associations created, click **Delete All**.

To save your associations to a file:

1. Click the **Save..**. button. The Save RealView Associations File... dialog box displays.

2. Locate an appropriate directory to save your file.
3. Click **Save**.

To load an associations file:

1. Click the **Load...** button. The Load RealView Associations File... dialog box displays.
2. Locate the directory containing your CoreSight associations files.
3. Select the appropriate `.txt` file.
4. Click **Open**.

To return to the Debug Hardware Config utility main window, click **OK**.

6.8.1 See also

Tasks

- *Setting up a CoreSight trace association file* on page 6-11.

Concepts

- *About CoreSight system configuration* on page 6-2.

6.9 Setting up a CoreSight trace association file

CoreSight systems can contain many trace sources and sinks. To enable your debugger to capture trace correctly from a system, and to associate the trace information with the source that generated it, you must use a CoreSight trace association file.

To set up a CoreSight trace association file:

1. Open the Debug Hardware Config utility.
2. Either:
   - connect to and configure a debug hardware connection to create a new debug hardware configuration file
   - open an existing debug hardware configuration file.
3. Click the **Trace Associations...** button to display the Trace Association Editor dialog box. The following figure shows an example:

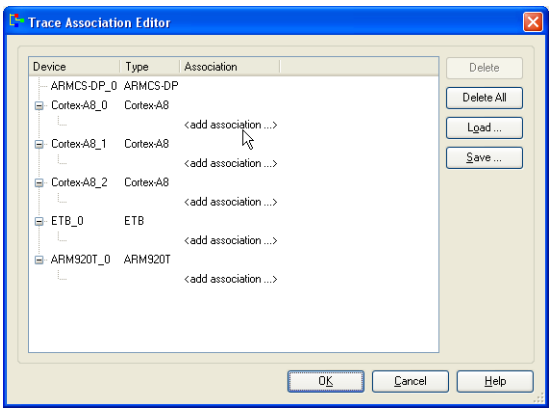


Figure 6-4 Trace Association Editor dialog box

   The devices shown reflect the order of those in the scan chain.
4. To expand the details for a device, double-click on the device name.
5. To add a new association for a device, double-click on **<add association...>** for that device to display the Edit Association dialog box.
6. Set the required associations from each of the available drop-down menus. The following figure shows an example:

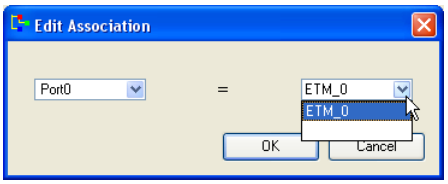


Figure 6-5 Edit Association dialog box

7. Click **OK** to return to the Trace Association Editor dialog box.
8. Assign associations for other devices as required:
   - To change any associations listed under the Association column, double-click on that association to display the Edit Association dialog box, and make the required changes.

     Click **OK** to return to the Trace Association Editor dialog box.

- To delete a single association, select the association and click **Delete**.
  You are not asked to confirm the deletion.
- To delete all the associations in the system, click **Delete All**.
  You are not asked to confirm the deletion.

9. Save your trace associations file:
   a. Click **Save...** to display the Save RealView Associations File... dialog box.
   b. Locate the directory where you want to save the file.
   c. Click **Save**.

   You might want to save a Trace Association file if you want to provide the associations with other users using a similar development platform.

10. Click **OK** to return to the Debug Hardware Config utility main window.

    An asterisk in a device box of the schematic diagram indicates that the device is associated with another device. The following figure shows that the CSETM and the Cortex-A8 devices have associations:

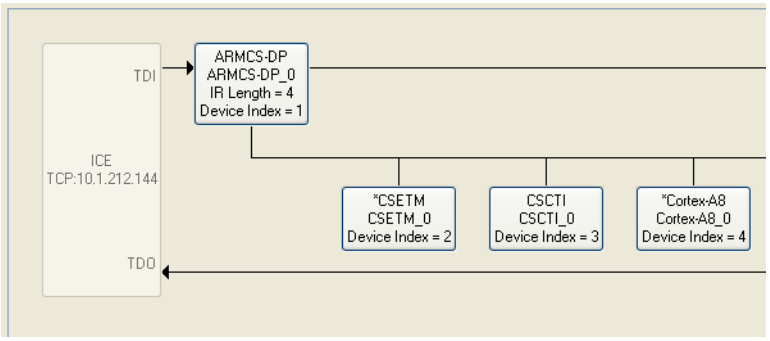


Figure 6-6 Devices with associations

6.9.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Loading a trace association file* on page 6-13
- *Reading the CoreSight ROM table* on page 6-3
- *CoreSight autodetection* on page 6-4.

Concepts

- *About CoreSight system configuration* on page 6-2
- *How the debug hardware unit autodetects Serial Wire Debug* on page 6-5
- *About trace associations* on page 6-6.

6.10 Loading a trace association file

To load a CoreSight trace association file:

1. Open the Debug Hardware Config utility.
2. Either:
   - connect to and configure a debug hardware connection to create a new debug hardware configuration file
   - open an existing debug hardware configuration file.
3. Click the **Trace Associations...** button to display the Trace Association Editor dialog box. The following figure shows an example:

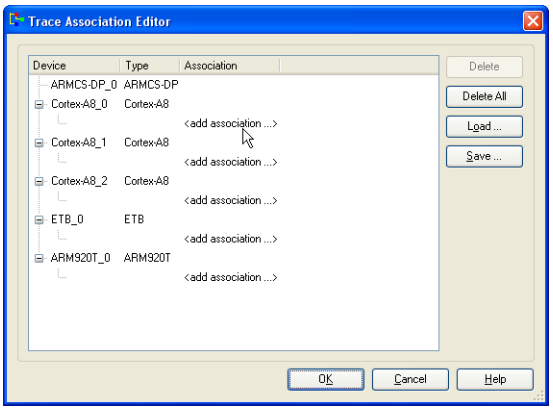


Figure 6-7 Trace Association Editor dialog box

The devices shown reflect the order of those in the scan chain.

4. Click **Load...** to display the Load RealView Associations File... dialog box.
5. Locate the directory where you saved your trace association file.
6. Click **Open** to load the file and return to the Trace Association Editor dialog box.
7. Click **OK** to return to the Debug Hardware Config utility main window.

   An asterisk in a device box of the schematic diagram indicates that the device is associated with another device. The following figure shows that the CSETM and the Cortex-A8 devices have associations:

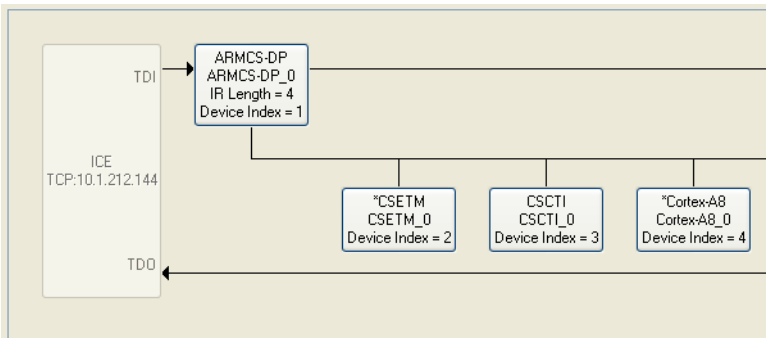


Figure 6-8 Devices with associations

6.10.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Setting up a CoreSight trace association file* on page 6-11
- *Reading the CoreSight ROM table* on page 6-3
- *CoreSight autodetection* on page 6-4.

Concepts

- *About CoreSight system configuration* on page 6-2
- *How the debug hardware unit autodetects Serial Wire Debug* on page 6-5
- *About trace associations* on page 6-6.

6.11 CoreSight topology and associations for the CoreSight DK11

The following figure shows the CoreSight topology diagram for CoreSight DK11:

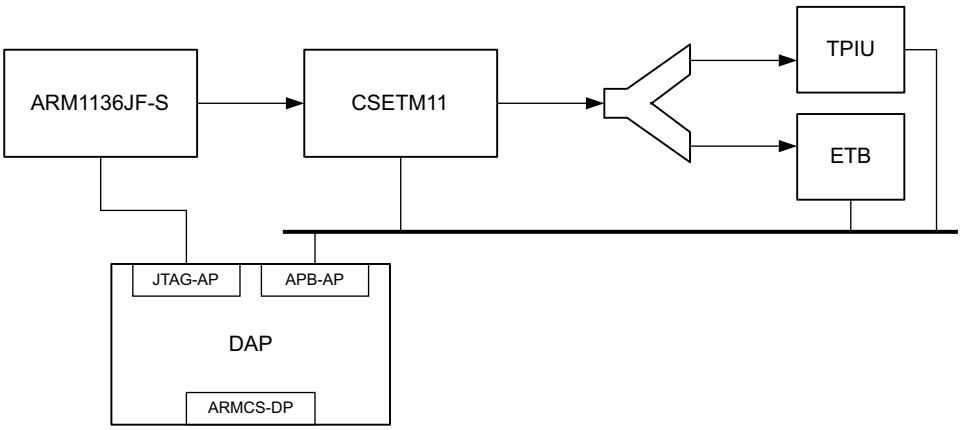


Figure 6-9 CoreSight system topology diagram - CoreSight DK11

The Association file for this is:

```
Name=ARMCS-DP;Type=ARMCS-DP;Name=ARM1136JFS-JTAG-AP;Type=ARM1136JFS-JTAG-AP;ETM=ETM11;N
ame=ETM11;Type=CSETM11;TraceOutput0=ETB;TraceOutput1=TPIU;Core=ARM1136JFS-JTAG-AP;Name=
ETB;Type=CSETB;Port0=ETM11;Name=TPIU;Type=CSTPIU;Port0=ETM11;
```

In this Association file:

Name=ARMCS-DP;Type=ARMCS-DP;

This line specifies the first device in the list is the ARM CoreSight Debug port. Any CoreSight components that are connected by the Debug Port associated with this template must follow this device.

Name=ARM1136JFS-JTAG-AP;Type=ARM1136JFS-JTAG-AP;ETM=ETM;

This line specifies that an ARM1136JF-S processor is connected to a JTAG-AP on the preceding ARMCS-DP. The `ETM=ETM11` section states that the processor has an associated ETM called ETM11.

Name=ETM11;Type=CSETM11;TraceOutput0=ETB;TraceOutput1=TPIU;Core=ARM 1136JFS-JTAG-AP;

This line specifies that an ETM is accessible using the preceding ARMCS-DP.

`TraceOutput0=ETB` signifies that this ETM can output into the component named ETB.

`TraceOutput1=TPIU` signifies that this ETM can output into the component named *Trace Port Interface Unit* (TPIU).

`Core=ARM1136JFS-JTAG-AP` signifies that the source for trace captured by this ETM is the ARM1136JFS-JTAG-AP device.

Name=ETB;Type=CSETB;Port0=ETM11;

This line specifies that a CoreSight ETB is accessible using the preceding ARMCS-DP.

`Port0=ETM11;` indicates that the source of trace that is stored in this ETB is the component named ETM11.

Name=TPIU;Type=CSTPIU;Port0=ETM;

This line specifies that a CoreSight TPIU is accessible using the preceding ARMCS-DP.

`Port0=ETM` indicates that the source of trace that is stored in this ETB is the component named ETM11.

Note

Although the funnel is located between the trace sources and trace sinks, it is not necessary to include it in the Association mapping. The funnel can be used to control the flow of trace through the system dynamically. The purpose of the Association file is to describe a static view of the trace flow through the system.

6.11.1 See also

Concepts

- *Setting up a CoreSight trace association file* on page 6-11
- *About trace associations* on page 6-6
- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Format of trace associations* on page 6-8
- *Trace Association Editor dialog box* on page 6-9.

6.12 CoreSight topology and associations for the Cortex-R4 FPGA

The following figure shows the CoreSight topology diagram for Cortex-R4 FPGA:

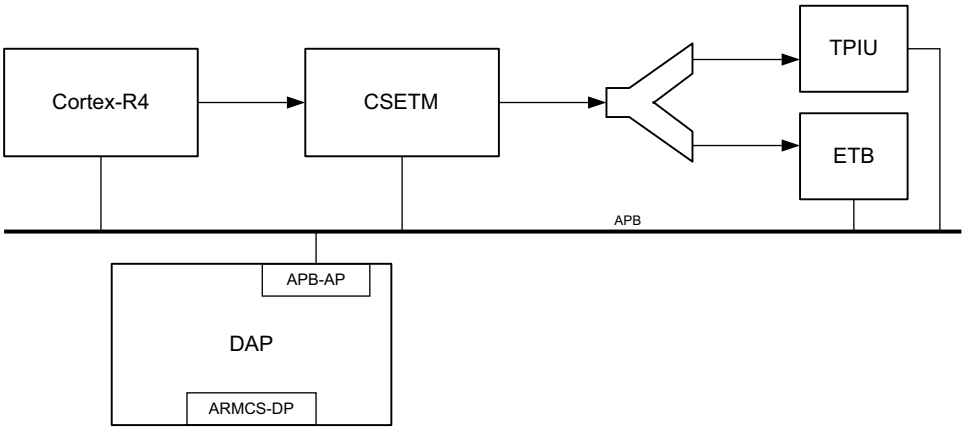


Figure 6-10 CoreSight system topology diagram - Cortex-R4 FPGA

The Association file for this is:

```
Name=ARMCS-DP;Type=ARMCS-DP;Name=Cortex-R4;Type=Cortex-R4;ETM=ETMR4;Name=ETMR4;Type=CSE
TM;TraceOutput0=TPIU;TraceOutput1=ETB;Core=Cortex-R4;Name=ETB;Type=CSETB;Port0=ETMR4;Na
me=TPIU;Type=CSTPIU;Port0=ETMR4;
```

In this Association file:

Name=ARMCS-DP;Type=ARMCS-DP;

This line specifies the first device in our list is the ARM CoreSight Debug port. Any CoreSight components that are connected using the Debug Port associated with this template must follow this device.

Name=Cortex-R4;Type=Cortex-R4;ETM=ETMR4;

This line specifies that a Cortex-R4 processor is connected using the preceding ARMCS-DP. The `ETM=ETMR4` section states that the processor has an associated ETM called ETMR4.

Name=ETMR4;Type=CSETM;TraceOutput0=TPIU;TraceOutput1=ETB;Core=Cortex-R4;

This line specifies that an ETM is accessible using the preceding ARMCS-DP.

`TraceOutput0=TPIU` signifies that this ETM can output into the component named *Trace Port Interface Unit* (TPIU).

`TraceOutput1=ETB` signifies that this ETM can output into the component named ETB.

`Core=Cortex-R4` signifies that the source for trace captured by this ETM is the Cortex-R4 device.

Name=ETB;Type=CSETB;Port0=ETMR4;

This line specifies that a CoreSight ETB is accessible using the preceding ARMCS-DP.

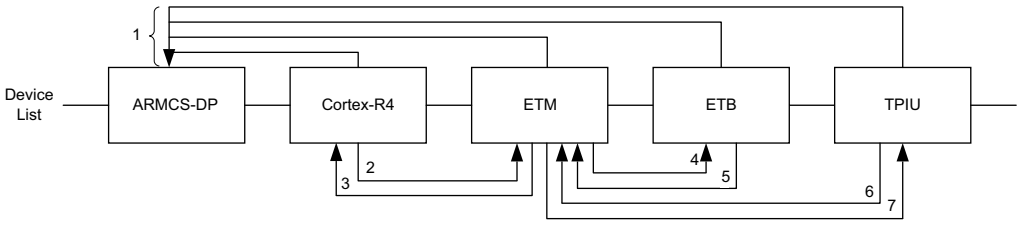
`Port0=ETMR4;` indicates that the source of trace that is stored in this ETB is the component named ETMR4.

Name=TPIU;Type=CSTPIU;Port0=ETMR4;

This line specifies that a CoreSight TPIU is accessible using the preceding ARMCS_DP.

`Port0=ETMR4;` indicates that the source of trace that is routed through this TPIU is the component named ETMR4.

The following figure shows the Cortex-R4 FPGA Associations:



1. Associations defined by order of devices
2. Association defined by ETM=ETM element for Cortex-R4
3. Association defined by Core=Cortex-R4
4. Association defined by TraceOutput2=ETB
5. Association defined by TraceInput=ETM
6. Association defined by TraceInput=ETM
7. Association defined by TraceOutput1=TPIU

Figure 6-11 Cortex-R4 FPGA Associations

6.12.1 See also

Concepts

- *Setting up a CoreSight trace association file* on page 6-11
- *About trace associations* on page 6-6
- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Format of trace associations* on page 6-8
- *Trace Association Editor dialog box* on page 6-9.

6.13 CoreSight topology and associations for the Cortex-M3 FPGA

The following figure shows the CoreSight topology diagram for Cortex-M3 FPGA:

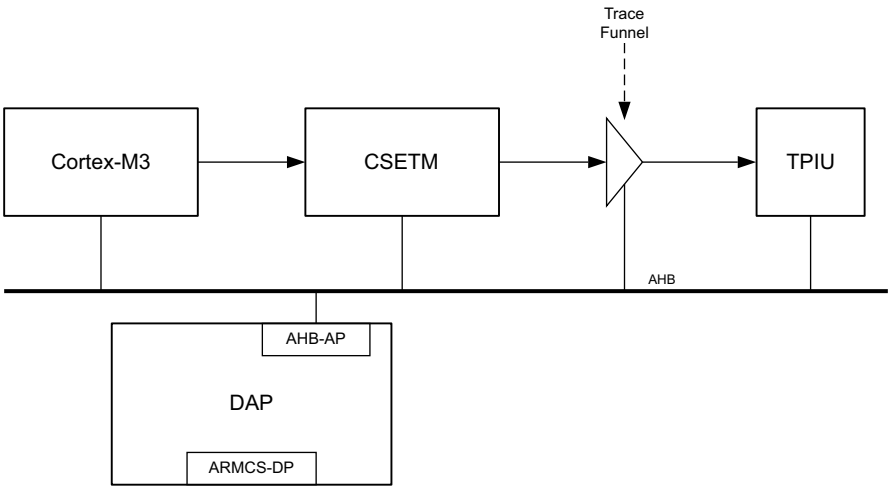


Figure 6-12 CoreSight system topology diagram - Cortex-M3 FPGA

The Association file for this is:

```
Name=ARMCS-DP;Type=ARMCS-DP;Name=Cortex-M3;Type=Cortex-M3;ETM=ETMM3;Name=ETMM3;Type=CSE
TM;Core=Cortex-M3;TraceOutput=TPIU;Name=Funnel;Type=CSTFunnel;Name=TPIU;Type=CSTPIU;Por
t0=ETMM3;
```

In this Association file:

Name=ARMCS-DP;Type=ARMCS-DP;

This line specifies the first device in our list is the ARM CoreSight Debug port. Any CoreSight components that are connected using the Debug Port associated with this template must follow this device.

Name=Cortex-M3;Type=Cortex-M3;ETM=ETMM3;

This line specifies that a Cortex-M3 processor is accessible using the preceding ARMCS-DP. The `ETM=ETMM3` section states that the processor has an associated ETM called ETMM3.

Name=ETMM3;Type=CSETM;Core=Cortex-M3;TraceOutput=TPIU;

This line specifies that an ETM is accessible using the preceding ARMCS-DP.

`TraceOutput=TPIU` signifies that this ETM can output into the component named *Trace Port Interface Unit* (TPIU).

`Core=Cortex-M3` signifies that the source for trace captured by this ETM is the Cortex-M3 device.

Name=Funnel;Type=CSTFunnel;

This line specifies that a CoreSight Trace Funnel is accessible using the preceding ARMCS-DP.

Name=TPIU;Type=CSTPIU;Port0=ETMM3;

This line specifies that a CoreSight TPIU is accessible using the preceding ARMCS_DP.

`Port0=ETMM3` indicates that the source of trace that is stored in this ETB is the component named ETMM3.

6.13.1 See also

Concepts

- *Setting up a CoreSight trace association file* on page 6-11
- *About trace associations* on page 6-6
- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for multiple trace sources* on page 6-21.

Reference

- *CoreSight device names and classes* on page 5-52
- *Format of trace associations* on page 6-8
- *Trace Association Editor dialog box* on page 6-9.

6.14 CoreSight topology and associations for multiple trace sources

The following figure shows there are two Cortex-R4 processors in the system, each with an associated ETM:

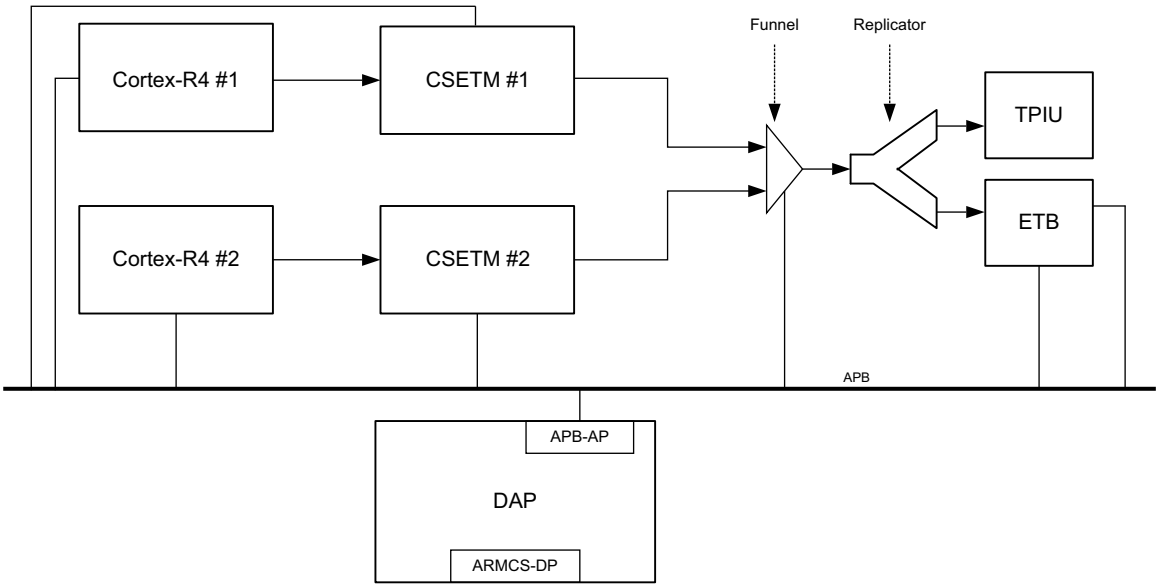


Figure 6-13 CoreSight system topology diagram - multiple trace source system

The processors and ETMs are numbered for convenience, and this numbering scheme is used in the Associations file.

The Association file for this is:

```
Name=ARMCS-DP;Type=ARMCS-DP; Name=Cortex-R4_1;Type=Cortex-R4;ETM=CSETM_1;
Name=Cortex-R4_2;Type=Cortex-R4;ETM=CSETM_2;
Name=CSETM_1;Type=CSETM;TraceOutput0=TPIU;TraceOutput1=ETB;Core=Cortex-R4_1;
Name=CSETM_2;Type=CSETM;TraceOutput0=TPIU;TraceOutput1=ETB;Core=Cortex-R4_2;
Name=Funnel;Type=CSTFunnel; Name=ETB;Type=CSETB;Port0=CSETM_1;Port1=CSETM_2;
Name=TPIU;Type=CSTPIU;Port0=CSETM_1;Port1=CSETM_2;
```

6.14.1 See also

Concepts

- *About trace associations* on page 6-6
- *Defining CoreSight trace associations* on page 6-7
- *CoreSight topology and associations for the CoreSight DK11* on page 6-15
- *CoreSight topology and associations for the Cortex-R4 FPGA* on page 6-17
- *CoreSight topology and associations for the Cortex-M3 FPGA* on page 6-19.

Reference

- *CoreSight device names and classes* on page 5-52
- *Format of trace associations* on page 6-8
- *Trace Association Editor dialog box* on page 6-9.

6.15 Configuring CoreSight processors

The following figure shows an example of the CoreSight device settings for a processor:

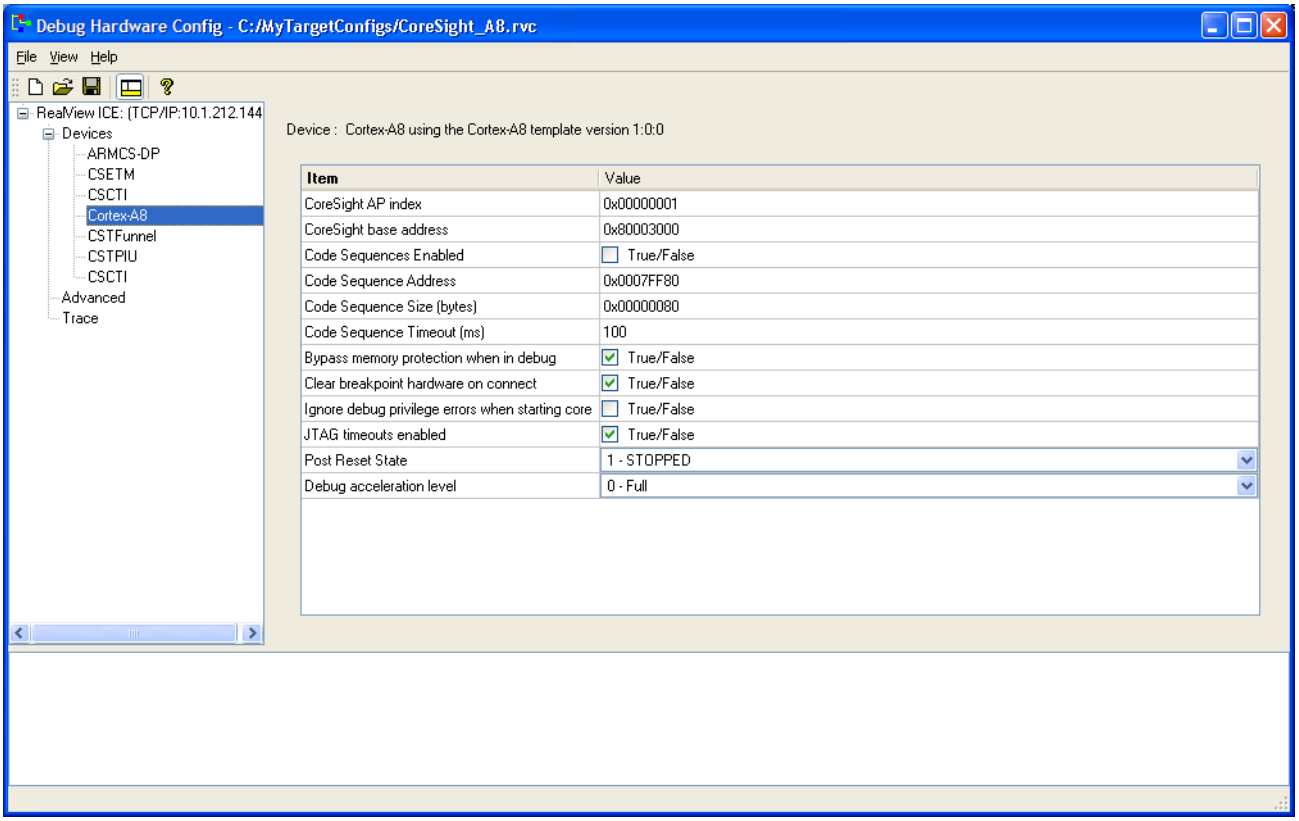


Figure 6-14 CoreSight device settings for a processor

The following configuration items are available when configuring CoreSight processors:

CoreSight AP index (CORESIGHT_AP_INDEX)

This is the index of the AP in the *Debug Access Port* (DAP) that must be used to access the CoreSight debug registers for the CoreSight component.

CoreSight base address (CORESIGHT_BASE_ADDRESS)

This is the base address of the CoreSight debug registers on the bus that is accessed through the AP as specified in the `CoreSight AP Index` configuration item.

6.15.1 See also

Concepts

- *Configuring ARM7, ARM9, and ARM11 processors in CoreSight systems* on page 6-24
- *Configuring CoreSight systems with multiple devices per JTAG-AP multiplexor port* on page 6-26
- *Configuring SecurCore behavior if the processor clock stops when stepping instructions* on page 5-38
- *Configuring TrustZone enabled processor behavior when debug privileges are reduced* on page 5-39.

Reference

- *Debug hardware Advanced configuration reset options* on page 5-37.

6.16 Configuring ARM7, ARM9, and ARM11 processors in CoreSight systems

The following ARM7, ARM9, and ARM11 processors are supported in CoreSight systems:

- ARM7EJ-S
- ARM7TDMI
- ARM7TDMI rev 4
- ARM926EJ-S
- ARM946ES
- ARM966ES
- ARM968ES
- ARM9EJ-S
- ARM1136JF-S
- ARM1156T2F-S
- ARM1176JZF-S
- MPCore.

You must add:

- the ARMJTAG-DP device
- the corresponding *JTAG Access Port* (JTAG-AP) for the processor.

The following configuration items area available when configuring these processors in CoreSight systems:

CoreSight AP index (CORESIGHT_AP_INDEX)

The index of the JTAG-AP in the DAP that must be used to access the CoreSight debug registers for the CoreSight component.

JTAG-AP Port index for core (JTAG_PORT_ID)

Each JTAG-AP implements eight JTAG port, each with its own TDI, TDO, TMS, and so on.The port index refers to the JTAG to which your CoreSight component is connected.

Fast memory download (FAST_MEM_WRITES)

The **Fast Memory Download** option is available for those targets where the DAP and the Core are running fast enough to handle the data being sent to them by the debug hardware unit without the debug hardware unit having to check that each individual transaction with the DAP has been successful. The processor is behind the DAP, so all processor accesses have to go through the DAP. As a guide, this setting must not be set for those targets that are FPGA-based.

Note

With this option set, error checking is disabled. If any errors occur, you are not informed. If problems are encountered when downloading images, uncheck this option to resolve them.

6.16.1 See also

Concepts

- *Configuring CoreSight processors* on page 6-22
- *Configuring CoreSight systems with multiple devices per JTAG-AP multiplexor port* on page 6-26
- *Configuring SecurCore behavior if the processor clock stops when stepping instructions* on page 5-38

- *Configuring TrustZone enabled processor behavior when debug privileges are reduced* on page 5-39.

Reference

- *Debug hardware Advanced configuration reset options* on page 5-37.

6.17 Configuring CoreSight systems with multiple devices per JTAG-AP multiplexor port

The debug hardware unit does not support auto-detection of devices behind a JTAG-AP. Therefore, you must manually specify the JTAG scan chain attributes so that the unit can put the other devices into BYPASS.

To debug CoreSight systems that have processors connected to the *Debug Access Port* (DAP) through JTAG-AP, debug hardware must know the pre-bits and post-bits for JTAG operations. The following figure shows a hypothetical scan chain that could be connected to a JTAG-AP.

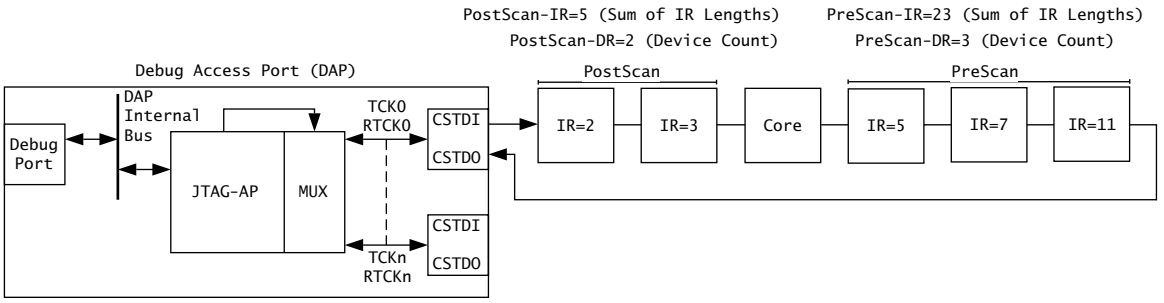


Figure 6-15 Scan chain connected to a JTAG-AP

Multiple devices on the scan chain are connected in series, with data flowing serially from TDI to TDO. This means that debugging a given target in the chain requires that certain pre-scan and post-scan bits are used to ensure that the other devices are not affected by the data directed at the target device, and that the data is positioned correctly in the serial scan for the target device.

To debug this system, you must set the following four configuration items:

- **Pre-scan IR bits for Devices after the core on the JTAG-AP scanchain (JTAG_AP_IR_PRE_BITS)**

  This is the total length of the JTAG *instruction registers* (IRs) for devices appearing between the processor being configured and the CSTDO input on the JTAG-AP port. In the figure above, the three devices that appear between the target processor and the CSTDO input on the JTAG-AP port have IR lengths 5, 7 and 11, respectively. Therefore, you must set this value to 23.

- **Post-scan IR bits for Devices before the core on the JTAG-AP scanchain (JTAG_AP_IR_POST_BITS)**

  This is the total length of the JTAG IRs for devices appearing between the CSTDI output on the JTAG-AP port and the processor being configured. In the figure above, the two devices that appear between the CSTDI output on the JTAG-AP port and the processor being configured have IR lengths 2 and 3, respectively. Therefore, you must set this value to 5.

- **Pre-scan DR bits for Devices after the core on the JTAG-AP scanchain (JTAG_AP_DR_PRE_BITS)**

  This is the total number of devices appearing between the processor being configured and the CSTDO input on the JTAG-AP port. In the figure above, there are three devices that appear between the processor being configured and the CSTDO input on the JTAG-AP port. Therefore, you must set this value to 3.

- **Post-scan DR bits for Devices before the core on the JTAG-AP scanchain (JTAG_AP_DR_POST_BITS)**

This is the total number of devices appearing between the CSTDI output on the JTAG-AP port and the processor being configured. In the figure above, there are two devices that appear between the CSTDI output on the JTAG-AP port and the processor being configured. Therefore, you must set this value to 2.

6.17.1 See also

Concepts

- *Configuring CoreSight processors* on page 6-22
- *Configuring ARM7, ARM9, and ARM11 processors in CoreSight systems* on page 6-24
- *Configuring SecurCore behavior if the processor clock stops when stepping instructions* on page 5-38
- *Configuring TrustZone enabled processor behavior when debug privileges are reduced* on page 5-39.

Reference

- *Debug hardware Advanced configuration reset options* on page 5-37.

Chapter 7
Using Trace

The following topics describe the trace features supported by your trace hardware:

7.1 About using trace hardware

Trace hardware is included in a DSTREAM unit, or is available as a separate RVT or RVT2 unit for use with RVI.

The trace data capture feature works in conjunction with the debug run control feature in debug hardware. Together, they provide real-time trace functionality for software running in leading edge *System-on-Chip* (SoC) devices with deeply embedded processors that contain the *Embedded Trace Macrocell* (ETM) logic.

The trace functionality enables:

- collection of trace information at clock speeds of up to 480MHz
- provision of a data streaming capability through a USB2 interface. This enables profiling directly from a hardware platform, in combination with a debug hardware unit.

The streaming of trace data removes the usual trace capture unit dependence on the size of the on-board buffer. It enables you to capture profiling data in a file on the system hosting the profiling software over long periods. The limitations are:

- the disk space available on the host system
- the amount of data you consider reasonable to analyze.

Note

Profiling with ARM Profiler is not supported from a DSTREAM unit. However, profiling with the trace view is supported.

7.1.1 See also

Tasks

- *Configuring trace lines (DSTREAM and RVT2 only)* on page 7-4
- *Configuring your debugger for trace capture* on page 7-6.

7.2 Trace hardware capture rates

If a *Trace Port Interface Unit* (TPIU) in continuous mode is used, all port widths from 1 to 32 can be output by the target. DDR clocking enables data to be output from the ETM on both edges of **TRACECLK**. This effectively halves the clock frequency for the same data rate.

7.2.1 DSTREAM trace hardware capture rates

The maximum capture rate of the DSTREAM unit is 600 Mbps per trace signal using a 300MHz DDR clock signal. The capture buffer is 4GB in size.

Note

Some debuggers have limitations when tracing with DSTREAM. See your debugger documentation for details of the trace capabilities of your debugger.

7.2.2 See also

Tasks

- *Configuring trace lines (DSTREAM and RVT2 only)* on page 7-4
- *Configuring your debugger for trace capture* on page 7-6.

Concepts

- *About using trace hardware* on page 7-2.

7.3 Configuring trace lines (DSTREAM and RVT2 only)

If you have a DSTREAM unit or RVT2 unit, you can configure delays on the trace lines. To do this:

1. Open the Debug Hardware Config utility.
2. If you have already configured a debug hardware unit, select **Open** from the **File** menu to locate and open the corresponding configuration file.

   Otherwise:

   a. Connect to the required debug hardware unit.

   b. Create a new debug hardware configuration.
3. Select the **Trace** node in the tree control. The following figure shows an example:

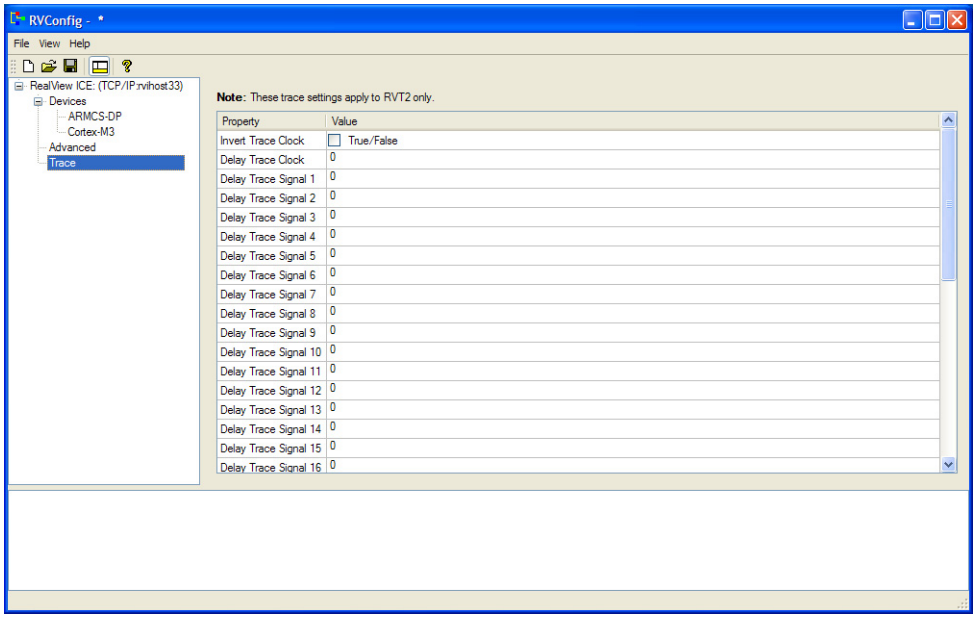


Figure 7-1 Trace node in Debug Hardware Config

You can delay each line by a specified amount of time (expressed in picoseconds, in 75ps intervals). Default delays are configured into the unit, and you are able to delay each signal by a specified amount relative to these defaults, allowing for variations in target hardware.

You can also invert the clock so that data is sampled on the falling edge (rather than on the rising edge) of the clock. To do this, select the **True/False** checkbox. The default is `False` (unchecked).

Note

Some debuggers have limitations when tracing with DSTREAM. See your debugger documentation for details of the trace capabilities of your debugger.

7.3.1 See also

Tasks

- *Connecting to a debug hardware unit* on page 2-7
- *Creating a debug hardware configuration file* on page 5-4
- *Configuring your debugger for trace capture* on page 7-6.

Concepts

- *About using trace hardware* on page 7-2.

Reference

- *Trace hardware capture rates* on page 7-3.

7.4 Configuring your debugger for trace capture

When you have installed the host software and connected and configured the debug hardware unit, you must configure your debugger to use trace.

For full details on how to capture trace with your debugger, see the documentation that accompanies your debugger.

7.4.1 See also

Tasks

- *Configuring trace lines (DSTREAM and RVT2 only)* on page 7-4.

Concepts

- *About using trace hardware* on page 7-2.

Reference

- *Trace hardware capture rates* on page 7-3.

Chapter 8
Debugging with your debug hardware unit

The following topics provide information about debugging with debug hardware:

Note

For more general information about debugging, see your debugger documentation.

8.1 Post-mortem debugging

Post-mortem debugging enables you to examine the state of a system that has previously been running but is currently not connected to debug hardware.

8.1.1 Prerequisites

Before you can examine a running target with debug hardware, you must configure the debug hardware unit for that target. If you have a target that is operating without a debug hardware unit connected, and you want to examine it to find out why it is behaving in a particular way, you must power-up the debug hardware unit and configure the connection without disturbing the state of the target. This requires that the debug hardware unit is powered before it is connected to the target.

The debug hardware unit includes power conditioning and switching circuitry that enables you to plug and unplug the JTAG cable without affecting the target.

Note

The voltage reference used by the debug hardware unit JTAG circuit is generated from the **VTref** signal present on the JTAG connector. If this signal is not connected at the target, you must modify the target or the JTAG cable to supply a suitable reference. Connecting **VTref** to **Vsupply** is usually sufficient.

8.1.2 Procedure

To connect to a running target:

1. Ensure that the JTAG input lines **TDI**, **TMS**, **nSRST**, and **nTRST** have pull-up resistors (normal practice), and **TCK** has a pull-down resistor, so that when the adaptor is disconnected from the target these lines are in their quiescent state.
2. Plug the power jack into the debug hardware unit and wait for it to boot.
3. Configure the debug hardware connection. You must do one of the following:
   - load a configuration that you have previously saved
   - manually configure the connection
   - autoconfigure using a separate test system.

   Note

   Do not use autoconfigure on the target to be debugged, because doing so might reset the processor.
4. If the target processor, such as an ARM7TDMI, does not have any system registers, you must explicitly configure the endianness.

   Note

   Do not automatically detect the endianness of target processors that do not have a system register. Doing so might disturb the state of the processor.
5. Plug the JTAG cable into the target.

   Caution

   To prevent unwanted resets during connection of the debug hardware, it is essential that either:
   - the target and debug hardware are properly earthed

- the ground pins of the debug connector make contact before the signal pins.

6. Start the debugger, and connect to the running target.

   In your debug hardware configuration, set the **Post Reset State** to **Running**.

   In your debugger, connect using the **Connect (Connection Modes)** of **No Reset / No Stop**.

7. To get a high-level (source code) view of the problem, load the symbol table for your target program into the debugger.

8. If the processor stopped during debugging, then in your debugger:

   a. Clear any breakpoints that you have set.

   b. Start the processor running

   c. Disconnect from all targets to which the debugger is connected on your development platform.

   If the processor is still running, then in your debugger disconnect from all targets to which the debugger is connected on your development platform.

9. Unplug the JTAG connector from the development platform.

8.1.3 See also

Tasks

- *Configuring the debug hardware Advanced settings* on page 5-47.

Concepts

- *Semihosting* on page 8-4
- *Strategies used by debug hardware to debug cached processors* on page 8-15
- *Debugging applications in ROM* on page 8-17.

8.2 Semihosting

Semihosting enables the ARM processor target to make I/O requests to the computer running the debugger. This means the target does not require a screen, keyboard, or disk during the development period. These requests are made as a result of calls to C library functions, for example, `printf()` and `getenv()`.

8.2.1 See also

Tasks

- *Post-mortem debugging* on page 8-2
- *Adding an application SVC handler when using debug hardware* on page 8-5.

Concepts

- *Cortex-M3 semihosting* on page 8-7
- *Strategies used by debug hardware to debug cached processors* on page 8-15
- *Debugging applications in ROM* on page 8-17.

8.3 Adding an application SVC handler when using debug hardware

Many applications require their own SVC handlers in addition to semihosting SVCs. To ensure that the application SVC handler cooperates with the debug hardware semihosting mechanism, use your debugger to:

1. Install the application SVC handler into the vector table.
2. Modify the value of SEMIHOST_VECTOR to point to a location that is only reached if your handler does not recognize the SVC, or recognizes it as a semihosting SVC.

For example, a particular SVC handler might detect if it has failed to handle a SVC and branch to an error handler. An example of a basic exception handler is shown in the following example.

Example 8-1 Basic SVC handler

```
                          ; r0 = 1 if SVC handled
CMP r0, #1                ; Test if SVC has been handled.
BNE NoSuchSVC             ; Call unknown SVC handler.
LDMFD sp!, {r0}           ; Unstack SPSR...
MSR spsr_cf, r0           ; ...and restore it.
LDMFD sp!, {r0-r12, pc}^ ; Restore registers and return.
```

You can modify this code for use in conjunction with debug hardware semihosting as shown in the following example.

Example 8-2 SVC handler with debug hardware link

```
                         ; r0 = 1 if SVC handled
CMP r0, #1               ; Test if SVC has been handled.
LDMFD sp!, {r0}          ; Unstack SPSR...
MSR spsr_cf, r0          ; ...and restore it.
LDMFD sp!, {r0-r12, lr} ; Restore registers.
MOVEQS pc, lr            ; Return if SVC handled.
Semi_SVC
MOVS pc, lr
```

You must then set up the SEMIHOST_VECTOR with the address of Semi_SVC. The instruction at this address is never actually executed because the debug software returns directly to the application after processing the semihosted SVC. Using a normal SVC return instruction ensures that the application does not crash if the semihosting breakpoint is not set up.

If the application is linked with the semihosted ARM C library, and therefore uses the C library startup code, you must change the contents of SEMIHOST_VECTOR before the application installs its own handler, typically by setting a breakpoint in the main code. This is because, if SEMIHOST_VECTOR is set to the fall-through part of the application SVC handler before the application starts execution, the semihosted SVCs that are called by the library initialization can trigger an unknown breakpoint error. At this point, the SVC vector has not yet had the application handler written to it, and might still contain the software breakpoint bit pattern. This triggers a breakpoint that the debug software does not know about, because the SEMIHOST_VECTOR address has moved to a place that cannot currently be reached.

Note

If semihosting is not used by your application, including the startup code, you can simplify this process by setting SEMIHOST_ENABLED to zero.

You must take care when moving an application that previously ran in conjunction with the Angel debug monitor onto a debug hardware system. On Angel debug monitor systems, application SVC handlers are typically added by moving and adjusting the contents of the Angel-installed SVC vector to another place, and installing the application SVC handler into the SVC vector. This method does not apply to the debug software because there is no instruction to move out of the SVC vector, and no code to jump to. Therefore, when moving an application onto a debug hardware-based system, you must convert to the debug hardware way of installing the application and semihosted SVC handlers.

8.3.1 See also

Concepts

- *Cortex-M3 semihosting* on page 8-7.

8.4 Cortex-M3 semihosting

Because Cortex-M3 does not provide vector catch on SVC, and the vector table contains jump addresses rather than instructions, semihosting cannot be supported using an SVC instruction.

As an alternative, semihosting is implemented using a specific software breakpoint that is recognized as a semihosting break by the debugger. The breakpoint instruction opcode contains an immediate 8-bit value. The C library uses the `BKPT 0xAB` opcode for semihosting. The debugger can test for this opcode pattern to determine if the breakpoint was a semihosting request or not.

When the semihosting break is executed, the semihosting call is processed in the normal way. After processing, execution continues from the instruction that follows the software breakpoint. The debugger does not stop on the breakpoint.

8.4.1 See also

Tasks

- *Adding an application SVC handler when using debug hardware* on page 8-5.

8.5 Hardware breakpoints

Depending on implementation options, most ARM processors contain dedicated hardware resources, such as ARM EmbeddedICE® logic, for matching against specific hardware events. Your debugger enables you to configure these resources to implement instruction and data breakpoints.

Note

Data breakpoints are also sometimes referred to as watchpoints.

The resources available depend on the processor you are using. See the data sheet for your processor for information.

Hardware breakpoints might also provide additional matching capabilities. Examples of this include matching on an external signal, and distinguishing between privileged and non-privileged accesses. The Set Address/Data Breakpoint dialog box displays the capabilities of your hardware.

Hardware instruction breakpoints do not require the instruction in memory to be changed. This means that they can be used to debug code in Flash and ROM, and can be used with self-modifying code.

8.5.1 See also

Concepts

- *Software instruction breakpoints* on page 8-9
- *Processor exceptions* on page 8-10
- *Breakpoints and the program counter* on page 8-11
- *Interaction between breakpoint handling in the debug hardware and your debugger* on page 8-12
- *Problems setting breakpoints* on page 8-14.

8.6 Software instruction breakpoints

For processors that do not support hardware instruction breakpoints, or in cases where you have used up all the available hardware breakpoint resources, you can use software instruction breakpoints. Software breakpoints modify the instruction in memory to create a special value that causes the processor to enter debug state when executed. The value written to memory depends on the processor you are using. For ARM processors, one of the following schemes is used, depending on the architecture and processor revision:

- An undefined instruction is written to memory, and a hardware breakpoint resource is used to spot this instruction being executed. The processor enters debug state when the hardware breakpoint unit spots the undefined instruction entering the execute pipeline stage.
- An ARMv5 `BKPT` instruction is written to memory, and a hardware breakpoint resource is used to spot the instruction being executed. The processor enters debug state when the hardware breakpoint unit spots the `BKPT` instruction entering the execute pipeline stage.
- An ARMv5 `BKPT` instruction is written to memory. When this instruction is executed, the processor automatically enters debug state.

Where a hardware breakpoint unit is used to spot software instruction breakpoints, only a single hardware resource is used, no matter how many software instruction breakpoints are set. If you have difficulty setting software instruction breakpoints, you might have to free up a hardware breakpoint resource first.

Software breakpoints cannot be used to debug code in Flash or ROM, and can be unreliable in self-modifying code.

Note

When viewing memory or disassembly, debug hardware reports the actual contents of memory. Prior to running, any software breakpoints are written to memory. When the processor halts, the software breakpoints are removed from memory. On a number of processors, it is not possible to access memory while running. This means that if you disconnect debug hardware from the processor while the target is running, the breakpoints are left in memory. If the processor subsequently executes one of the instructions, then (depending on the processor architecture) the processor either stops at the software breakpoint or causes the processor to take an undefined exception.

8.6.1 See also

Concepts

- *Hardware breakpoints* on page 8-8
- *Processor exceptions* on page 8-10
- *Breakpoints and the program counter* on page 8-11
- *Interaction between breakpoint handling in the debug hardware and your debugger* on page 8-12
- *Problems setting breakpoints* on page 8-14.

8.7 Processor exceptions

Depending on implementation options, most ARM processors provide dedicated hardware to enter debug state when a predetermined event occurs.

Most recent ARM processors provide hardware for trapping exceptions. When enabled, the effect is similar to placing a breakpoint on the selected vector table entry. This is called *vector catch*. However:

- Some ARM processors, such as ARM7, do not provide vector catch hardware. For these processors, debug hardware simulates vector catch using instruction breakpoints.
- For Cortex-M3, this is equivalent to putting a breakpoint at the target of the vector. Cortex-M3 has a restricted set of vector catches available.
- If the exception vectors are in ROM, debug hardware must use hardware breakpoints to simulate vector catch. This reduces the number of resources available for other purposes, if the processor does not have vector catch support.

When the debug hardware simulates vector catch on earlier ARM processors that do not have vector catch support, it uses a software breakpoint when the vector table is located in RAM.

You must take care when debugging through a system reset. Some hardware targets alter the memory map after reset, so the location of in physical memory containing software breakpoint might not be in the correct reset position. The following warning is output to the your debugger console if debug hardware simulates reset vector catch using an instruction breakpoint:

```
Warning: A software breakpoint is being used to simulate reset vector catch.
This may fail to be hit if the memory is remapped when a reset occurs.
```

The exact behavior of the ARM vector catch hardware depends on the processor. ARM9 processors enter debug state only when the specified exception occurs. Other processors, such as ARM11 or older processors that use a breakpoint, enter debug state whenever the instruction at the exception vector is executed, regardless of whether the exception occurs or not.

8.7.1 See also

Concepts

- *Hardware breakpoints* on page 8-8
- *Software instruction breakpoints* on page 8-9
- *Breakpoints and the program counter* on page 8-11
- *Interaction between breakpoint handling in the debug hardware and your debugger* on page 8-12
- *Problems setting breakpoints* on page 8-14.

8.8 Breakpoints and the program counter

The following events describe the value of the program counter when a breakpoint is taken:

Hardware data breakpoints

The address of the program counter after hitting a hardware data breakpoint depends on the processor being used.

For ARM processors, a skid of either one or two instructions occurs after a data breakpoint is hit. This means that the instruction that generated the breakpoint, and possibly the one after that, are both executed. The program counter shown in your debugger might not be the address of the instruction that generated the breakpoint.

Hardware instruction breakpoints

The address of the program counter after hitting a hardware instruction breakpoint depends on the processor being used.

For ARM processors, no skid occurs after hitting a hardware breakpoint. This means that the instruction that generated the breakpoint has not been executed, and the program counter is set to this address.

Software instruction breakpoints

The address of the program counter after hitting a software breakpoint is always the address of the breakpoint. Unless the instruction is a `BKPT` instruction, the instruction that generated the breakpoint is not yet executed.

Processor events

The address of the program counter after a processor event is hit depends on the processor being used. For ARM processors, vector catch hardware stops with the program counter on exception vector, before the instruction at that address is executed.

8.8.1 See also

Concepts

- *Hardware breakpoints* on page 8-8
- *Software instruction breakpoints* on page 8-9
- *Processor exceptions* on page 8-10
- *Interaction between breakpoint handling in the debug hardware and your debugger* on page 8-12
- *Problems setting breakpoints* on page 8-14.

8.9 Interaction between breakpoint handling in the debug hardware and your debugger

The following describe the interaction between breakpoint handling in the debug hardware and breakpoint handling in your debugger:

Break details or break capabilities

You can find out what hardware breakpoint resources are available by viewing the break details or break capabilities in your debugger.

Memory maps

Your debugger enables you to define a memory map to describe the layout and type of memory in your system. When you set a breakpoint, areas of memory that are marked as read-only, such as Flash and ROM, automatically use hardware instruction breakpoints. All other types of memory use software instruction breakpoints by default.

Stepping

When you step through code, the debugger usually sets a temporary breakpoint on the destination address. If the code is in read-only memory, or if the software breakpoint implementation requires hardware assistance, a hardware breakpoint is used for this. If you are unable to step, you might have to free up a hardware breakpoint resource.

Some processors, such as ARM9, provide dedicated single-step hardware. debug hardware uses this hardware if it is available, but steps larger than a single instruction might revert back to using breakpoints, to improve efficiency.

Note

For ARM7, ARM9, ARM11 or Cortex-A8 processors, interrupts are disabled when single-stepping with debug hardware. For the Cortex-M3 processor, interrupts are enabled when single-stepping with debug hardware.

Interrupt behavior applies only to debug hardware single-instruction stepping. Higher-level stepping depends on the strategy in your debugger, that is, whether you have used the place Breakpoint and run method, or the multiple single-instruction steps method.

Note

When hardware single-step is used, debug hardware prevents the processor from processing any pending interrupts.

Resource allocation

Debug hardware allocates hardware breakpoint resources as they are received, rather than allocating all the resources at the same time when the debugging session begins. Therefore, if you attempt to set a breakpoint when there are insufficient resources available, debug hardware displays an error message as soon as you try to set the breakpoint, rather than waiting until debugging begins.

8.9.1 See also

Concepts

- *Semihosting* on page 8-4
- *Hardware breakpoints* on page 8-8
- *Software instruction breakpoints* on page 8-9
- *Processor exceptions* on page 8-10

- *Breakpoints and the program counter* on page 8-11
- *Problems setting breakpoints* on page 8-14.

8.10 Problems setting breakpoints

If you have problems stepping or setting breakpoints, it might be because you have run out of hardware breakpoint resources. To work around this, you can try freeing some hardware breakpoint resources then repeating the action. Some examples of how you can free hardware breakpoint resources include:

- disable any breakpoints that you do not require
- change hardware breakpoints to software breakpoints where possible
- disable vector catch if you are debugging an early processor, such as the ARM7TDMI, and the vector table is in ROM
- disable semihosting if you are not using it.

8.10.1 See also

Concepts

- *Hardware breakpoints* on page 8-8
- *Software instruction breakpoints* on page 8-9
- *Processor exceptions* on page 8-10
- *Breakpoints and the program counter* on page 8-11
- *Interaction between breakpoint handling in the debug hardware and your debugger* on page 8-12.

8.11 Strategies used by debug hardware to debug cached processors

When debugging a cached processor, debug hardware uses the following strategies.

On debug entry

- Debug hardware forces *Write-Through* (WT) on processors that support this debug feature.
- Debug hardware disables cache line fill on processors that support disabling of this feature in debug.
- Debug hardware disables *Translation Look-aside Buffer* (TLB) loads on processors that support disabling of this feature in debug.
- If data is read from cacheable memory, it is only read into the caches if, and only if, disable linefill is not possible.
- TLB entries and caches remain enabled.

On data write

- If WT is possible, nothing cache-related is performed.
- If WT is not possible, the write depends on processor size and data size:
  1. Debug hardware can write to memory with caches enabled, and then write disabled, effectively simulating write through.
  2. Debug hardware can clean and invalidate the D_{cache} and disable it.

     Note

     The ARM940T processor requires that Code Sequences are enabled to do this.

On restart into debug

- On processors that support the features, forced WT is removed, linefills are re-enabled, and TLB loads are enabled. If, and only if, data has been written, the I_{cache} is invalidated. If, and only if, D_{cache} has been disabled, then it is re-enabled.

Data writes that could cause the cache operations described include user accesses using your debugger, downloads, and any software breakpoints present in the system.

Note

For the ARM940T processor you must configure the code sequence settings before attempting to debug with caches enabled.

When the cache is enabled, the speed of semihosting decreases, because of the additional cache maintenance overhead performed by the debugger.

8.11.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Semihosting* on page 8-4
- *Debugging applications in ROM* on page 8-17.

8.12 Considerations when debugging processors with caches enabled

When debugging a processor with caches enabled, you might have to provide the address of an area of memory on the target that can be used exclusively by debug hardware. On some targets, the debug software downloads code sequences to this area to perform various tasks, such as cleaning the cache, and accessing the system registers. Debug hardware does not preserve the contents of this area.

A code sequence area is only required for certain processors where the required operations cannot be performed directly over JTAG. If debug hardware requires a code sequence area, and one has not been enabled, errors are displayed within the debugger. For example:

- `Error V28305 (Vehicle): Memory operation failed`
- `Warning: Code sequence memory area size error`
- `Unable to load code sequence into defined memory area.`

Note

The code sequence area must be 128 bytes long and in a non-cacheable, readable and writeable area.

To set up a code sequence area, use the options for each specific processor in the Debug Hardware Config utility. This provides access to configuration items for each processor for:

- enabling code sequences
- setting the address and size of the code sequence areas.

Note

These settings might also be available in your debugger Registers view. Any settings modified using the Registers view in your debugger are only modified for the duration of the debug session. Any settings modified using the Debug Hardware Config utility are persistent until modified again.

8.12.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Semihosting* on page 8-4
- *Strategies used by debug hardware to debug cached processors* on page 8-15
- *Debugging applications in ROM* on page 8-17.

8.13 Debugging applications in ROM

Some of the issues involved with debugging applications in ROM using debug hardware are described in the following:

- *Debugging from reset* on page 8-18
- *Debugging systems with ROM at the exception vector* on page 8-22.

8.13.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Semihosting* on page 8-4
- *Strategies used by debug hardware to debug cached processors* on page 8-15.

8.14 Debugging from reset

You can debug systems running in ROM. A typical embedded system has the application programmed in non-volatile memory, such as ROM or Flash. When target hardware is powered up, the application starts running. When connecting the debugger to a target already running the application, the application stops at an arbitrary point in the code. The default behavior of the debugger is to stop the target on connection. Loading the image symbols gives you source code view of the current location.

This means that you can examine the state of the system and restart execution from the current place. In some cases, this is sufficient. However, in many cases it is preferable to restart execution of the application as if from power-on. There are three ways to do this:

- debugging with a simulated a reset
- debugging with a reset register
- debugging with a target reset.

When you debug code that is running from ROM, you must ensure that at least one breakpoint unit remains available so that you can set breakpoints on code in ROM, because you cannot use software breakpoints for this purpose. On a processor without vector catch hardware, you must disable semihosting and vector catching as soon as possible after starting up the debugger. This can reduce the chances of the debugger taking these units for its own use.

On ARM processors that use a breakpoint resource to implement software breakpoints, such as the ARM7TDMI, you must remove all software breakpoints if you are out of breakpoint resources. This enables you to place a single hardware breakpoint in ROM.

Another factor in debugging a system in ROM is that the ROM image does not contain any debug information. When debugging, symbol or source code information is available by loading the relevant information into the debugger from the ELF image on the host.

8.14.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Debugging with a simulated reset* on page 8-19
- *Debugging with a reset register* on page 8-20
- *Debugging with a target reset* on page 8-21
- *Debugging systems with ROM at the exception vector* on page 8-22.

8.15 Debugging with a simulated reset

You can, where supported, simulate a reset from within the debugger by setting:

- the pc to the address of the reset vector
- the CPSR to change into Supervisor mode with interrupts disabled.

This simulates the state of the ARM processor at power-on or reset, but it does not perform post-reset tasks such as resetting the memory map, or initializing any target-specific features such as peripheral registers. It is recommended that you modify these target-specific features to resemble their startup state before executing the application again, if possible. You can automate this procedure using the scripting facilities of your debugger.

8.15.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Debugging from reset* on page 8-18
- *Debugging with a reset register* on page 8-20
- *Debugging with a target reset* on page 8-21
- *Debugging systems with ROM at the exception vector* on page 8-22.

8.16 Debugging with a reset register

Where supported, some processors, particularly CoreSight ones such as the Cortex-M3, include a reset register that can reset the processor without using the physical reset lines. In a multi-processor system, this can be used to reset only the target processor and not the complete system. This type of reset can be selected by setting the reset type to `Ctrl_Reg`.

8.16.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Debugging from reset* on page 8-18
- *Debugging with a simulated reset* on page 8-19
- *Debugging with a target reset* on page 8-21
- *Debugging systems with ROM at the exception vector* on page 8-22.

8.17 Debugging with a target reset

Depending on the design of the reset circuitry, you might be able to carry out a target reset of the board. Two forms of reset are required on the board:

- A full power-on reset that resets everything on the board.
- A Reset button that resets your development platform depending on whether or not it is a CoreSight system:
  - For non-CoreSight systems, everything on the board is reset except the EmbeddedICE logic.The EmbeddedICE logic is the debug logic in the processor.
  - In a CoreSight system, the **nSRST** (system reset) resets the entire design, except for the debug subsystem and trace subsystem. This includes the debug logic from all devices, debug and trace bus, access ports, and *Debug Access Port* (DAP).

Note

The Reset button mentioned here must not be confused with the RESET button located on the debug hardware unit itself.

If your target implements a Reset button that drives **nTRST** in addition to **nSRST**, then the EmbeddedICE logic is reset along with the board, and the debugger might not be able to regain synchronization. This design is not recommended.

If a vector catch is set on the reset vector (or on the start address of the reset code) and the recommended reset circuit is used, when the target is reset, it halts on reset as required.

8.17.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Debugging from reset* on page 8-18
- *Debugging with a simulated reset* on page 8-19
- *Debugging with a reset register* on page 8-20
- *Debugging systems with ROM at the exception vector* on page 8-22.

Reference

ARM® DSTREAM™ Setting-up the Target Hardware:

- The DSTREAM unit, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0481e/CHDCJEFH.html`.

ARM® DSTREAM™ System and Interface Design Reference:

- System Design Guidelines, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0499e/Chdbdcid.html`.

8.18 Debugging systems with ROM at the exception vector

When debugging processors without vector catch hardware and with ROM rather than RAM at the exception vector, you must disable vector catching. This prevents debug hardware from trying to set hardware breakpoints on the vector table.

The default setting is to enable, exception trapping on reset, prefetch abort, and data abort. On a processor without vector catch hardware, three breakpoint resources are used in ROM. Therefore, if a processor has three or fewer resources, you cannot debug applications running on that processor.

8.18.1 See also

Tasks

- *Post-mortem debugging* on page 8-2.

Concepts

- *Debugging with a simulated reset* on page 8-19
- *Debugging with a reset register* on page 8-20
- *Debugging with a target reset* on page 8-21.

Chapter 9
Configuring debug hardware for GDB

The following topics describe the basic steps required to configure the debug hardware unit to a state where you can begin debugging your image using the *GNU Debugger* (GDB):

9.1 About configuring debug hardware for debugging with GDB

Your debug hardware provides functionality that extends the debugging features available in GDB.

Note

GDB does not directly support USB.

Note

To find the latest information on GDB compatibility with debug hardware, see the debug hardware Release Notes.

For information on GDB and Command Monitor error codes, see the ARM web site.

9.1.1 See also

Tasks

- *Methods of connecting from remote GDB sessions* on page 9-9.

Concepts

- *Feature support when debugging with GDB* on page 9-4
- *About building for standalone target platforms* on page 9-8
- *Preparing your debug hardware for remote GDB connections* on page 9-21
- *Loading and booting a complete system* on page 9-26
- *Multiprocessor debugging with GDB and debug hardware* on page 9-34.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6.

Other information

- ARM web site, `http://www.arm.com`.

9.2 Feature support when debugging with GDB

Your debug hardware unit supports connections from remote GDB sessions over TCP/IP. These GDB connections support all non-OS specific functionality.

9.2.1 Features supported

When using GDB, your debug hardware unit supports:

- full memory and register access
- run and stop
- software and hardware breakpoints and watchpoints
- target reset (restart)
- binary program downloading
- step-over-range
- single stepping.

9.2.2 Features not supported

When using GDB, debug hardware unit does not provide support for:

- threads (in start-stop debugging)
- debugging over the debug hardware USB port
- synchronized start and step on multi-processor systems.

9.2.3 See also

Concepts

- *Debug hardware TCP/IP port numbering* on page 9-6
- *About building for standalone target platforms* on page 9-8
- *Methods of connecting from remote GDB sessions* on page 9-9.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7.

9.3 Debugging modes for GDB

You can use the following debugging modes with GDB:

- Halt-mode debugging, where the target stops while you examine it.
- Monitor-mode debugging, where the target continually runs under control of monitor software on the target. GDB communicates with the monitor using Virtual Ethernet/TTY connections through the *Debug Communications Channel* (DCC).

9.3.1 See also

Concepts

- *Feature support when debugging with GDB* on page 9-4
- *About building for standalone target platforms* on page 9-8.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7.

9.4 Debug hardware TCP/IP port numbering

To use the debug hardware Command Monitor and debug your target with GDB, the debug hardware unit uses the TCP/IP ports described in the following table:

Table 9-1 Debug hardware TCP/IP ports

Ports	Description
4000 series	This port range is used to connect to a target from GDB, and to perform halt-mode debugging. Each device on the JTAG scan chain, or behind a CoreSight DAP, is allocated a TCP/IP port number for connection from GDB. Ports are allocated in sequence, with port 4000 connected to the first device in the scan chain. Synchronized start and step are not supported. **Note** Any attempt to connect to a non-processor device fails, but the port number is still reserved for that device.
5000 series	This port range is used for Monitor-mode debugging and other Virtual Ethernet/TTY mode connections. **Note** To use these ports you must set the DCC mode to a non-zero value.

9.4.1 See also

Tasks

- *Connecting to targets from GDB through debug hardware* on page 9-22.

Reference

- *DCC modes* on page 9-7.

9.5 DCC modes

If your target application communicates using DCC, you must configure the DCC mode. You can set the following DCC modes:

Mode 0: raw DCC

Raw, unprocessed, data is fed to the client through the DCC register on the target.

Mode 1: redirected raw DCC

In this mode, the data is fed over TCP/IP ports starting at 5000. Data is sent from the host to the target 4 bytes at a time. If fewer than 4 bytes are available, the data is padded with 0 bytes until it is 4 bytes long. Data from the target to the host is received 4 bytes at a time, and no padding or trimming is performed.

Note

It is recommended that you use the Virtual Ethernet/TTY mode (mode 2). However, if the Virtual Ethernet/TTY mode is unsuitable for your application, then you can use mode 1 DCC but you must also implement a suitable communications protocol.

Mode 2: Virtual Ethernet/TTY mode

Virtual Ethernet/TTY is used for providing virtual serial and Ethernet connections to the target. As far as the debug host and the target are concerned, Virtual Ethernet/TTY mode is a debug hardware feature that provides:

- A virtual Ethernet feature using the DCC channel and a collection of software tools in debug hardware and the host PC. It enables TCP/IP to be used to the target as though the target has an Ethernet port of its own.
- A virtual serial port feature using the DCC channel and a collection of software tools in debug hardware and the host PC.

9.5.1 See also

Tasks

- *Connecting to targets from GDB through debug hardware* on page 9-22.

Concepts

- *Setting DCC parameters* on page 9-23
- *DCC and interrupts* on page 9-25.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *rvigdbconfig command syntax* on page 9-27
- *rviload command syntax* on page 9-28.

9.6 About building for standalone target platforms

If you are building for a standalone target platform (that is, without an operating system), the precompiled C library of the GNU toolchain for ARM architectures assumes that a debug monitor is resident in ROM.

If you are not using a debug monitor, Red Hat eCos/Redboot, or any other operating system, you must provide the following components:

- Your own I/O routines and optionally a target GDB stub.
- The `crt0.S` source file (mandatory). This source file provides the C startup procedure that is responsible for setting up the stack and heap, and for initializing C static and global variables.

Note

If you are using a debug monitor, Red Hat eCos/Redboot or other operating system, you must provide at least these components and possibly a gdbserver.

Documentation on how to do this is readily available from the Internet.

9.6.1 See also

Concepts

- *Feature support when debugging with GDB* on page 9-4.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6.

9.7 Methods of connecting from remote GDB sessions

The method you use to connect to a target depends on:

- the resources required by the target application (for example, an IP stack)
- the debugging facilities available on the target (for example, a GDB stub)
- whether or not your target has an embedded OS, such as Linux, that is running gdbserver instances.

9.7.1 See also

Tasks

- *Preparing your debug hardware for remote GDB connections* on page 9-21
- *Loading and booting a complete system* on page 9-26
- *Multiprocessor debugging with GDB and debug hardware* on page 9-34.

Concepts

- *About configuring debug hardware for debugging with GDB* on page 9-3
- *Connection methods for each debugging mode* on page 9-10
- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

9.8 Connection methods for each debugging mode

How you connect to a target determines the debugging mode. The following connection methods are available for each debugging mode:

9.8.1 Halt-mode debugging

Halt-mode debugging is the simplest method of debugging a target with GDB. You directly connect to debug hardware, that then controls the starting and stopping of the processor. This method of connecting is subsequently referred to as an RVI-GDB connection.

9.8.2 Monitor-mode debugging

Monitor-mode debugging requires that your target application communicate with GDB using the *Debug Communications Channel* (DCC) of an ARM architecture-based processor. However, if your target application includes an Ethernet feature, you do not have to use DCC. Different DCC modes are available depending on the requirements of your target.

The connection methods for Monitor-mode debugging are:

Target-GDB connections

Semi-transparent connections to GDB stubs. The GDB stub communicates with the GDB client using the DCC channel as a serial port. The debug hardware unit makes this connection available on a TCP/IP port to which the GDB clinet connects. The GDB stub must be compiled into the target application.

Target-GDB-Virtual Ethernet connections

An extension to Target-GDB connections for standalone applications running an IP stack. The GDB stub communicates with the GDB client using the DCC channel as an Ethernet channel. The debug hardware unit makes this connection available on a TCP/IP port to which the GDB client connects.

GDBserver connections

An alternative to Target-GDB-Virtual Ethernet connections where the target is running gdbserver running under an *operating system* (OS).

GDB-NFS connections

Connections to the root filesystem on the target OS that is mounted over NFS. The debug hardware unit acts as a bridge between the debug host and the target OS.

9.8.3 See also

Tasks

- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

Concepts

- *DCC modes* on page 9-7.

9.9 Connections to a target without built-in GDB support (RVI-GDB)

These are connections to targets where no GDB stub has been built into the target application, or when you want to perform halt-mode debugging. Connections of this type use the built-in GDB protocol interpreter of debug hardware to control the CPU directly, and are referred to as RVI-GDB connections. When you want to examine the internal state of the CPU (such as registers, memory, and variables), the image on the target stops executing. After examining the required state, you must start the image again. The following figure shows the configuration:

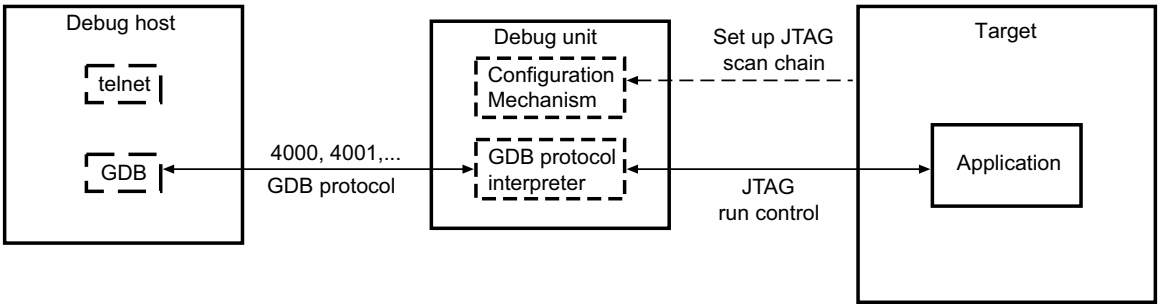


Figure 9-1 RVI-GDB connections

Note

GDB does not support Semihosting over JTAG. Therefore, any prompts and messages that are output by the application cannot be displayed in your debugger.

9.9.1 RVI-GDB Scenarios

Use the RVI-GDB connection method to:

- perform run and stop debugging of a single ARM processor
- perform run and stop debugging with GDB at the same time as debugging the application. That is, for example, if connecting to a target with a GDB stub (Target-GDB connections), or if your target application requires TCP/IP communication with the debug host (Target-GDB-Virtual Ethernet connections).

Note

When the image stops, so does the handling of interrupt routines. This might not always be desirable when debugging a real-time system.

9.9.2 Prerequisites

To use the RVI-GDB connection method, it is recommended that you compile your target application using a GNU toolchain for ARM architectures.

9.9.3 Procedure

If your application does not have GDB support linked-in, you can use the GDB protocol built into the debug hardware unit to debug your application. However, this controls the CPU directly, and the CPU stops whenever you want to examine its internal state.

To debug an application through a RVI-GDB connection:

1. Power-up your target hardware and debug hardware unit.

2. Configure the processor using rvconfig, using either automatic or manual configuration. Save the rvc file in a convenient location.
3. Run rvigdbconfig, specifying the rvc file that was created in step 2: **rvigdbconfig –f rvi.rvc**
4. Start GDB, load the symbols if required, and connect to the first processor (using port 4000 of debug hardware in this example):

```
arm-elf-gdb(gdb) file demo.elf
(gdb) target remote rvi5:4000
Remote debugging using rvi5:4000
0x00000000 in $a ()
(gdb)
```

   GDB is now connected to the processor, and an image can be loaded and debugged.

   Note

   To load and boot a complete system, use the `rviload` utility.

5. Set up any breakpoints or other debugging features, then run the application. Debug your application in the usual way.

9.9.4 See also

Tasks

- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *Connection methods for each debugging mode* on page 9-10
- *rvigdbconfig command syntax* on page 9-27.

9.10 Connections to a target with a GDB stub (Target-GDB)

These are connections to a target that is running an application with a GDB stub, and are referred to as Target-GDB connections. The GDB stub enables the target application to communicate with a host application through debug hardware, using the DCC of an ARM architecture-based processor. The DCC carries the GDB protocol packets between the target and the remote GDB session over the TCP/IP ports 5000, 5001,... as shown in the following figure:

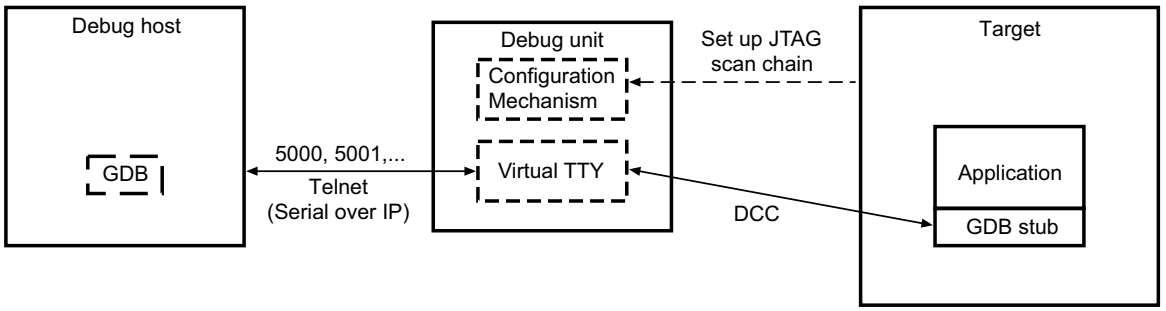


Figure 9-2 Target-GDB connections

9.10.1 Target-GDB Scenarios

Use the Target-GDB connection method to:

- debug a target system that does not have an OS
- debug a target system with an OS that supports GDB.

9.10.2 Prerequisites

To use the Target-GDB connection method, it is recommended that you compile the DCC driver and GDB stub into your target application using a GNU toolchain for ARM architectures. You can either:

- link the example GDB stub into your target application or operating system
- port your existing serial GDB stub to use the DCC driver.

Note

On the GDB connection to the target, it is recommended that you enable DCC and Virtual Ethernet/TTY mode before starting the processor.

9.10.3 Procedure

If your application includes a target-resident GDB stub, it can communicate over DCC.

To debug an application using a Target-GDB connection:

1. Power-up your target hardware and debug hardware unit.
2. Configure the processor using rvconfig, using either automatic or manual configuration. Save the `rvc` file in a convenient location.
3. Run rvigdbconfig, specifying the `rvc` file that was created in step 2, and the appropriate DCC mode, for example mode 2: **rvigdbconfig –f rvi.rvc -d 2:2**
4. Start GDB, load the symbols if required, and connect to the second processor (for example) to load and run your application in the usual way. This example uses port 4001 of debug hardware:

```
arm-elf-gdb(gdb) file demo.elf
(gdb) target remote rvi5:4001
Remote debugging using rvi5:4001
0x00000000 in $a ()
(gdb)
(gdb) load demo.elf
Loading section .vectors, size 0x30 lma 0x0
Loading section .text, size 0x1dbcc lma 0x8000
Loading section .rodata, size 0x1bcb4 lma 0x25bcc
Loading section .data, size 0xc84 lma 0x41980
Start address 0x8000, load size 238900
Transfer rate: 106177 bits/sec, 318 bytes/write.
(gdb)
(gdb) c
Continuing.
```

5. Start another GDB session to debug the image in the usual way, using (in this example) port 5001, the first available port of debug hardware:

```
(gdb) set remotetimeout 10
(gdb) file myprogram
(gdb) target remote rvi5:5001
```

Note

You only have to perform steps 1 to 3 once at the start. You can perform steps 4 and 5 as often as required.

9.10.4 See also

Tasks

- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

Reference

- *Connection methods for each debugging mode* on page 9-10
- *rvigdbconfig command syntax* on page 9-27.

9.11 Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)

If your target application requires TCP/IP communication with the debug host, you can connect to the target using Virtual Ethernet/TTY mode. Connections of this type are referred to as Target-GDB-Virtual Ethernet connections. This method is an extension to that used for connections to a target running an application with a GDB stub. The following figure shows an example:

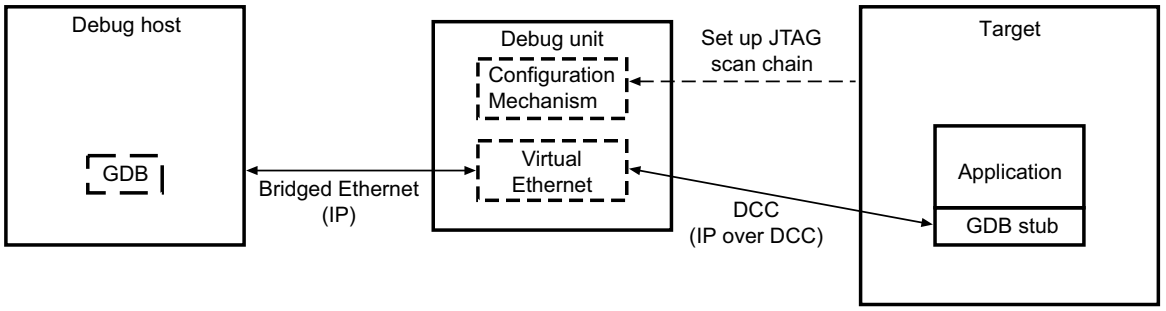


Figure 9-3 Target-GDB-Virtual Ethernet connections

In this method, debug hardware provides a network bridging feature to targets, and enables a target with only a JTAG connection to debug hardware to have access to the same network resources available to debug hardware. This works by intercepting IP packets on the network and examining them, and those packets that are addressed to the target are then sent over DCC alongside the normal GDB protocol. A driver is required on the target to interface the DCC channel to the protocol stack of the target, making the bridged network connection appear as an Ethernet device on the target. IP is the only network layer protocol supported.

Note

To reduce the load on the DCC and JTAG connection, broadcast packets are not sent to the target.

9.11.1 Target-GDB-Virtual Ethernet Scenario

Use the Target-GDB-Virtual Ethernet connection method to communicate with a standalone application that has a TCP/IP stack. For example, an application might provide a web server that serves web pages to the host.

9.11.2 Procedure

To use the Target-GDB-Virtual Ethernet connection method:

- It is recommended that you compile the DCC driver and GDB stub into your target application using a GNU toolchain for ARM® architectures. The DCC driver is available as a Linux OS download from the ARM products and solutions website.

  Note

  On the GDB connection to the target, you must enable DCC and Virtual Ethernet/TTY mode before starting the processor.

- The target application must be running a TCP/IP stack.

- debug hardware acts as a network bridge between the target processor and the host PC using a virtual Ethernet link. The target must have its own IP address that is either fixed or obtained from a DHCP server, and that appears on the virtual Ethernet as an independent host.

9.11.3 See also

Tasks

- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

Concepts

- *Connection methods for each debugging mode* on page 9-10
- *Setting DCC parameters* on page 9-23.

Other information

- ARM downloads, `http://www.arm.com/products`

9.12 Connections to a target OS using gdbserver (GDBserver)

If your target application requires TCP/IP communication with the debug host, you must connect to the target using bridged Ethernet. Connections of this type are referred to as GDBserver connections. This method is an extension to that used for connections to a target running an application with a GDB stub. The following figure shows an example:

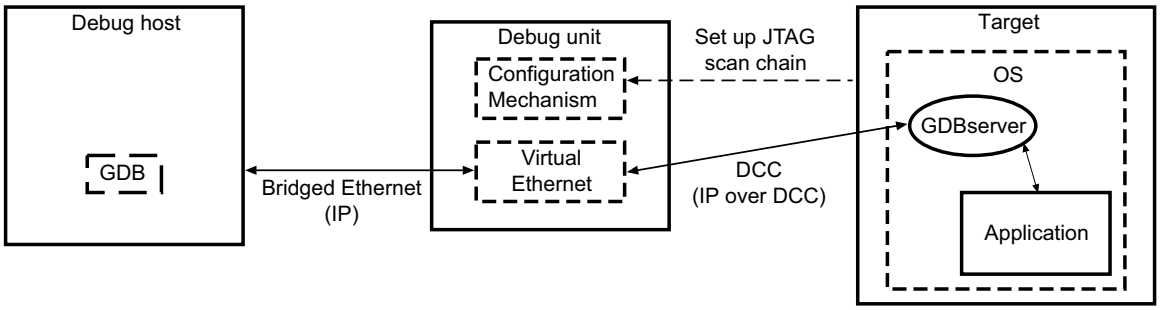


Figure 9-4 GDBserver connections

In this method, IP packets can be carried over the same link alongside the normal GDB protocol.

9.12.1 GDBserver Scenario

Use the GDBserver connection method to debug an application on a target that has an embedded OS, such as Linux, that supports independent processes. In this case you can run GDB server (gdbserver) instances. The GDB server can have TCP/IP connections to the debug host that is running GDB. The DCC driver is available as a Linux OS download from the ARM web site.

9.12.2 Prerequisites

To use the GDBserver connection method:

- On the GDB connection to the target, you must enable DCC and Virtual Ethernet/TTY mode before starting the processor.
- The target OS must be running a TCP/IP stack and gdbserver.
- debug hardware acts as a network bridge between the target processor and the host PC using a virtual Ethernet link. The target must have its own IP address that is either fixed or obtained from a DHCP server, and that appears on the virtual Ethernet as an independent host.

9.12.3 Procedure

If your application uses an IP stack, it can communicate over DCC through a bridged Ethernet connection.

To debug an application using a GDBserver connection:

1. Power-up your target hardware and debug hardware unit.
2. Download and boot the target using the `rviload` utility.
3. When the Linux kernel has finished booting, start the gdbserver as follows:

   ```
   ~ # gdbserver localhost:portnum filename
   ```

The TCP/IP port number you specify here is the port number you must use with the GDB `target remote` command from subsequent GDB sessions. Also, make sure the port number is not in use by another service.

9.12.4 See also

Tasks

- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using NFS (GDB-NFS)* on page 9-19.

Concepts

- *Connection methods for each debugging mode* on page 9-10
- *Setting DCC parameters* on page 9-23
- *Loading and booting a complete system* on page 9-26.

Other information

- ARM downloads, `http://www.arm.com/products`

9.13 Connections to a target OS using NFS (GDB-NFS)

This is useful for developing software on deeply embedded systems, and also for debugging brand new targets where only the CPU and RAM are initially known to work. Connections of this type are referred to as GDB-NFS connections. The following figure shows the GDB-NFS connection method:

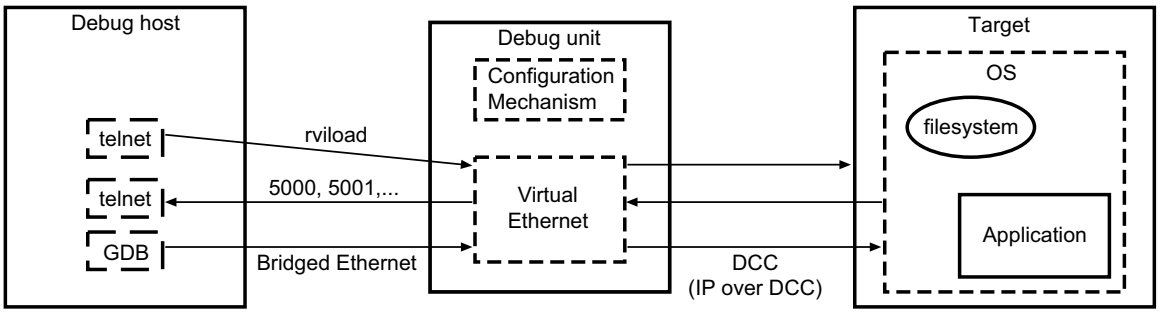


Figure 9-5 GDB-NFS connections

For example, the minimum I/O support that Linux requires is a system console and a root file system. You can connect the console to a GDB command-line console and, with the appropriate driver, you can mount the root file system over NFS with debug hardware acting as a bridge. Alternatively, you might have a file system and kernel loaded into memory and booted, and an NFS file system mounted to be shared later.

Note

This network connection is not as fast as an office LAN, because of the limited bandwidth of DCC.

9.13.1 Procedure

If your target has a complete operating system, you can mount a file system over NFS. In this case, debug hardware acts as a bridge.

To debug a target using a GDB-NFS connection:

1. Power-up your target hardware and debug hardware unit.
2. Load the Linux kernel and uBoot image using `rviload`. For example:

   ```
   ./rviload --host=rvi5 -j1000000 -mVEC -a0 7fc0:uImage 1000000:u-boot.bin
   ```

   Press Return. Messages appear showing u-boot loading and running the Linux kernel.

   Note

   This is similar for `rviload.exe` for MSDOS users.
3. Mount the directory exported by NFS using the following command:

   ```
   ~ # mount -t nfs -n client_IP_address:/exported_directory /mnt
   ```

9.13.2 See also

Tasks

- *Connections to a target without built-in GDB support (RVI-GDB)* on page 9-11
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13

- *Connections to a target GDB stub using Virtual Ethernet/TTY mode (Target-GDB-Virtual Ethernet)* on page 9-15
- *Connections to a target OS using gdbserver (GDBserver)* on page 9-17.

Concepts

- *Connection methods for each debugging mode* on page 9-10
- *Loading and booting a complete system* on page 9-26
- *rviload command syntax* on page 9-28.

9.14 Preparing your debug hardware for remote GDB connections

To prepare debug hardware to accept remote GDB connections and be able to communicate with a GDB session, you must:

1. Create a debug hardware configuration file for your develpoment system.
2. Use the `rvigdbconfig` command-line utility and specify:
   - the debug hardware configuration file you created in step 1
   - DCC parameters, if necessary.
3. Use GDB to:
   - load the image symbols if required
   - connect to the processor.

   You might have to specify a port number if your system has multiple processors.

9.14.1 See also

Tasks

- *Connecting to targets from GDB through debug hardware* on page 9-22
- *Setting DCC parameters* on page 9-23.

Concepts

- *About configuring debug hardware for debugging with GDB* on page 9-3
- *Methods of connecting from remote GDB sessions* on page 9-9
- *Loading and booting a complete system* on page 9-26
- *Multiprocessor debugging with GDB and debug hardware* on page 9-34.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7
- *rvigdbconfig command syntax* on page 9-27.

9.15 Connecting to targets from GDB through debug hardware

To connect to a target from GDB, you must configure debug hardware to recognize your target devices. You do this by using rvigdbconfig to configure debug hardware with the scan chain details. GDB is then connected to the processor.

To connect to a target from GDB:

1. Power-up your target hardware and debug hardware unit.
2. Start the Debug Hardware Config utility, and configure the processor.
3. Run rvigdbconfig, specifying the appropriate `rvc` file:

   ```
   rvigdbconfig -f rvi.rvc
   ```

4. Start GDB, load the symbols if required, and connect to the first processor (using port 4000 of your debug hardware in this example):

   ```
   arm-elf-gdb(gdb) file demo.elf
   (gdb) target remote rvi5:4000
   Remote debugging using rvi5:40000x00000000 in $a ()
   (gdb)
   ```

   GDB is now connected to the processor, and an image can be loaded and debugged.

9.15.1 See also

Tasks

- *Starting the debug hardware configuration utilities* on page 2-3
- *Setting DCC parameters* on page 9-23.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7
- *rvigdbconfig command syntax* on page 9-27.

9.16 Setting DCC parameters

Ethernet bridging works by examining incoming packets at debug hardware, then deciding which are destined for debug hardware itself and which are destined for the target. To do this, debug hardware must know the IP address, subnet mask and default gateway parameters for the target. These parameters are normally determined through DHCP, where the target asks for a configuration, and one is supplied by a server over the network. In this case, debug hardware is able to intercept the incoming DHCP packet containing the parameters and configure itself appropriately. It is, however, possible to configure a target with a static IP address. In this case there is no DHCP transaction to intercept, and debug hardware has no way of determining the target configuration. You must set these parameters in debug hardware for correct operation.

You can configure DCC Ethernet bridging with the `rvigdbconfig` command, and you must set the appropriate parameter when using DCC mode.

If you use `rviload`, you must set the DCC mode to either VEC or VEP.

Note

When Ethernet bridging is running, normal LAN services are accessible (including DHCP and NFS).

9.16.1 Examples of setting DCC parameters

```
rvigdbconfig -f rvi.rvc -d 1:2
```

sets device 1 (the first device on the scan chain) to DCC mode 2.

Additional devices are configured in a similar way. For example:

```
rvigdbconfig -f rvi.rvc -d 1:2 -d 2:2
```

configures devices 1 and 2 to mode 2.

The IP parameters for static IP configurations are set up in the following way:

```
rvigdbconfig -f rvi.rvc -d 1:2 -s 1:10.0.0.10:255.255.255.0:10.0.0.1
```

This configures device 1 to use DCC mode 2 with IP address `10.0.0.10`, subnet mask `255.255.255.0`, and default gateway `10.0.0.1`. This format can be used to configure multiple processors if required. For example:

```
rvigdbconfig -f rvi.rvc -d 1:2 -d 2:2 -s 1:10.0.0.10:255.255.255.0:10.0.0.1 -s
2:10.0.0.11:255.255.255.0:10.0.0.1
```

9.16.2 See also

Tasks

- *Configuring a target processor for virtual Ethernet* on page 5-51
- *Connections to a target with a GDB stub (Target-GDB)* on page 9-13
- *Connecting to targets from GDB through debug hardware* on page 9-22.

Concepts

- *DCC and interrupts* on page 9-25.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7
- *rvigdbconfig command syntax* on page 9-27

- *rviload command syntax* on page 9-28.

9.17 DCC and interrupts

The use of DCC interrupts has significant speed implications when using Virtual Ethernet/TTY mode. If possible, you must tie DCC interrupts into the interrupt system of the target and be able to enable and disable the read and write interrupt individually.

Note

This is a configuration item that you must select when configuring the kernel, when using an ARM driver.

debug hardware uses JTAG to control debug operations, and JTAG is used to send and receive data over DCC. debug hardware polls the target JTAG for status:

- If interrupts are used, the target is interrupted when data is written to the DCC register or read from it. This enables the target to deal quickly with the data, and continue normal processing.
- If interrupts are not available, the target must regularly poll the DCC register for any new data. This means that the target wastes time checking the register for data when none is present. Subsequent data is only discovered at the next poll.

If debug hardware finds that there is data to be transferred into or out of DCC, it attempts to transfer as many words as possible in one burst, up to a predefined limit. However, if the target has not sent more data or emptied its transfer register, debug hardware breaks out of its burst and begins polling the execution status and DCC.

9.17.1 See also

Tasks

- *Connecting to targets from GDB through debug hardware* on page 9-22
- *Setting DCC parameters* on page 9-23.

Reference

- *Debug hardware TCP/IP port numbering* on page 9-6
- *DCC modes* on page 9-7.

9.18 Loading and booting a complete system

You can load and boot a complete system at a command-line without having to run GDB or a supported debugger to control the debug hardware unit.

To load and boot a complete system at the command-line use the following commands:

- `rviload`
- `RVIahbload`
- `RVIvec`.

9.18.1 See also

Tasks

- *Preparing your debug hardware for remote GDB connections* on page 9-21.

Concepts

- *About configuring debug hardware for debugging with GDB* on page 9-3
- *Methods of connecting from remote GDB sessions* on page 9-9
- *Multiprocessor debugging with GDB and debug hardware* on page 9-34.

Reference

- *rviload command syntax* on page 9-28
- *RVIahbload command syntax* on page 9-30
- *RVIvec command syntax* on page 9-32.

9.19 rvigdbconfig command syntax

`rvigdbconfig` enables you to configure your debug hardware for debugging with GDB.

9.19.1 Syntax

`rvigdbconfig [-h] [-v]  [-d DEVICE:MODE] [-s DEVICE:IP-CONFIG] [-f RVC-FILE]`

`-d DEVICE:MODE`

Set the DCC mode for the device *DEVICE* to *MODE*, where *MODE* can be one of:

0 None (raw DCC)

1 Redirected raw DCC

2 Virtual Ethernet over DCC.

`-f RVC-FILE` Load scan chain configuration from *RVC-FILE*.

`-h` Display the command help.

`-s DEVICE:IP-CONFIG`

Set the DCC mode for the device *DEVICE* to *MODE*, where *IP-CONFIG* is of the form *ip-address:subnet-mask:default-gateway*.

`-v` Print progress messages.

9.19.2 Examples

The following are examples of how to use `rvigdbconfig`:

rvigdbconfig –f rvi.rvc -d 1:2

Sets device 1 (the first device on the scan chain) to DCC mode 2.

rvigdbconfig –f rvi.rvc -d 1:2 -d 2:2

Configures devices 1 and 2 to mode 2.

rvigdbconfig –f rvi.rvc –d 1:2 –s 1:10.0.0.10:255.255.255.0:10.0.0.1

Sets up the IP parameters for static IP configurations. This configures device 1 to use DCC mode 2 with IP address `10.0.0.10`, subnet mask `255.255.255.0`, and default gateway `10.0.0.1`.

rvigdbconfig –f rvi.rvc –d 1:2 –d 2:2 –s 1:10.0.0.10:255.255.255.0:10.0.0.1 –s 2:10.0.0.11:255.255.255.0:10.0.0.1

Sets up IP parameters for multiple processors.

9.19.3 See also

Tasks

- *Setting DCC parameters* on page 9-23.

Reference

- *DCC modes* on page 9-7.

9.20 rviload command syntax

`rviload` enables you to load a given binary file onto the target board through a specified debug hardware unit.

9.20.1 Syntax

`rviload [options]... address:file [address:file]...`

address:*file* A raw binary file *file* to be loaded at the target memory address *address* in hexadecimal.

`--autoconfig=DELAY`

Autoconfigure the scan chain and then wait the number of seconds specified by *DELAY*.

Option synonym: `-a DELAY`

`--check` Check memory as it is being written.

Option synonym: `-c`

`--devnum=DEVNUM`

The JTAG scan chain device number (default 1). Device 0 refers to the debug hardware unit, so you can only specify devices greater than 0.

Option synonym: `-d DEVNUM`

`--help` Display the command help.

Option synonym: `-h`

`--host=HOST`

The host IP address/name of the debug hardware unit.

Option synonym: `-H HOST`

`--jtagclock`

The JTAG clock speed in Hz, 0==‘RTCK’ (default 10MHz).

Option synonym: `-s`

`--jump=JUMPTO`

Start executing from this (hex) address after loading.

Option synonym: `-j JUMPTO`

`--dccmode=MODE`

Enable debug communications in the particular `MODE`, and must be one of the following:

DCC Raw DCC through client. Raw (unprocessed) data exactly as it exited the DCC register on the target, and is fed to the client.

DCP Raw DCC through TCP/IP port range from 5000. Raw data fed to TCP/IP port in the port range specified.

VEC Virtual Ethernet/TTY with tty channel through client.

VEP Virtual Ethernet/TTY with tty channel through port range from 5000.

The DCC mode for rviload is specified by using a three-letter mode name such as VEP or VEC, whereas the DCC mode for rvigdbconfig is specified by a mode number such as 0, 1 or 2.

Option synonym: `-m MODE`

`--page=PAGE`

The target memory page number.

Option synonym: `-p PAGE`

`--quiet` Do not print any messages.

Option synonym: `-q`

`--rule=RULE`

Target rule code.

Option synonym: `-r RULE`

9.20.2 Examples

To use the `rviload` utility from a Cygwin bash or Red Hat Linux shell, enter:

```
$ rviload [option]... address:file [address:file]...
```

For example:

```
rviload --host=192.168.1.200 -s0 -j7300000 -mVEC -a0 7300000:C:\DEMOS\
Linux_RVI_DCC\vp-boot.bin
```

9.20.3 See also

Tasks

- *Setting DCC parameters* on page 9-23.

Reference

- *RVIahbload command syntax* on page 9-30
- *RVIvec command syntax* on page 9-32.

9.21 RVIahbload command syntax

RVIahbload uses an access port that is bridged to the system bus, which might be *Advanced High-performance Bus* (AHB) or *Advanced Extensible Interface* (AXI). Although this direct route has speed improvements, there are additional compliations to consider. For example, loading is performed in the physical memory map.

You specify the files to load in the same way as for rviload, that is, by using `-bin addr:binary file` or you can specify ELF files which load to a fixed location (`--elf file.elf`). The debug unit to which you want to connect is specified by a `.rvc` file, and you cannot supply an address or hostname as in the case of rviload. When the file has downloaded, you can use the `-jump` option to start the target executing. For this to succeed, however, the device must support execution.

9.21.1 Syntax

`RVIahbload [options]`

`--bin address:file`

Load binary image file at specified (hex) address.

Option synonym: `-i address:file`

`--bus busnum`

Override the detected AHB bus to use for the download.

Option synonym: `-b busnum`

`--config RVCfile`

Provides the full path of the `.rvc` config file to use.

Option synonym: `-f RVCfile`

`--devnum device`

The device to use for the download. Default 1.

Option synonym: `-d device`

`--elf filename`

Elf file to load. Cannot be supplied at same time as `-i`.

Option synonym: `-e filename`

`--help` Display the command help.

Option synonym: `-h`

`--jump address`

Start executing from this (hex) address after loading if supported by device.

Option synonym: `-j address`

9.21.2 Examples

The following are examples of how to use RVIahbload:

RVIahbdownload -f rvi.rvc -i 8000:myprog.bin -d3 -b1 -j 0x8000

This loads the binary file myprog.bin to address 0x8000 on device 3 using bus 1 and starts executing it.

RVIahbdownload --file rvi2.rvc --elf my.elf

This loads the ELF file `my.elf` to the target, selecting the AHB bus by default.

9.21.3 See also

Reference

- *rviload command syntax* on page 9-28
- *RVIvec command syntax* on page 9-32.

9.22 RVIvec command syntax

RVIvec enables you to make a virtual Ethernet connection to a device. RVIvec uses a passive connection that it can connect to the device at the same time as a debugger or other application, as long as both applications use an identical .rvc file. The virtual Ethernet connection is then maintained, as long as RVIvec is still running, even after the debugger that started the device executing has quit.

9.22.1 Syntax

RVIvec [*options*]

--config *RVCfile*

Provides the full path of the .rvc config file to use.

Option synonym: -f *RVCfile*

--devnum *device*

The device to connect to. Default 1.

Option synonym: -d *device*

--help Display the command help.

Option synonym: -h

--jump *address*

Optional. Start processor from this (hex) address.

Note

This means that a passive connection cannot be used, so this option prevents a connection while a debugger or other application is running.

Option synonym: -j *address*

--mode *mode*

As with rviload, *mode* can be either VEC or VEP. The default is VEC.

Option synonym: -m *mode*

--verbose

Optional. This option means that RVIvec polls for any asynchronous messages that are returned from the debug hardware unit, and displays them to stdout.

Option synonym: -v

9.22.2 Examples

Typically, an image is first loaded onto a target (using rvd, rviload or rviahbload) and has started executing. For example:

```
RVIvec -f rvi.rvc

RVI virtual ethernet utility.Started channel 0x504344.Virtual ethernet enabled.Hit
return to quit.
```

9.22.3 See also

Reference

- *rviload command syntax* on page 9-28

- *RVIahbload command syntax* on page 9-30.

9.23 Multiprocessor debugging with GDB and debug hardware

Debug hardware is capable of simultaneously and synchronously debugging multiple targets. However, GDB does not support multiprocessor debugging directly.

Note

Although multiprocessor debugging is possible using GDB, it is recommended that you debug multiple processors using higher level tools.

9.23.1 How connections to multiple processors are allocated

When you make connections to multiple target processors, the connections on ports 5000, 5001,... are allocated by debug hardware, that is, redirected virtual DCC is allocated a port for each device in a similar way to GDB halt-mode debugging. The port range starts from port 5000, so virtual Ethernet or raw redirected DCC for the first device appears on port 5000, the second device on port 5001, and this continues in the same way.

9.23.2 Considerations when debugging multiple targets with GDB

Be aware of the following if you are debugging multiple targets with GDB:

- Multiprocessor debugging with GDB requires that you open multiple command windows (such as Xterms). You must have one GDB session for each target processor to which you want to connect.
- If you have multiple targets on the debug hardware scan chain, then:
  - the target processors are numbered consecutively, starting at one
  - the available bandwidth over DCC is shared between all target processors
  - communications to all target processors are through a single JTAG chain.

9.23.3 See also

Tasks

- *Preparing your debug hardware for remote GDB connections* on page 9-21
- *Loading and booting a complete system* on page 9-26.

Concepts

- *About configuring debug hardware for debugging with GDB* on page 9-3
- *Methods of connecting from remote GDB sessions* on page 9-9.

Chapter 10
Troubleshooting your debug hardware unit

If you encounter problems when attempting to connect to a debug hardware unit or upgrade the firmware, see the following topics:

10.1 Multiple programs attempting to scan

Only one program on each host computer can scan the TCP/IP network or USB ports for available debug hardware units. If another configuration utility is scanning, the following error message is displayed:

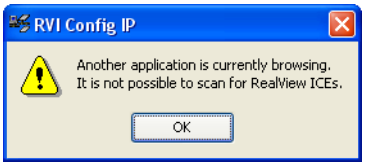


Figure 10-1 Error message when another program is scanning

You must stop the other configuration utility from scanning. To do this, click the **Scan** button, or select **Stop Scan** from the **RVI** menu in the configuration utility that you want to stop scanning.

10.1.1 See also

Tasks

- *Scanning for available debug hardware units* on page 2-4.
- *Connecting to a debug hardware unit* on page 2-7.

Concepts

- *USB server not accessible* on page 10-3
- *Connection times out* on page 10-4
- *Other active connections* on page 10-5.

10.2 USB server not accessible

The following error message is displayed if a communication error occurs between Debug Hardware Config and the USB server application:

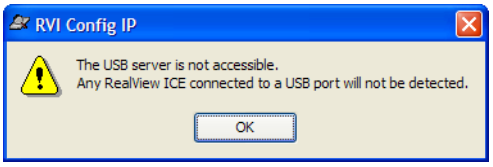


Figure 10-2 Error message when no USB devices present

The USB server might not be accessible because:

- The USB driver did not load. In this case you might have to reinstall the host software.
- The USB server application has stopped working. In this case, kill the USB server process and restart Debug Hardware Config.

Note

If you do not want to use a USB connection, then use the debug hardware unit over a TCP/IP connection. If you have not yet done so:

1. Connect the unit to your network.
2. Configure the network settings for the unit.

10.2.1 See also

Tasks

- *Determining the correct network settings* on page 3-3
- *Configuring the network settings for a debug hardware unit* on page 3-8.

Concepts

- *About configuring network settings* on page 3-2
- *Multiple programs attempting to scan* on page 10-2
- *Connection times out* on page 10-4
- *Other active connections* on page 10-5.

10.3 Connection times out

The default timeout for establishing a TCP/IP connection is five seconds. If you repeatedly get timeouts when attempting to connect to a debug hardware unit, you can change this setting. To do this:

1. Create the environment variable `RVI_COMMS_CONNECT_TIMEOUT` if it does not already exist.
2. Set the value of this variable to the timeout that you want, in seconds. This must be an integer in the range 0-120.

For details of how to create and set an environment variable, see the documentation for the operating system that is supplied with your host computer.

Note

Be aware that TCP/IP latency might affect the response times and performance of your debugger.

10.3.1 See also

Concepts

- *Multiple programs attempting to scan* on page 10-2
- *USB server not accessible* on page 10-3
- *Other active connections* on page 10-5.

10.4 Other active connections

If you connect to a debug hardware unit that has other active connections, the following error message is displayed:

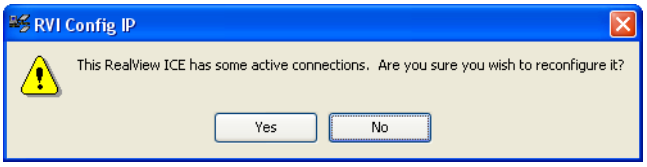


Figure 10-3 Error when other connections are active

If you continue, the changes that you make might interfere with the correct operation of these configuration utilities. Do one of the following:

- ensure that the other configuration utilities are disconnected, then click **Yes** to continue using the configuration utility
- click **No** to stop using the configuration utility, and try again later.

10.4.1 See also

Concepts

- *Multiple programs attempting to scan* on page 10-2
- *USB server not accessible* on page 10-3
- *Connection times out* on page 10-4.

10.5 A debug hardware unit is not listed

A debug hardware unit might not be listed in the debug hardware utility for the following reasons:

- The unit is on a different subnet to your PC. In this case, obtain either the IP address or the host name of the unit and enter the value in the IP Address / Host Name field of the utility you are using.
- The unit has not yet been configured for use on a network. Configure the network settings if you have provilege to do so. Otherwise, contact the person responsible for the unit.
- The unit did not boot correctly. Reboot the unit. If the unit is on the same subnet as your PC, it appears in the list of units when the reboot is successful.

10.5.1 See also

Tasks

- *Determining the correct network settings* on page 3-3
- *Configuring the network settings for a debug hardware unit* on page 3-8.

10.6 Auto Configure button is disabled in Debug Hardware Config

The **Auto Configure** button is disabled in Debug Hardware Config when you have a platform assigned to your debug hardware configuration.

If you want to autoconfigure the scan chain again:

1. Click **Clear Platform**.
2. Click **Auto Configure**.

10.6.1 See also

Tasks

- *Autoconfiguring a scan chain* on page 5-11
- *Clearing a platform assignment from a debug hardware configuration* on page 5-44.
- *Adding devices to the scan chain* on page 5-12.

10.7 Remove button is disabled in Debug Hardware Config

The **Remove** button is disabled in Debug Hardware Config when you have a platform assigned to your debug hardware configuration.

If you want to remove a device from the scan chain:

1. Click **Clear Platform**.
2. Either:
   - autoconfigure the scan chain again
   - manually add only those devices you want to keep.

10.7.1 See also

Tasks

- *Autoconfiguring a scan chain* on page 5-11
- *Clearing a platform assignment from a debug hardware configuration* on page 5-44.
- *Adding devices to the scan chain* on page 5-12.

10.8 Troubleshooting firmware upgrade installations

The main types of error that might occur during installation of a firmware patch are:

10.8.1 Version problems

A patch targets a particular *major.minor* release version of the software. It might contain:

- new components that are not in the targeted software
- updates to components that are already in the targeted software.

If there is a problem installing a patch, a dialog box appears to inform you of the problem:

- If the patch targets a version of the software that is not installed, the dialog box shown in the following figure appears:

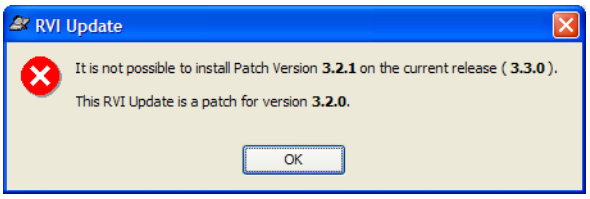


Figure 10-4 Error when installing a patch to uninstalled software

In this case the patch is not installed, and the software on the debug hardware unit remains unchanged. Make sure that you have the patch for the version of the firmware that you have installed.

- If the patch does not contain any new or updated components (typically because a later patch has already been installed), the dialog box shown in the following figure appears:

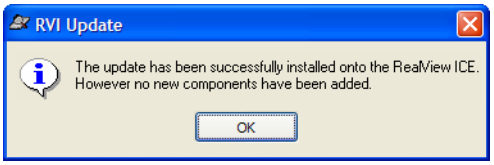


Figure 10-5 Message when installing a patch that has no new components

If you see one of these dialog boxes, click **OK**.

10.8.2 Errors during file operation on the host

If an error occurs during file operation on the host, a dialog box appears to inform you of the problem:

- If the error occurs before any data has been written to the compact flash, the dialog box shown in the following figure appears:

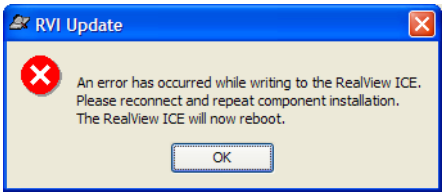


Figure 10-6 Error before data has been written to compact flash

Click **OK** and begin the installation again.

- If some data has already been written to the compact flash when the error occurs, the debug hardware unit must reboot to clean up the failed installation and revert to the backed up state. The dialog box shown in the following figure appears:

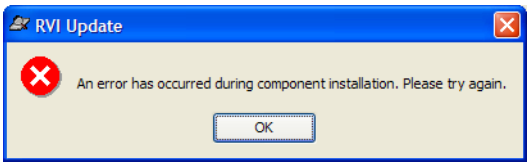


Figure 10-7 Error during writing to compact flash

Click **OK**. The debug hardware unit reboots. You can then begin the installation again.

10.8.3 See also

Tasks

- *Viewing software version numbers* on page 4-4.

Concepts

- *About templates and firmware files* on page 4-2.

10.9 Troubleshooting autoconfigation of a scan chain

When autoconfiguring a scan chain, you might see one of the following errors:

- If debug hardware detects any unpowered devices, it displays the error shown in the following figure:

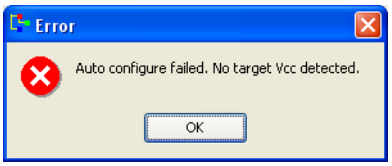


Figure 10-8 Error shown when unpowered devices are detected

  This error message can also display if the target is connected by the JTAG ribbon cable if debug hardware is started when the *Low Voltage Differential Signaling* (LVDS) probe is connected, or if the probe is connected and used after you started debug hardware.

  If you see this error:

  — Check the JTAG connection between the debug hardware unit and the target hardware.

  — Ensure that power is supplied to all your devices.

- If debug hardware cannot identify any devices, it displays the error shown in the following figure:

Figure 10-9 Error shown when no devices are detected

  If you see this error:

  — manually configure the scan chain if your target has unsupported devices

  — try auto-configuring again with a lower clock speed.

  Note

  You might have to power-cycle your target hardware when changing the clock speed.

- The Read ROM Table phase for a CoreSight system fails to find any devices. This might be because the ROM table is corrupt. Manually configure the scan chain.

- If communication cannot be made with the debug hardware unit in your current configuration, it displays the error shown in the following figure.

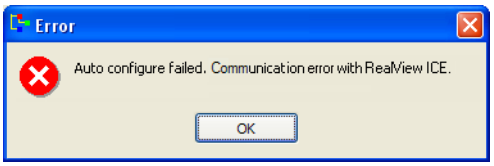


Figure 10-10 Error shown when there is no communication with debug hardware

  If you see this error, it might mean that the debug hardware unit in your configuration file no longer exists, or has been configured with different network settings.

10.9.1 See also

Tasks

- *Autoconfiguring a scan chain* on page 5-11
- *Adding devices to the scan chain* on page 5-12.

10.10 Log Client Utility

The Log Client utility receives logging messages from a DSTREAM or RVI unit. The unit can be connected over a network or USB.

Output from all logging processes is displayed.

10.10.1 Syntax

```
rvi_log_client rvi_unit_identifier
rvi_log_client -help
```

where *rvi_unit_identifier* is the identifier for the RVI unit can be in any of the following formats:

IP Address Specify either:

xxx.xxx.xxx.xxx

TCP:*xxx.xxx.xxx.xxx*

For example, **TCP:100.16.89.12**

Hostname Specify either:

rvi_unit_name

TCP:*rvi_unit_name*

For example, **my-rvi-unit**.

USB Specify either

USB

USB:*usb_port_number*

usb_port_number is the number shown in the tree control or the scan chain schematic diagram of the Debug Hardware Config utility, for example:

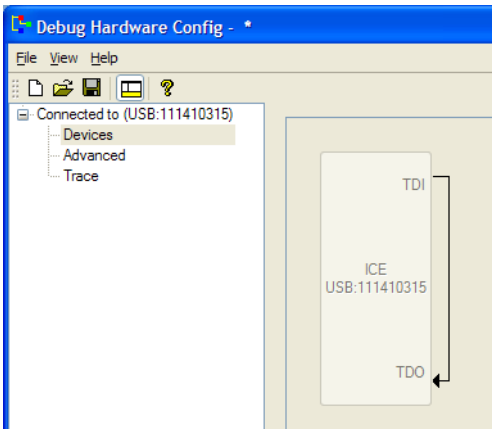


Figure 10-11 USB port number in the Debug Hardware Config utility

In this example, the port number is 111410315.

10.10.2 Usage

Specify -h or -help to display the help message and usage information.

Press Ctrl+C to close the Log Client utility.

Note

When using RVIUpdate, make sure that you close the Log Client before trying to update. If you leave the Log Client running while the update is in progress, the update takes longer than normal. However, the update still completes.

10.10.3 Example

The following example shows the logging output when connecting to an RVI unit and autconfiguring a scan chain that has a single ARM926EJ-S processor:

```
C:\Program Files\DS-5\bin>rvi_log_client USB:111410315

ARM RVI Log Client.
Copyright 2001, 2007 ARM Limited. All rights reserved.
V1.1.0 10-09-2007

Connected. Now building process list, please wait...
Process List built. Connected and waiting for logging.

usbd           info     CUsbClientCon: created udp listen port 30000
usbd           info     CUsbClientCon: created udp listen port 30000
serverd        error    CLVDSProbe::ReadEEPROM: Failed to read probe EEPROM (probably no probe plugged in)
serverd        error    CLVDSProbe::ReadEEPROM: Failed to read probe EEPROM (probably no probe plugged in)
rifsd          error    CRIFSCmdProc::CommandFILEOPEN open failed file(/real-ice/backup/_version_) errno=2 {No
such file or directory}
serverd        error    CLVDSProbe::ReadEEPROM: Failed to read probe EEPROM (probably no probe plugged in)
cfgqueryd      info     CJTAGDriver::DetectRTCLK did detect RTCLK
cfgqueryd      info     CFPGA_RVIHW::SetJTAGClock @ 10000000 Hz, not using RTCK
cfgqueryd      info     CClkCtrl::SetJTAGClkFreq requested freq=10000000; using freq=80000000 with FPGADiv=2
cfgqueryd      info     CJTAGDriver::DetectRTCLK did detect RTCLK
cfgqueryd      info     CFPGA_RVIHW::SetJTAGClock @ 50000000 Hz, using RTCK
cfgqueryd      info     CClkCtrl::SetJTAGClkFreq requested freq=50000000; using freq=100000000 with FPGADiv=0
cfgqueryd      info     CFPGA_RVIHW::SetJTAGClock @ 50000000 Hz, using RTCK
cfgqueryd      info     CClkCtrl::SetJTAGClkFreq requested freq=50000000; using freq=100000000 with FPGADiv=0
cfgqueryd      info     CRVIAutoDetect::MeasureDeviceCount measuring the length of DR chain in BYPASS
cfgqueryd      info     DR Chain 0..01..1 filled[1024]:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000011111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111110

cfgqueryd      info     CRVIAutoDetect::MeasureTotalDRLength Measured total DR length as 1
cfgqueryd      info     CRVIAutoDetect::MeasureDeviceCount Measured JTAG device count as 1
cfgqueryd      info     DR Chain after tap reset[32]: 00000111100100100110000001000001
cfgqueryd      info     Device 0 has IDCODE = 0x07926041
cfgqueryd      info     CRVIAutoDetect::deviceCountForIDCode device count for ID 0x07926041 is 1
cfgqueryd      info     CRVIAutoDetect::MeasureScanChainLength returned 1 devices
cfgqueryd      info     CFPGA_RVIHW::SetJTAGClock @ 50000000 Hz, using RTCK
cfgqueryd      info     CClkCtrl::SetJTAGClkFreq requested freq=50000000; using freq=100000000 with FPGADiv=0
cfgqueryd      info     CRVIAutoDetect::autodetect_config measuring the total IR length
cfgqueryd      info     IR Chain 1..10..0 filled[1024]:
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
11111111111111111111111111111111111111111111111111111111111111100000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000001

cfgqueryd      info     CRVIAutoDetect::MeasureTotalIRLength Measured total IR length as 4
cfgqueryd      info     CRVIAutoDetect::autodetect_config measured the total IR length = 4
cfgqueryd      info     CRVIAutoDetect::autodetect_config measuring the number of taps
cfgqueryd      info     CRVIAutoDetect::MeasureDeviceCount measuring the lengthof DR chain in BYPASS
cfgqueryd      info     DR Chain 0..01..1 filled[1024]:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000001111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111110

cfgqueryd      info     CRVIAutoDetect::MeasureTotalDRLength Measured total DR length as 1
cfgqueryd      info     CRVIAutoDetect::MeasureDeviceCount Measured JTAG device count as 1
cfgqueryd      info     DR Chain after tap reset[32]: 00000111100100100110000001000001
cfgqueryd      info     Device 0 has IDCODE = 0x07926041
cfgqueryd      info     AutoDetect: get_ir_data, Dev 0 IRLen=4
cfgqueryd      info     CRVIAutoDetect::autodetect_config measured the number of taps = 1
cfgqueryd      info     Device 0 detected as ARM926EJ-S from STMicroelectronics
cfgqueryd      info     DR Chain 0..01..1 filled[1024]:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000001111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1110000
cfgqueryd      info     CRVIAutoDetect::MeasureTotalDRLength Measured total DR length as 5
cfgqueryd      info     CRVIAutoDetect::get_scan_chain_length Detecting length of scan chain 6
cfgqueryd      info     DR Chain 0..01..1 filled[1024]:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000011111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
0000001

cfgqueryd      info     CRVIAutoDetect::MeasureTotalDRLength Measured total DR length as 40
^C
C:\Program Files\DS-5\bin>
```