Multi ICE™

Multi-processor EmbeddedICE™ Interface Unit

User Guide

Document Number: ARM DUI 0048A
Released: June1998

ENGLAND
ARM Ltd.
90 Fulbourn Road
Cambridge
CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY
ARM Ltd.
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN
ARM K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA
ARM, INC.
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

Proprietary Notice

ARM, the ARM Powered logo, EmbeddedICE, and TrackingICE are trademarks of ARM Limited.
Visual C++ is a trademark of Microsoft Corporation.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

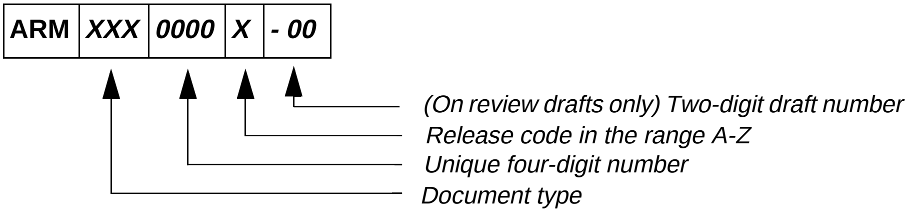
The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Ltd. shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.


ARM
XXX
0000
X
- 00
(On review drafts only) Two-digit draft number
Release code in the range A-Z
Unique four-digit number
Document type

Document Status

The document’s status is displayed in a banner at the bottom of each page.
This describes the document’s confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above, and to staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	June 1998	BJH/JM	First Release

Information on Electromagnetic Compatibility (EMC)

The Multi-ICE unit has passed tests for EMC compliance against the following standards:

- FCC part 15 Class A
- EN55022 Class A
- EN50082-1

In accordance with US FCC regulations, the following is provided as information to the user.

> Note: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.
>
> The user is cautioned that changes or modifications not expressly approved by the party responsible for compliance could void the user's authority to operate the equipment.

In accordance with the European standard for EMC compliance of Information Technology Equipment (EN55022) the following warning is stated here:

> **Warning**
>
> This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

Safety Compliance

The Multi-ICE unit complies with the basic safety requirements of EN60950 for Information Technology equipment. There are no user-serviceable parts inside the unit—any user interference may invalidate safety compliance.

Contents

1

Getting Started

This chapter describes the requirements for running Multi-ICE effectively. It covers:

- system requirements
- information on connecting the Multi-ICE hardware

1.1 System Requirements

1.1.1 Software requirements

You must be using one of the following operating systems to install and run Multi-ICE:

- Windows NT version 4.0 or higher
- Windows 95

If you wish to view the online manual—*Multi-ICE User Guide* (ARM DUI 0048)—you must have a copy of Adobe Acrobat Reader, which is provided on the CD-ROM.

Networking software

For the Multi-ICE server to run, your operating system must be installed with its supplied networking software. The Multi-ICE server cannot run without a TCP/IP stack. If this stack is not present, the following warning text is displayed:

```
TCP/IP protocol does not appear to have been set up on this computer.
Setup will continue, but the TCP/IP protocol will need to be
installed before the server or debugger can be used.
```

Automatic dialup

Please disable the automatic dialup facility in your PC’s Dialup Networking software. Automatic dialup is triggered by the presence of network packets; running Multi-ICE produces network packets which will trigger automatic dialup if this is enabled on the PC.

1.1.2 Hardware requirements

The following are the minimum recommended hardware requirements for installing and running the Multi-ICE server:

- 200MHz Pentium PC
- 32MB RAM for Windows 95; 64MB RAM for Windows NT
- CD-ROM drive (can be used across a network)
- VGA monitor, or better
- Parallel port
- Network card

Performance is based on two main factors:

- processor performance
- parallel port performance

Some newer machines have poor parallel port performance, which slows up the PC’s interfaces. If you have a machine that falls into this category, it is recommended that you fit a fast parallel port card. Limitations in performance will be noticed primarily when downloading code.

Disk space

If you wish to carry out a full installation of the software, 5MB of hard disk space is required.

The list of files installed in Multi-ICE is given in the *Multi-ICE Installation Guide* (ARM DSI 0005A) under section ***2, List of Installed Files***.

1.2 Connecting the Multi-ICE Hardware

A standard male-to-male 25-way parallel cable connects the Multi-ICE unit to the PC's parallel port.

The connection to the target board is made by a 20-way female IDC header cable (BT224 type) with all pins connected straight through (1-1, 2-2, ... 20-20). For further pin-out details, refer to ***Appendix A, JTAG Interface Connections***.

An adaptor is supplied to convert the 20-way header to the 14-way header used on existing ARM header cards. Newer ARM header cards use the 20-way Multi-ICE interface.

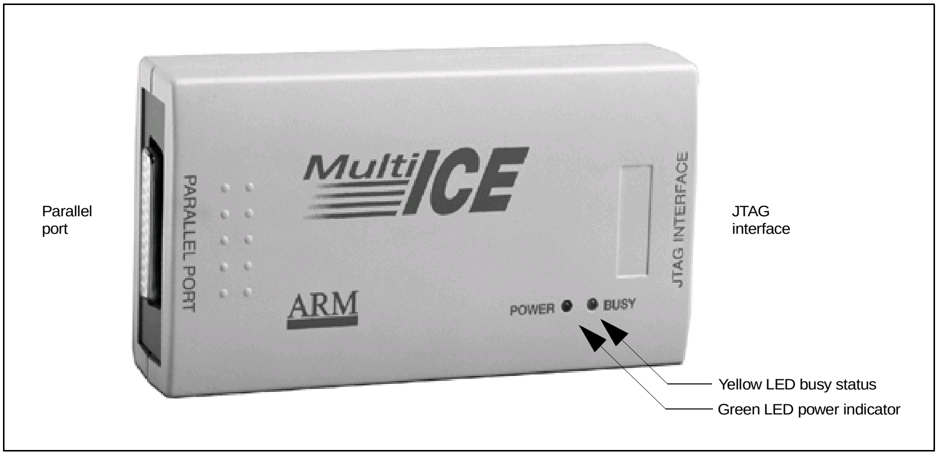


Figure 1-1: Multi-ICE unit

1.2.1 Multi-ICE power supply

Power is supplied to the Multi-ICE unit via pin 2 of the 20-way IDC connector. This is normally fed by the target `Vdd`.

Note *The target Vdd must not have a series resistor in the feed to pin 2.*

If the target supply voltage (minimum 2V@250mA) or its current capability is too low, an external power supply is required. You can calculate the approximate current using this formula:

$$\texttt{80mA} \times \frac{\texttt{5V}}{\texttt{targetV}}$$

If the 20-way to 14-way adaptor is being used, there is a jumper connection available for connecting an external power supply.

1.2.2 Compatibility with existing boards

For compatibility with existing ARM boards, the following modifications are needed. These modifications do not make the board incompatible with the EmbeddedICE interface unit.

The following must be shorted:

- resistor R1 on the ARM7TDMI header card (HHI-0016B) for the ARM Development Board (HBI-0011B)—only if modification box number 1 is not marked with ‘X’
- resistor R12 on the ARM PIV7T board (HBI-0008B)
- resistor R53 on the ARM PIE7 board (HBI-0004B)

To use Multi-ICE with Piccolo on an ARM Development Board, the following arrangement is also required:

- link 14 on the PID must be shorted between A and C, and a 1kΩ resistor inserted between B and C

Notes *(1) The ARM Development Board mentioned above is also known as the PID board.*

(2) If the ARM Development Board is not manually reset by using the red reset button, the user may occasionally experience problems loading and running an image. This is because power up does not always provide a clean reset when used with a PC power supply.

For details on loading an image, see ***4.4.1 Loading an image*** *on page 4-11, and for details on running an image, see* ***4.9.2 Running an image*** *on page 4-41.*

2

Multi-ICE Concepts

This chapter describes how Multi-ICE uses configuration data and interacts with devices and debug software. It details the differences between Multi-ICE and EmbeddedICE and also explains how Multi-ICE remains synchronous with all target devices.

2.1 Introduction

Multi-ICE is the latest EmbeddedICE compatible debug solution from ARM. It comprises:

- an interface unit which connects between the parallel port of a PC and the JTAG interface of an ASIC with debug and EmbeddedICE capability
- software to allow an ARM debugger to communicate with the interface unit. The software has 2 major components:
  - the Multi-ICE Server
  - a DLL for the *Multi-processor Debugger for Windows (MDW)*

Multi-ICE is specifically designed to be used with multiple ARM and ARM+DSP/Piccolo ASICs, but single cores are also supported.

This document assumes you have some knowledge of JTAG technology. If you need more information on JTAG, please refer to the *IEEE Standard 1149.1* on JTAG technology.

2.1.1 Features of Multi-ICE

The main features of Multi-ICE can be summarized as follows:

- debugger connections to multiple cores on the same ASIC
- stored or automatic configuration
- networked connections
- rapid code download
- low-voltage target interface operation
- remains synchronized with cores using “sleep” modes
- small, lightweight unit improves ease of use
- additional power supply not normally required (powered from target)
- simple parallel port connection
- software-configurable JTAG rate to support slow- or variable-speed clock devices
- easy software upgrades—100% host-based software
- designed to integrate with third-party elements (for example, non-ARM DSPs) by providing open interfaces for user-supplied debuggers
- extra user input/output bits for user-supplied drivers

2.1.2 Comparing Multi-ICE and EmbeddedICE

The Multi-ICE system is similar to EmbeddedICE interface unit system except that, whereas the EmbeddedICE interface unit contains a processor and software, the Multi-ICE interface unit contains no processor and the interface software runs on the host PC. There are many advantages to this approach. These include:

- a smaller, lighter interface unit with a lower power consumption (Multi-ICE is normally target powered)
- easier updating of software
- more flexible interfaces

2.1.3 Multi-ICE Server

The Multi-ICE Server provides virtual connections to single processors on a multi-processor ASIC and manages the interdependencies.

The Multi-ICE Server runs on a PC with the Multi-ICE interface unit attached, (this is then known as the Server PC). The Server is responsible for driving the hardware through the PC’s parallel port, using a supplied low-level driver. The Server also exposes and manages virtual connections—via an RPC (remote procedure call) interface—to single processors on the target ASIC that debuggers use to access individual processors.

The debugger connects to a single processor through a single connection to the Server, and has no knowledge of other processors on the ASIC or other debuggers. This is achieved using a configuration file that defines the content of the ASIC. If the attached ASIC contains only ARM processors, an Auto-configure feature on the Server can write the configuration file with no user intervention.

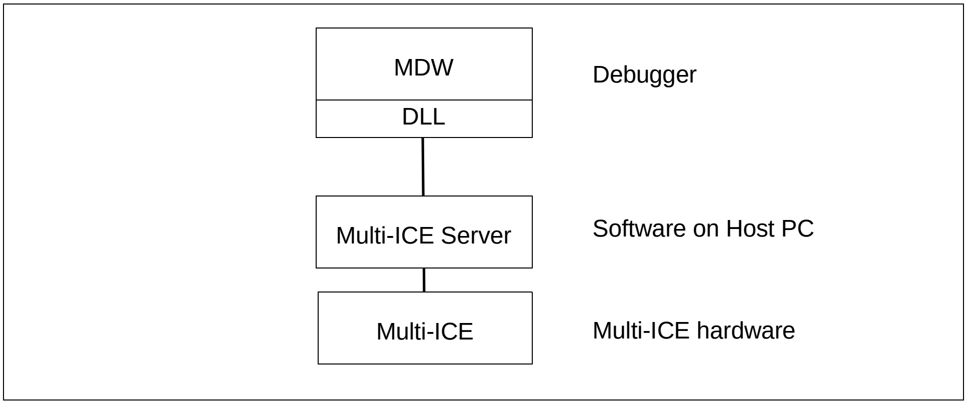


Figure 2-1: Client-Server relationship

2.1.4 Multi-ICE DLL for MDW

Multi-ICE is supplied with the *Multi-processor Debugger for Windows (MDW)*, which is a multi-processor version of ADW.

The Multi-ICE DLL replaces the `remote_a` / `remote_d` DLL used by the *ARM Debugger for Windows (ADW)* and connects MDW to the Multi-ICE Server using *Remote Procedure Calls (RPC)*. This means that the debugger can be run on the same machine as the Server or on another machine on the same network as the Server PC. It is even possible to connect a debugger through a modem, but this is extremely slow.

The transport mechanism used is TCP (the machines must be set up to provide TCP/IP network services). This is true even when used locally, as the RPCs are still used for the connection between the debugger and Server. In this case, `localhost` is used as the Server location and Windows routes the packets between the two applications. The DLL interrogates the Server to find out what the ASIC contains and the user can then choose which processor to connect to each debugger.

2.2 Multi-ICE Configuration Data

Multi-ICE requires configuration information about the devices on the ASIC; it needs this data to direct debugger calls to the correct device, independent of any other devices on the ASIC. This is achieved by manipulating the contents of the TAP controllers' *Instruction Registers (IR)*. Multi-ICE needs to know the length of the IR register for each device, and already has configuration data on all ARM devices, which are held in a supplied file called `IRlength.arm`. This defines the length of IR registers for ARM TAP controllers when connected to ARM devices.

There are two ways to create configuration information:

- automatically, using the menus on the Multi-ICE Server
- manually, by creating a text file containing configuration data and loading it into the Multi-ICE Server

2.2.1 Automatic configuration

If all cores are ARM cores, Multi-ICE automatically creates the configuration file by scanning the ASIC, referring to the `IRlength` file, and writing back data about the devices it finds.

Note *Autodetection of 710T/720T/740T sometimes gives UNKNOWN, as the processor cannot be stopped before reading CP15 r0. This can be prevented by resetting the processor.*

2.2.2 Manual configuration

A configuration file is a text file with the file suffix `.cfg`. You can store the file anywhere on your machine; you are prompted for its name during the configuration.

The configuration file contains information on:

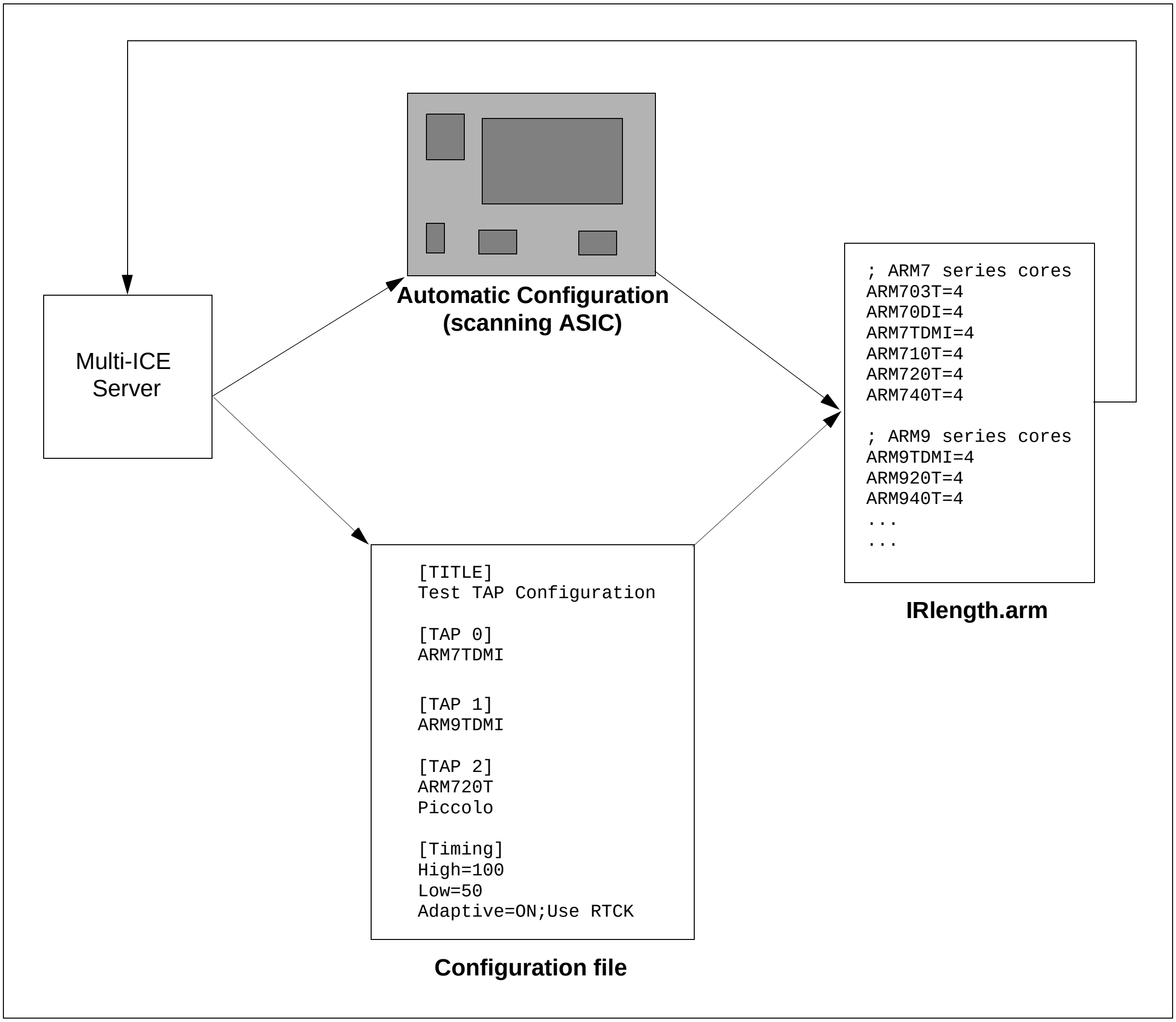
- TAP controllers on the ASIC
- devices attached to each TAP controller
- an optional title for the configuration
- optional JTAG timing information
- options to control which operations are performed during configuration

2.2.3 Relationship between Multi-ICE configuration items

Figure 2-2: Relationship between configuration information shows how Multi-ICE uses configuration files to configure an ASIC.

1. Multi-ICE either scans the ASIC or reads a configuration file to gather the names of the devices being used.
2. It then refers to the `IRlength` file and uses these device names to look up the length of the IR register for each device.

With this information, and any optional timing parameters, Multi-ICE can configure the Server and draw a pictorial representation of the devices.



```
; ARM7 series cores
ARM703T=4
ARM70DI=4
ARM7TDMI=4
ARM710T=4
ARM720T=4
ARM740T=4

; ARM9 series cores
ARM9TDMI=4
ARM920T=4
ARM940T=4
...
...
```

```
[TITLE]
Test TAP Configuration

[TAP 0]
ARM7TDMI


[TAP 1]
ARM9TDMI

[TAP 2]
ARM720T
Piccolo

[Timing]
High=100
Low=50
Adaptive=ON;Use RTCK
```

Figure 2-2: Relationship between configuration information

2.2.4 The IRlength file

Information on ARM devices is held in the `IRlength` file in the Multi-ICE directory. This file contains a definition of the number of bits in the TAP controller's IR register for each core. Using this list, Multi-ICE can create a virtual connection to each device.

Here is an extract from the supplied `IRlength` file:

```
; ARM7 series cores
ARM70DI=4
ARM7TDMI=4
ARM7TDMI-S=4
ARM710T=4
ARM720T=4
ARM740T=4

; ARM9 series cores
ARM9TDMI=4
ARM920T=4
ARM940T=4

; Other devices
Piccolo=0   ;Shares its TAP controller with the host processor
```

Counting IR bits

The IR register is accessed serially through the ASIC's TDI and TDO pins so the register width (number of bits) is described as the register length. This information is used to calculate the total number of bits that must be shifted from TDI through TDO for multiprocessor ASICs that have the TAP controllers serially chained.

For example, if three ARM7TDMI cores (each with their own TAP controllers) are fabricated onto a single ASIC, the TAP controllers have their TDI-TDO signals serially chained. The four bits of each IR register appear in series, giving a total chain length of 12 bits.

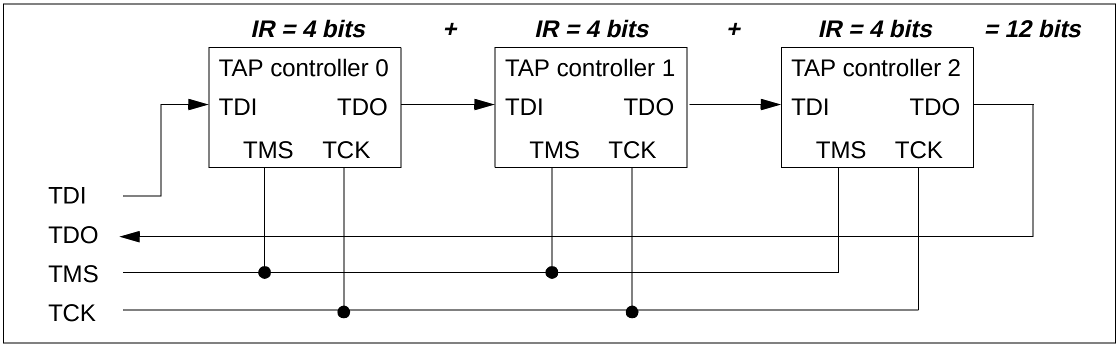


Figure 2-3: Serial chaining of TAP controllers

Shared TAP controllers

A value of 0 indicates that the IR is shared with another processor; two or more devices can share one TAP controller.

For example, ARM's Piccolo DSP coprocessor can only be used in conjunction with a host processor. It shares the host's TAP controller so the number of bits that must be scanned through the TAP when the IR is selected, is no more than for the processor alone.

```
; Other devices
Piccolo=0   ;Shares its TAP controller with the host processor
```

Furthermore, the IR length is determined by the host processor entry in this file, so any IR length for a Piccolo has no meaning.

2.2.5 Configuring non-ARM devices

Manual configuration

Any drivers that are referenced in the configuration file have to be present in the `IRlength` file, or Multi-ICE displays an error when the file is loaded.

User-defined devices can be accommodated if they are named in the `IRlength` file, and the device referenced in the configuration file.

For example, it you want to add a DSP (with its own TAP controller) to the scan path, you must make an entry at the end of the `IRlength` file to define the IR length of the DSP's TAP controller:

```
; Other devices
DSP=5
```

This entry means the driver called DSP requires five extra bits in the scan chain in order to bypass the DSP's TAP controller when the debugger wants to access the ARM devices.

The driver name 'DSP' can now be used in the configuration file without error.

Automatic configuration

If you use the automatic configuration facility and a non-ARM device is detected, the Server looks for a file called `UserdrvN.txt` in the Multi-ICE directory (where `N` is a single-digit number that gives the length of the IR register). This text file contains a single word (the device name), and is displayed on the Server in the box for that TAP controller. This name is also exported to the debugger so that a connection can be made to the device.

If there is no file for the IR length found, UNKNOWN is displayed.

For example, a device called NEWCHIP has an IR length of 6. The text file is called `Userdrv6.txt` and contains only the word `NEWCHIP`.

This method works if there is only one unknown device with the same `IRlength` on an ASIC.

2.3 Debugging Using Multi-ICE

Figure 2-4: Core debugging session gives a pictorial overview of how you can debug multiple cores in one debugging session.

The Server can be accessed by a debugger running on the same machine (`localhost`), or by remote users.

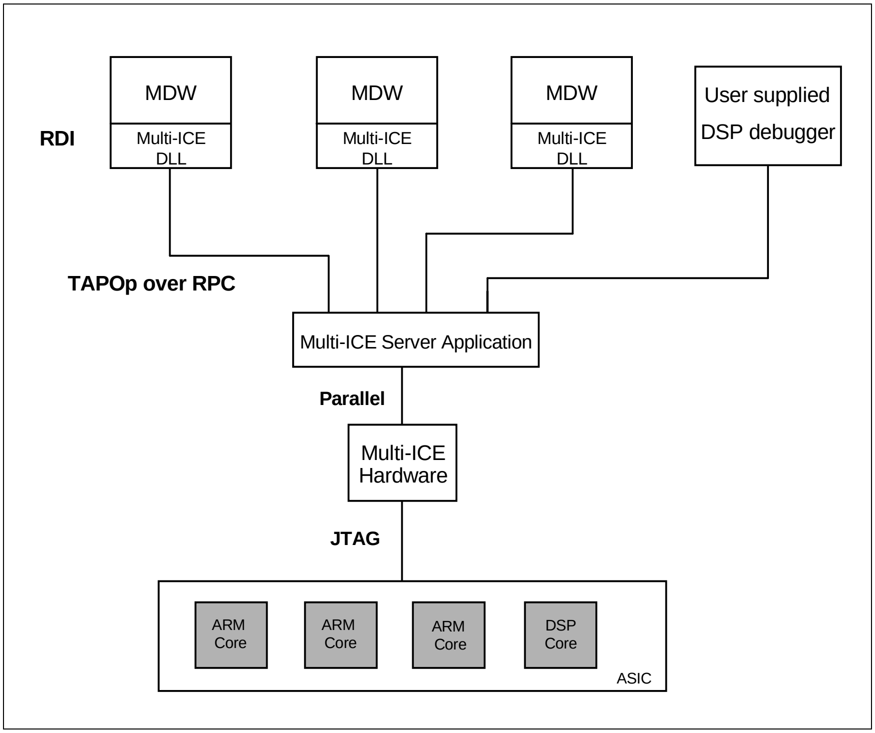


Figure 2-4: Core debugging session

2.4 Adaptive Clocking

Adaptive Clocking is a feature which ensures that Multi-ICE never loses synchronization with the target device, whatever the clock speed.

To achieve this, Multi-ICE uses two signals, **TCK** and **RTCK**.

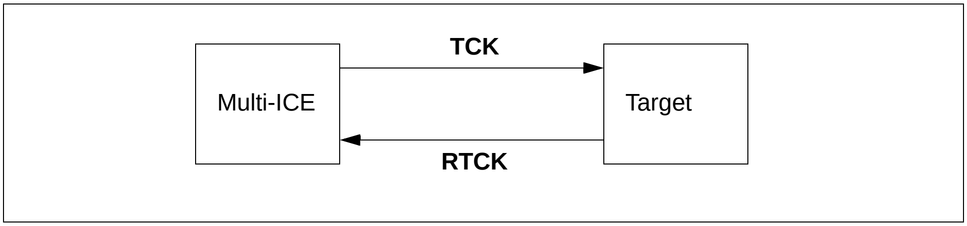


Figure 2-5: TCK and RTCK signals

When Adaptive Clocking is enabled, Multi-ICE issues a **TCK** signal and waits for the **RTCK** (**R**eturned **TCK**) to come back. Multi-ICE does not progress to the next **TCK** until **RTCK** is received.

As a simple example, ***Figure 2-6: RTCK with simple loopback*** shows how Multi-ICE uses **RTCK** to remain synchronous with the target over a short cable. The cable delay is minimal.

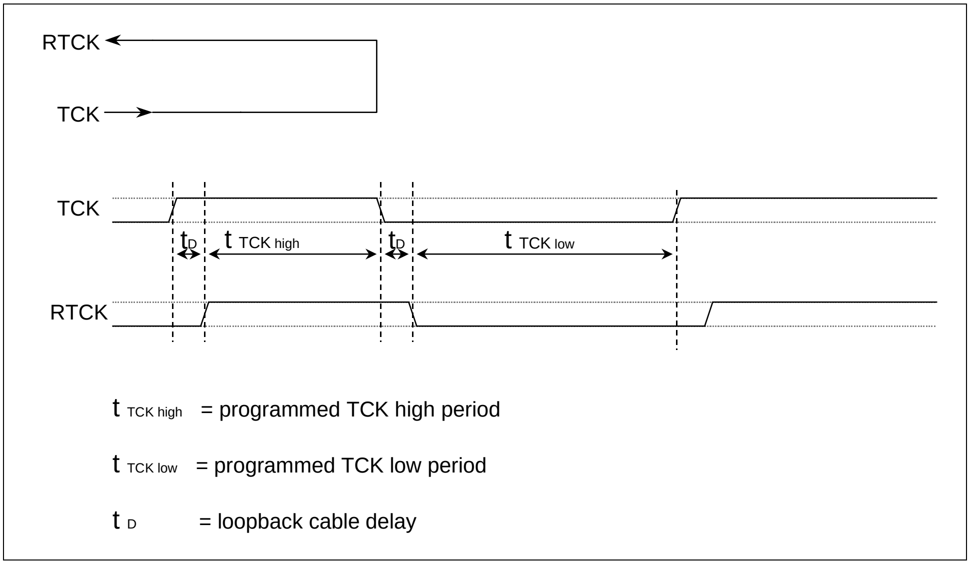

RTCK
TCK
TCK
tD
t TCK high
tD
t TCK low
RTCK
t TCK high = programmed TCK high period
t TCK low = programmed TCK low period
t D = loopback cable delay

Figure 2-6: RTCK with simple loopback

Using the same approach, Adaptive Clocking solves synchronization problems caused by:

- very long cables
- variable clock-rate devices
- TrackingICE

Note *If Adaptive Clocking is set to ON (either by a configuration file, or via the JTAG settings dialog box—see **3.8 Clocking** on page 3-20) and **RTCK** is not correctly driven, MDW will report the error:*

`Hardware interface timeout. Please reconnect to server.`

The server does not perform a check to ensure that this setting is made correctly, as this would be intrusive to the target.

2.4.1 Very long cables

With a long cable between Multi-ICE and the target device, there is a delay before the target receives a signal from Multi-ICE. Without Adaptive Clocking, Multi-ICE would keep issuing **TCK** signals without any indication that the signals were being received by the target, and could lose synchronization with the target.

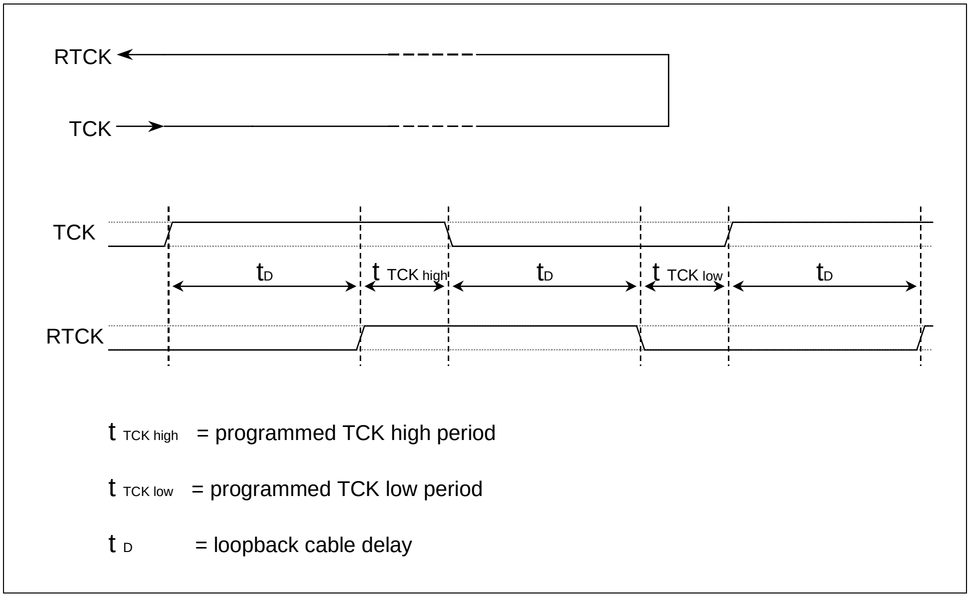

RTCK
TCK
TCK
t_D
$t_{TCK\ high}$
t_D
$t_{TCK\ low}$
t_D
RTCK
$t_{TCK\ high}$ = programmed TCK high period
$t_{TCK\ low}$ = programmed TCK low period
t_D = loopback cable delay

Figure 2-7: RTCK over a very long cable

When Adaptive Clocking is enabled, Multi-ICE waits for **RTCK** before progressing to the next **TCK**. This ensures that the target has received and acknowledged the previous **TCK** signal.

2.4.2 Variable clock-rate devices

The adaptive clocking feature allows Multi-ICE to operate with variable clock rate devices. Using a flip flop, the Master Clock signal for the target device (**MCLK**) allows the **TCK** from Multi-ICE only when the target is ready. This also permits the **RTCK** signal to return to Multi-ICE and trigger the next **TCK**.

The speed of the target device is therefore taken into account and there is no loss of synchronization. At high speeds, there is hardly any delay, as **MCLK** is running fast. At low speeds, the delay is matched to the sleep mode of the target device.

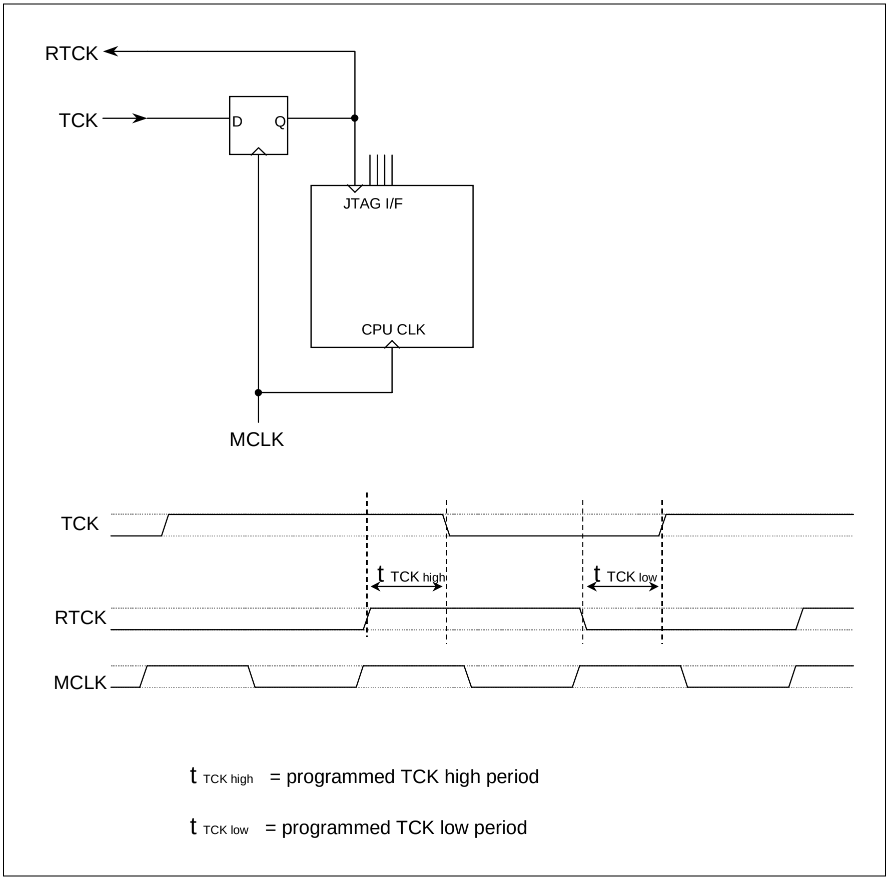

RTCK
TCK
D
Q
JTAG I/F
CPU CLK
MCLK
TCK
t TCK high
t TCK low
RTCK
MCLK
t TCK high = programmed TCK high period
t TCK low = programmed TCK low period

Figure 2-8: RTCK and a variable clock device

2.4.3 TrackingICE

Adaptive Clocking deals with the problem of synchronous and asynchronous clocks by using a flip flop to turn asynchronous **TCK** signals into synchronous signals. The Master Clock signal (**MCLK**) controls the **TCK** signals so that they become synchronous; this then allows the **RTCK** to return to Multi-ICE with no loss of synchronization.

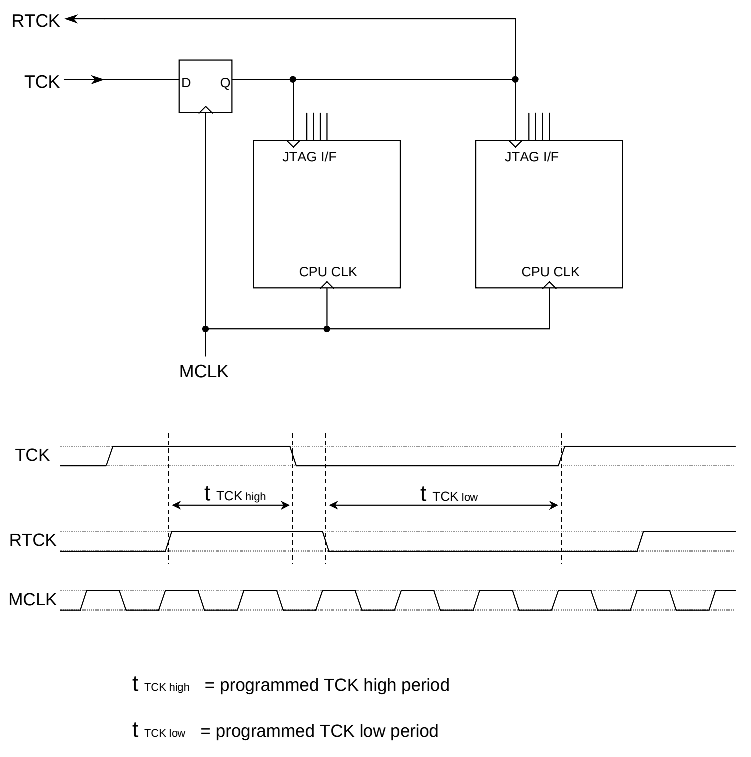

RTCK
TCK
D
Q
JTAG I/F
CPU CLK
JTAG I/F
CPU CLK
MCLK
TCK
t TCK high
t TCK low
RTCK
MCLK
t TCK high = programmed TCK high period
t TCK low = programmed TCK low period

Figure 2-9: RTCK and TrackingICE

2.4.4 Other features of Adaptive Clocking

- To use Adaptive Clocking, you need to connect **TCK** and **RTCK**.
- When you use the automatic configuration function to scan the ASIC, this issues a **TCK** signal and waits for a returned **RTCK** from the target. If **RTCK** comes back, the automatic configuration function enables Adaptive Clocking. If there is no **RTCK** within 1ms, automatic configuration turns Adaptive Clocking off. Automatic configuration is described in ***3.3.2 Automatic configuration*** on page 3-9.
- When you are downloading code to a variable speed device, you can ensure that the download happens as quickly as possible by selecting **Set on Download** from the User Output Bits dialog. The user output bit can then be used to speed up the target **MCLK** during download by adding further external logic. This is described in ***3.6 User Output Bits*** on page 3-17.

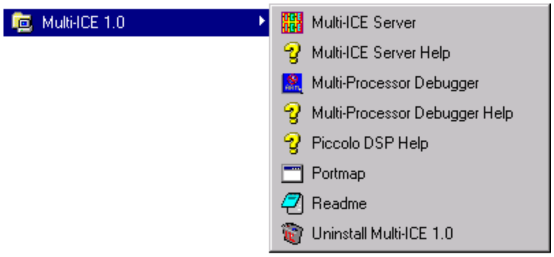
3

Using Multi-ICE

This chapter describes how you use Multi-ICE.

3.1 Starting Multi-ICE

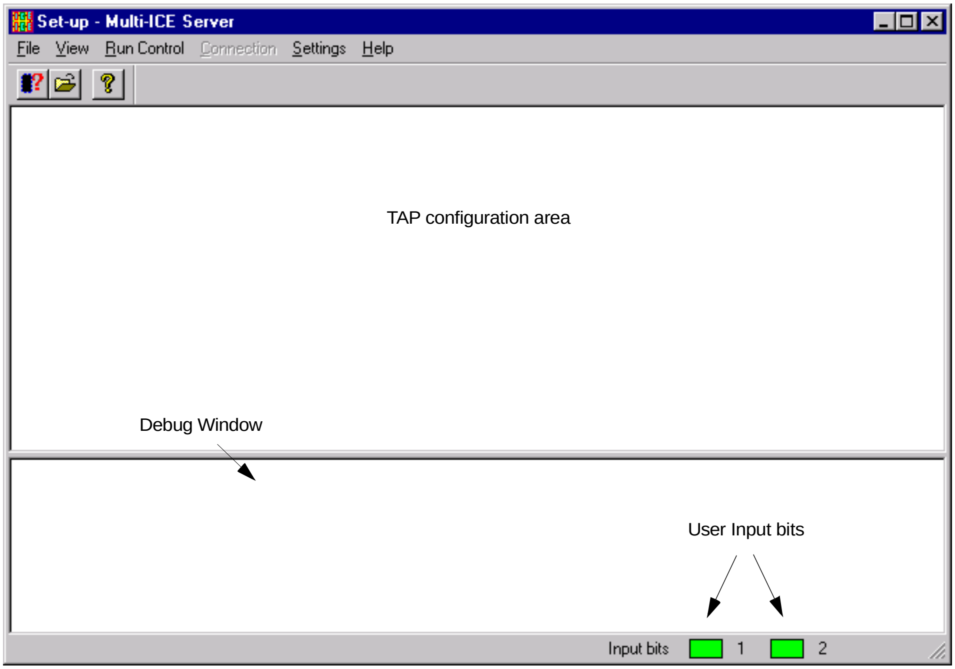
After you have installed Multi-ICE on your machine, the following menu options are available, but the options shown depend on the installation options you have selected:



If only the **Source files for non-ARM debuggers** option was selected during installation, the Multi-ICE folder will not be added to the programs menu.

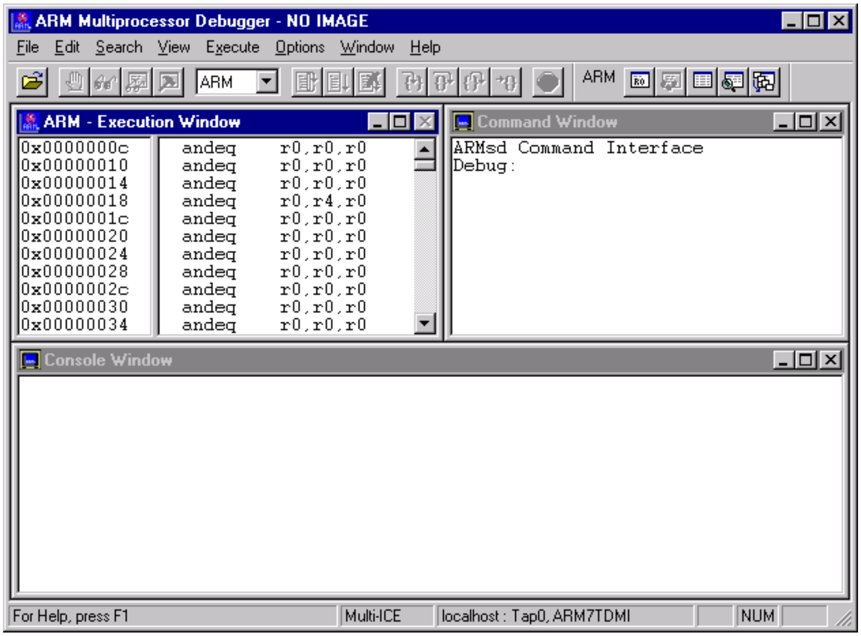
To start Multi-ICE:

1 Click on the **Multi-ICE Server** option, which displays the Multi-ICE Server window. This window shows the TAP configuration area, User Input Bits and Debug window:



The Multi-ICE menu options are explained in section ***3.2 Overview of Multi-ICE Server Menus*** on page 3-4.

2. Configure the Server, as described in section ***2.2 Multi-ICE Configuration Data*** on page 2-5.
3. If required, start the *Multi-processor Debugger for Windows (MDW)*. This displays the MDW window:



```
0x0000000c    andeq    r0,r0,r0
0x00000010    andeq    r0,r0,r0
0x00000014    andeq    r0,r0,r0
0x00000018    andeq    r0,r4,r0
0x0000001c    andeq    r0,r0,r0
0x00000020    andeq    r0,r0,r0
0x00000024    andeq    r0,r0,r0
0x00000028    andeq    r0,r0,r0
0x0000002c    andeq    r0,r0,r0
0x00000030    andeq    r0,r0,r0
0x00000034    andeq    r0,r0,r0
```

```
ARMsd Command Interface
Debug:
```

For information on how to connect to the MDW, refer to section ***3.4 Connecting an ARM Debugger to the Multi-ICE Server*** on page 3-13.

Note *This option is only available if you installed the Debugger from the installation CD.*

MDW is described in ***Chapter 4, Multi-processor Debugger for Windows (MDW)***.

3.1.1 Problems starting Multi-ICE server

Networking errors may occur if the network configuration has not been fully set up, or if a DNS lookup fails because there is no network currently connected. In the Control Panel, select **Network** to display the network settings. In Windows 95, access the ‘Configuration’ tab and then select the TCP/IP option to display the TCP/IP configuration. You may find that if your PC does not have a network connection, you will have to set up a Hosts file (in `C:\Windows` for Windows 95) which aliases the PC/DNS domain name to the `localhost` IP address.

3.2 Overview of Multi-ICE Server Menus

This section gives an overview of the menu options on the Multi-ICE Server. ***Figure 3-1: Multi-ICE Server menu options*** shows the sub menus and their options:

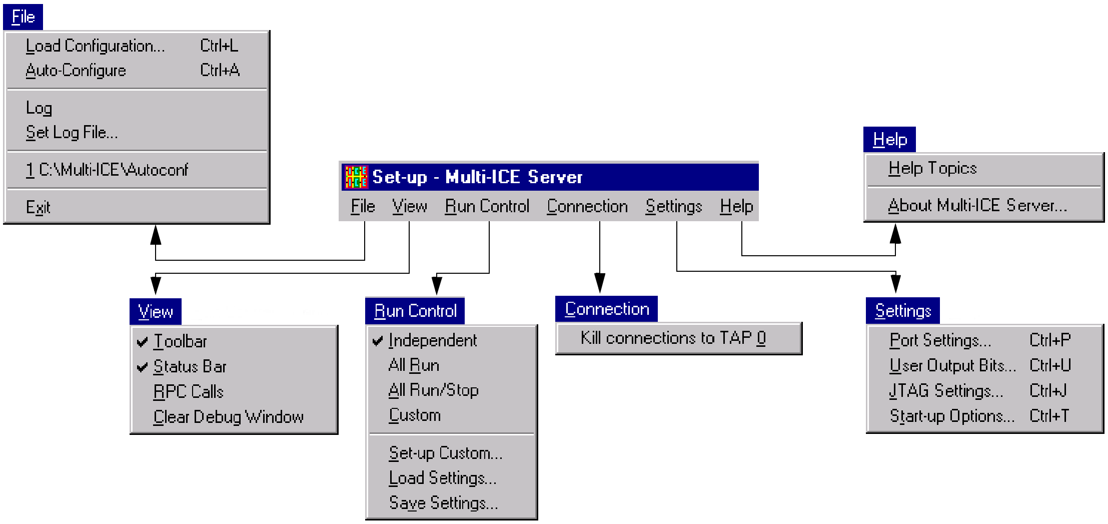
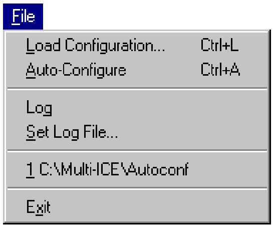


Figure 3-1: Multi-ICE Server menu options

3.2.1 File menu

The **File** menu displays the following options:

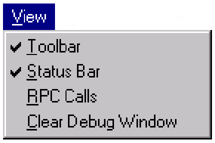


Load Configuration	displays a dialog box that you use to enter the name and path of a configuration file. This option is used for the manual configuration of Multi-ICE. This is described in ***2.2.2 Manual configuration*** on page 2-5.

Auto Configure	interrogates the ASIC and creates a configuration file naming all devices found. This is described in ***2.2.1 Automatic configuration*** on page 2-5. Unrecognized devices are marked as UNKNOWN, but you can add these to a lookup table, as described in ***2.2.5 Configuring non-ARM devices*** on page 2-8. Note that autodetection of 710T/720T/740T sometimes gives UNKNOWN, as the processor cannot be stopped before reading CP15 r0. This can be prevented by resetting the processor.
Log	turns RPC logging on or off. Data is written to the log file specified in **Set Logfile**.
Set Logfile	displays a dialog box which you use to enter the name and path of a log file.
Recent File List	displays a list of the four most recent configuration files you have used.
Exit	quits from the Multi-ICE Server.

3.2.2 View menu

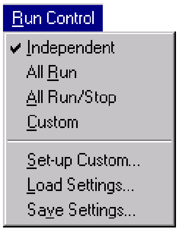
The **View** menu controls the display of the Multi-ICE Server window:



Toolbar	turns the tool bar on or off. The tool bar gives you quick access to the following functions: creates an automatically generated configuration file. prompts you for the name of a configuration file. displays information about the current version of Multi-ICE.
Status Bar	turns the status bar on or off. The Status bar displays information on the current state of Multi-ICE.
RPC Calls	enables or disables the display of all *Remote Procedure Calls (RPC)* to the debug window.
Clear Debug Window	clears all text in the debug window.

3.2.3 Run Control menu

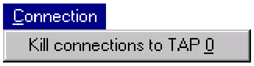
The **Run Control** menu controls the stopping and starting of cores. The options from this menu are described in detail in section ***3.9 Run Control*** on page 3-22.



Independent	makes all configured devices behave independently. There is no interaction between devices.
All Run	starts all devices together.
All Run/Stop	all configured devices run if all debuggers have requested to start; all configured devices stop if only one of the devices stops.
Custom	makes configured devices interact as customized by the user.
Set-up Custom	displays a dialog that allows you to specify the way in which devices interact. See section ***3.9.1 Setting up interaction between devices*** on page 3-22 for more information.
Load Settings	displays a dialog box that is used to enter the name and path of a file comprising previously-saved settings.
Save Settings	displays a dialog box by which the Run Control settings can be saved to a specified file.

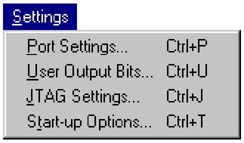
3.2.4 Connection menu

The **Connection** menu lists all TAP controllers in use and gives you the option to kill each connection individually.

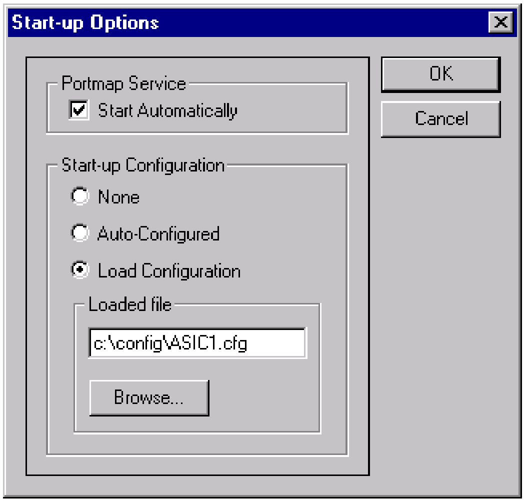


3.2.5 Settings menu

The **Settings** menu allows you to control the parallel port selection, user output bits, JTAG settings and start-up options.



Port Settings	displays a dialog box that you use to select the required parallel port address, with the option of forcing 4-bit access. It also shows the current port mode. Port settings are described in more detail in section ***3.5 Setting Ports*** on page 3-16.
User Output Bits	displays a dialog to control the user output bits. These bits correspond to two logic-level outputs available from the user I/O connector (see ***3.6 User Output Bits*** on page 3-17 and ***Appendix B, User I/O Pin Connections***). These signals can be used to remotely control user logic at the Server location.
JTAG Settings	displays a dialog that you use to set the clock speed. The clock speed can be selected from pre-set frequencies, or selected by the user either by using the **Set Periods Manually** option, or by including the information in a configuration file. See section ***3.8 Clocking*** on page 3-20 for more information. If timing information is included in the configuration file, it is selected automatically.
Start-up options	displays a Start-up Options dialog box as shown below:



Portmap Service
☑ Start Automatically

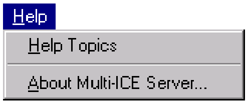
Start-up Configuration
○ None
○ Auto-Configured
● Load Configuration

Loaded file
c:\config\ASIC1.cfg

Portmap Service	if you select this option, when you next start the Multi-ICE Server the Portmap program will also start. If the Portmap Service box is not checked, the portmap service will have to be manually started from the Multi-ICE program folder *before* the Server is started, unless a portmapper service is provided by another application. Note that if the automatic start of the portmap service is not selected and a portmapping service is not being provided by another application, no warning is provided. The result is that MDW will fail to connect to the Server, and after some time the Server will report that a fatal rpc error has occurred. To overcome this: 1. Start the Multi-ICE Server again. 2. Select the Portmap Service option. 3. Close the Multi-ICE Server. 4. Restart the Multi-ICE Server.
None	if this is chosen, no auto configuration or load configuration file action is performed (see entries below).
Auto-Configured	this option creates a configuration file naming all devices found, and is described in ***2.2.1 Automatic configuration*** on page 2-5. Unrecognized devices are marked as UNKNOWN, but you can add these to a lookup table, as described in ***2.2.5 Configuring non-ARM devices*** on page 2-8.
Load Configuration	provides access to a dialog box that you use to enter the name and path of a configuration file. This option is used for the manual configuration of Multi-ICE, and is described in ***2.2.2 Manual configuration*** on page 2-5.

3.2.6 Help menu

The **Help** menu gives you access to information on Multi-ICE.



Help Topics	starts Multi-ICE help.
About Multi-ICE Server	displays information on the hardware and parallel port driver.

3.3 Multi-ICE Server Configuration Files

Multi-ICE uses a configuration file to store information on the devices on the board. Multi-ICE needs configuration data to select the correct driver. It already has configuration data on the supported ARM devices.

There are two ways to create a configuration file.

- automatically, using the menus
- manually, by creating a text file containing configuration data and loading it into Multi-ICE, as described in section ***2.2 Multi-ICE Configuration Data*** on page 2-5

3.3.1 Supported devices

The supported processors at the first release of Multi-ICE are:

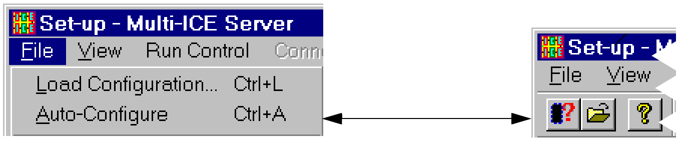
- ARM7TDMI
- ARM7TDMI-S
- ARM70DI
- ARM710T
- ARM720T
- ARM740T
- ARM9TDMI
- ARM940T
- Piccolo DSP coprocessor

Please contact ARM for the latest information regarding support of other processors.

3.3.2 Automatic configuration

If all cores are ARM cores, Multi-ICE automatically creates the configuration file by scanning the ASIC and creating the file `autoconfig.cfg`. This contains information on each TAP controller. Multi-ICE already has data on ARM devices, and if you use non-ARM devices on your board, you can declare these to Multi-ICE so that you can still use the automatic configuration facility. This is described in ***2.2 Multi-ICE Configuration Data*** on page 2-5.

You create a configuration file automatically by selecting the following menu item or shortcut:



If all devices on the board are recognized, Multi-ICE displays a pictorial representation of the devices found. ***Figure 3-2: Pictorial display of configured devices*** on page 3-10 shows a configuration with four devices.

These devices are:

- two ARM7TDMIs
- one ARM9TDMI
- one ARM720T, with Piccolo

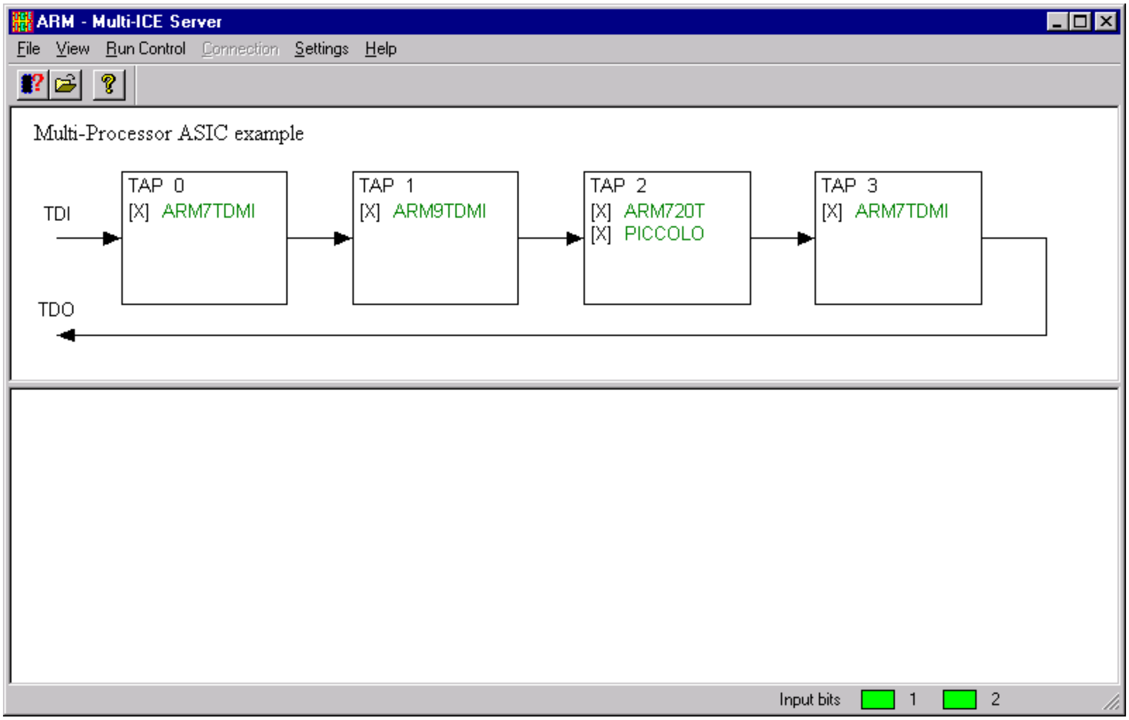


Figure 3-2: Pictorial display of configured devices

3.3.3 Information on connected cores

You can access information on the connected cores by double-clicking on the TAP controller name displayed on the schematic. The schematic diagram also shows the state of each of the cores:

[S]	denotes that the core is stopped
[R]	denotes that the core is running
[D]	denotes that the core is downloading
[X]	denotes that the core state is unknown (no debugger connected)

3.3.4 Creating a configuration file

Multi-ICE needs configuration data to calculate the length of the scan chain, by adding the length of each IR register it finds on the ASIC. This information is held in a file called `IRlength,` which you can edit to store information on other devices that you use—for example, DSP. This file is stored in the Multi-ICE directory chosen at installation. This file is further described in section ***2.2 Multi-ICE Configuration Data*** on page 2-5.

A configuration file is a text file with the file suffix `.cfg`. You can store the file anywhere on your machine; you are prompted for its name during the configuration. To create a configuration file, enter the following information into a text file:

- A title for the configuration, which is displayed with the pictorial representation of the configuration. This title is optional.
- And for each device:
    - the unique number of the TAP controller for each device
    - the device name—in the case of a non-ARM core, this must correspond to the device name added to the file `IRlength.arm`
    - TAPINFO setting (optional)
    - any special timing or clocking information (optional)

3.3.5 Example configuration file

The following shows an example of a configuration file with all types of data:

```
[TITLE]
Test TAP Configuration

[TAP 0]
ARM7TDMI

[TAP 1]
ARM9TDMI

[TAP 2]
ARM720T
Piccolo

[TAPINFO]
YES

[Timing]
High=100
Low=50
Adaptive=ON         ;Use RTCK
```

A `TAPINFO` option is included in a `.cfg` file to provide flexibility for ASIC developers during testing. When set to YES, the `TAPINFO` denotes that after loading a `.cfg` file, the Server interrogates the ASIC to gather details of each core. This can be viewed by double-clicking on a TAP controller within the Server Configuration window.

When the Autoconfigure facility is used, `TAPINFO` is always provided; the default for load configuration is `TAPINFO=NO`.

For more information on timing parameters, see ***3.8 Clocking*** on page 3-20.

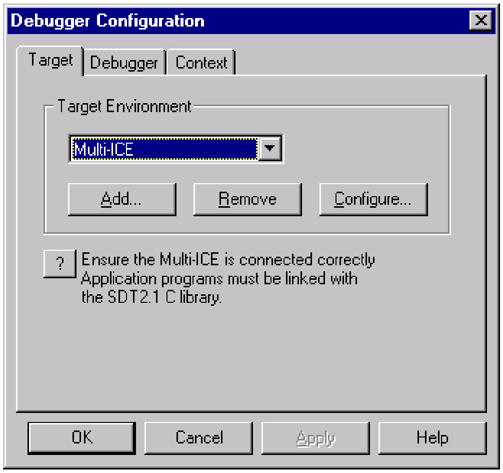
3.3.6 Loading a configuration file

To load a configuration file, select the **Load Configuration File** option from the **File** menu, and enter the name and location of the file to load in the dialog displayed. If all devices in the file are recognized, Multi-ICE displays a pictorial representation of the devices found.

3.4 Connecting an ARM Debugger to the Multi-ICE Server

After loading a configuration file or using the **Auto-Configure** option, connect the debugger to the Multi-ICE Server. This is detailed below.

1. Select **Configure Debugger** from the **Options** menu in the MDW window. This displays the **Debugger Configuration** window:



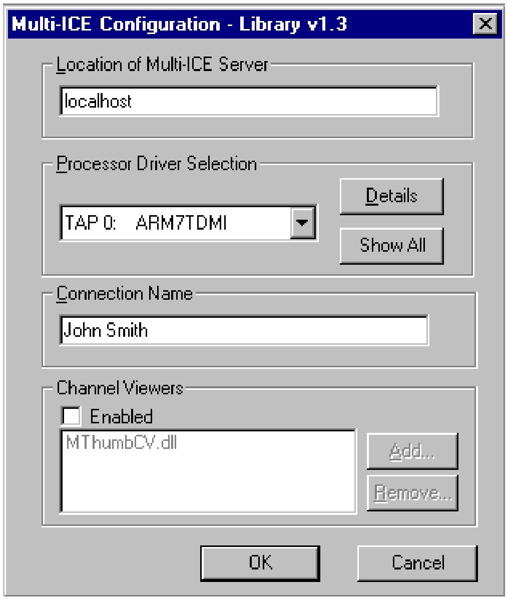
Ensure the Multi-ICE is connected correctly
Application programs must be linked with
the SDT2.1 C library.

Select the **Target** tab to display this menu. The **Debugger** and **Context** tabs are described in ***Chapter 4, Multi-processor Debugger for Windows (MDW)***.

2. Set the **Target Environment** to Multi-ICE and select **Configure**. This displays the Multi-ICE Configuration dialog box shown on the following page.

3. Add the following information into the text boxes:

Location of Multi-ICE Server	This informs the debugger where the Server is running. Enter `localhost` if the Server is running on the same PC as MDW. Enter the target PC's network ID if the Server is remote from your PC. Enter the name exactly as it appears in the Network Neighborhood. For example: `PC146` Alternatively, you can specify an IP address.



Multi-ICE Configuration - Library v1.3

Location of Multi-ICE Server: localhost

Processor Driver Selection: TAP 0: ARM7TDMI

Connection Name: John Smith

Channel Viewers

☐ Enabled

MThumbCV.dll

Processor Driver Selection	This list box is used to choose the desired processor device (or "core") for connection, and you should select a device from the list displayed. The list of devices corresponds to those shown in the TAP configuration area on the Multi-ICE Server window. By means of the **Show All**/**Show Free** toggling button, the listbox can be configured to show either: • all devices on the Server • unconnected devices If all devices are displayed, those available are shown in black, while connected devices are shown in red. If a Piccolo coprocessor is selected, the corresponding ARM device is also claimed automatically.
Connection Name	(optional) Enter a name for this connection. This helps you identify which connection you are working with. The name is displayed in the Server's debug window on connection.

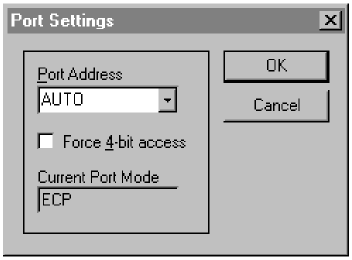
Channel Viewers	Enable or disable the selected Channel Viewer DLL. See ***4.8.5 Channel viewers*** on page 4-37 and *Application Note 38* (ARM DAI 0038) for more information on Channel Viewers.

Add	adds a Channel View DLL
Remove	removes the selected DLL

4. Click on **OK**. You will return to the Debugger Configuration dialog box.
5. Click **OK** again to connect to the target processor. If the connection is successful, the device name turns red on the schematic and connection information is displayed in the Server console window. Otherwise, an error message is displayed.

3.5 Setting Ports

This section describes how you use the **Port Settings** dialog. This is accessed from the **Settings** menu.



Port Settings

Port Address: AUTO

☐ Force 4-bit access

Current Port Mode: ECP

Note *The Win 95 driver does not use ECP mode, as the Windows ECP-aware driver interferes with the operation of the Multi-ICE driver.*

3.5.1 Port address

The parallel port address to be used can be one of:

AUTO	automatically selects the parallel port to use
LPT1	selects LPT1 as the parallel port to use
LPT2	selects LPT2 as the parallel port to use

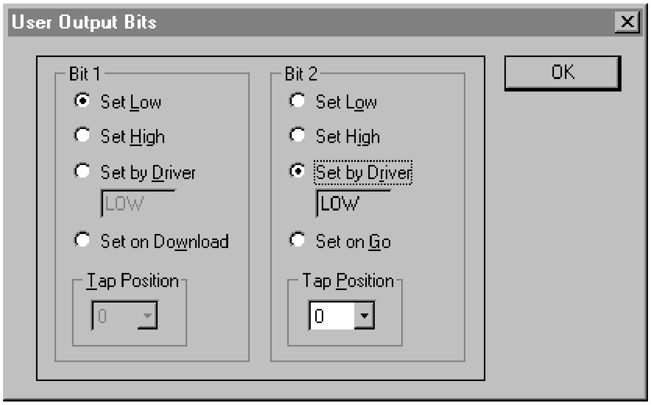
3.5.2 Force 4-bit access

Forces the parallel port to use 4-bit data transfer.

3.6 User Output Bits

The User Output Bits correspond to two TTL logic-level outputs available from the user I/O connector (see ***Appendix B, User I/O Pin Connections***). These signals can be used to remotely control user logic at the Server location.

You access the **User Output Bits** dialog box from the **Settings** menu.



User Output Bits

Bit 1
- ● Set Low
- ○ Set High
- ○ Set by Driver
  LOW
- ○ Set on Download

Tap Position: 0

Bit 2
- ○ Set Low
- ○ Set High
- ● Set by Driver
  LOW
- ○ Set on Go

Tap Position: 0

3.6.1 Bit settings

You see changes to the User Output Bits as soon as you click on them. You do not have to click on **OK** for the changes to take effect.

There are three common settings for Bits 1 and 2:

Set Low	turns the Bit permanently off (LOW)
Set High	turns the Bit permanently on (HIGH)
Set by Driver	enables the output bits to be controlled by the TAPOp procedure calls `TAPOp_writeMICEUser1` and `TAPOp_writeMICEUser2`. These are described in ***Chapter 5, TAPOp Procedure Calls***.

In addition:

Set on Download	Bit 1	can optionally be set HIGH while a debugger connection is downloading to the particular TAP controller
Set on Go	Bit 2	can optionally be set HIGH while a debugger connection is executing an image file on a specified TAP controller.

3.6.2 Tap position

You can select which TAP controller to use with the User Output bits by selecting its number from the **Tap Position** menu option. All available TAP controllers are listed

3.7 User Input Bits

The User Input Bits correspond to two TTL-level inputs at the user I/O connector. These input signals are available for use by user applications. See ***Appendix B, User I/O Pin Connections*** for further information.

The Multi-ICE JTAG port automatically adapts its input and output thresholds to the voltage levels in the target system (based on the VTref pin). The inputs on the Multi-ICE user I/O connector operate at standard TTL levels. If you need to drive one of the user-defined inputs with a signal operating at the target system's logic levels, the circuit shown in ***Figure 3-3: Converting user-input signals to TTL levels*** can be used to convert this to TTL levels.

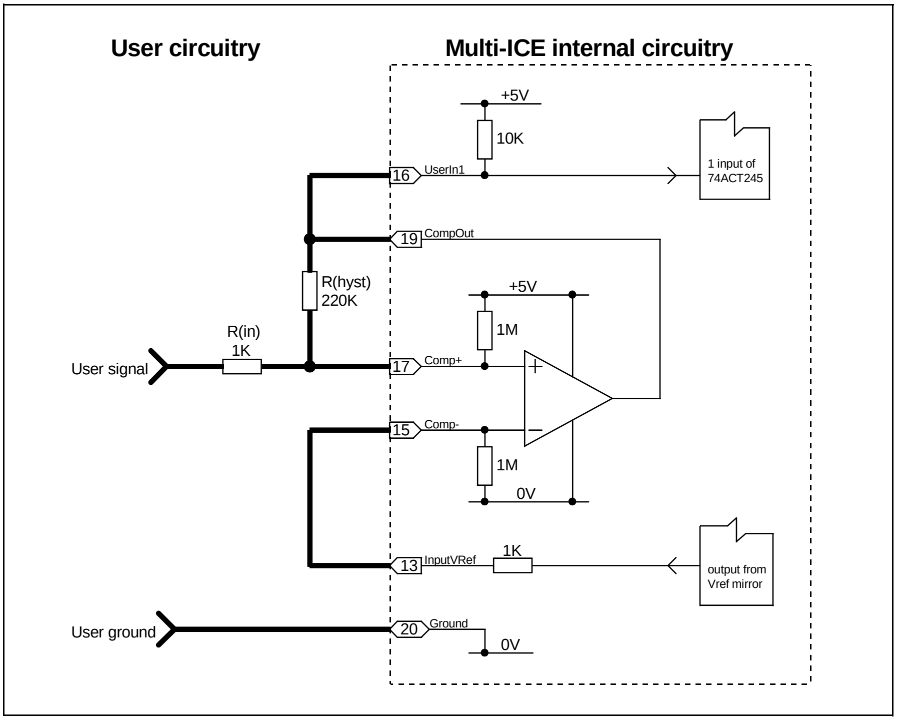


Figure 3-3: Converting user-input signals to TTL levels

Pins 19, 17 and 15 are connected to an LM339 type comparator within the Multi-ICE unit. The open collector comparator output (pin 19) drives the user input (pin 16), which includes a suitable pull-up resistor. The inverting input to the comparator (pin 15) is driven by an output from the voltage reference mirror circuit (pin 13). The non-inverting input to the comparator (pin 17) is driven by the signal to be monitored via a small series resistor (Rin).

The output of the comparator is also fed back to this input through a large resistor (Rhyst) to provide a small amount of hysteresis (around 20mV with the values shown). A ground reference for the input signal should be connected to pin 20 to provide a more direct return path than via the JTAG connector.

3.7.1 Viewing the User Input Bits

These bits are shown at the bottom-right corner of the Multi-ICE Server window. Each bit is:

light green	when high
dark green	when low

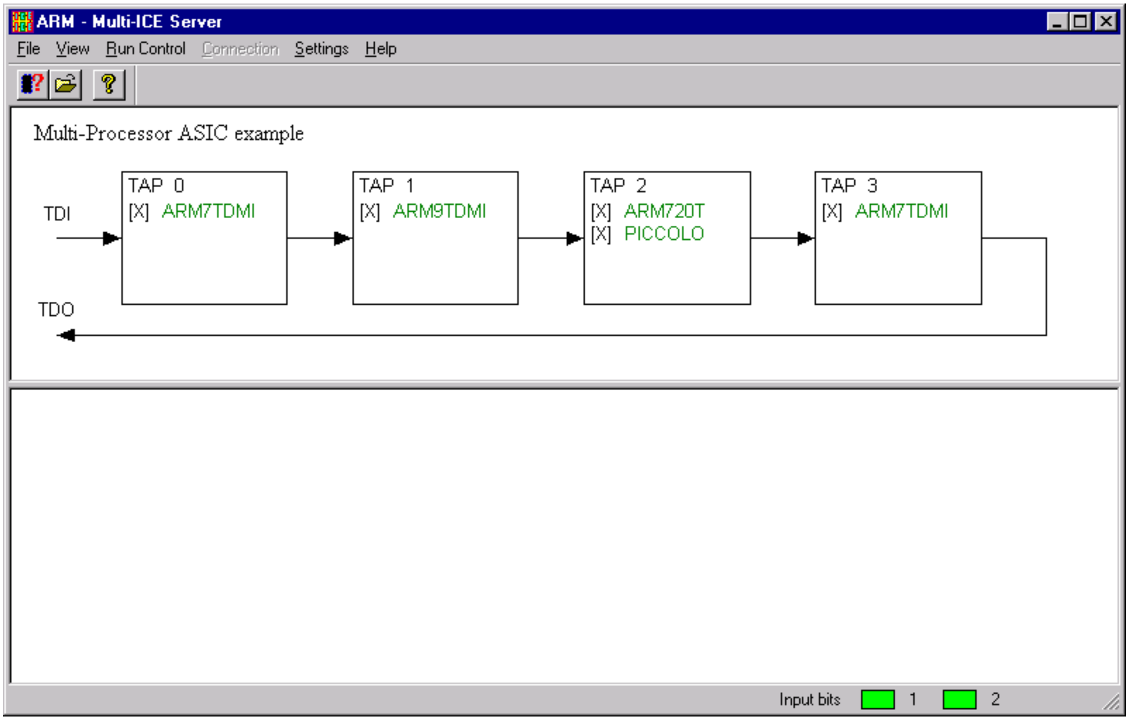
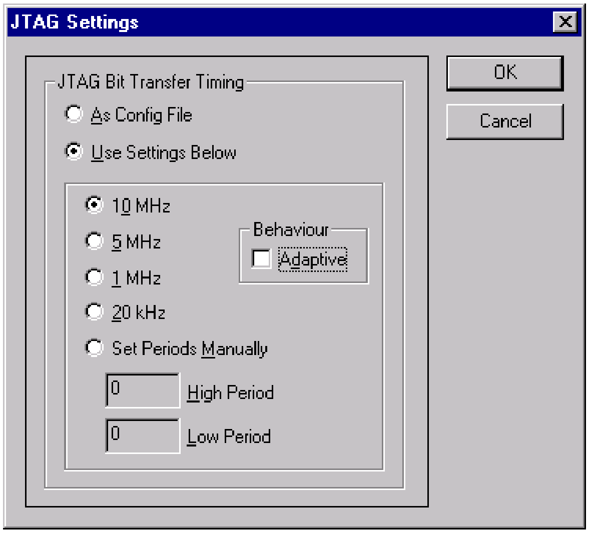


Figure 3-4: Status of the User Input Bits

3.8 Clocking

You can select the JTAG clock speed by:

- using one of the pre-set options
- selecting the clock's high and low frequencies as defined in the configuration file. If timing information is present in the configuration file, it is selected automatically.



JTAG Settings

JTAG Bit Transfer Timing

○ As Config File

● Use Settings Below

● 10 MHz

○ 5 MHz

○ 1 MHz

○ 20 kHz

○ Set Periods Manually

Behaviour

☐ Adaptive

0 High Period

0 Low Period

Note *At very low JTAG clock rates, the parallel port driver used by the Server uses a large proportion of processing time. This causes any applications that are running to execute at a reduced speed.*

3.8.1 Using your own values

If you select the **Set Periods Manually** option, or include the clocking information in a configuration file, you calculate the high and low periods using the following formula:

```
t = 50ns * ( scale * ( multiplier + 1) )
```

You can enter values between 0–255 for the HIGH and LOW periods. The 8-bit values you enter are split into 3 and 5 bits to form the scale (`S`) and the multiplier (`M`) values:

7							0
S	S	S	M	M	M	M	M

Multiplier value

The multiplier is formed from the lower 5 bits—in other words, values 0–31.

Scale value

Use ***Table 3-1: Scale values for clocking speeds*** to obtain the scale value. SSS are the three most significant bits.

SSS	Scale
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Table 3-1: Scale values for clocking speeds

Sample frequencies

```
100kHz (approx)    HIGH = LOW  = 162  [SSS = 5 (S = 32), M = 2]
500kHz             HIGH = LOW  = 19   [SSS = 0 (S = 1),  M = 19]
2MHz               HIGH = LOW  = 4    [SSS = 0 (S = 1),  M = 4]
```

For a list of tabulated frequencies and values, refer to ***A.3 TCK Frequencies*** and ***A.4 TCK Values***.

3.8.2 Adaptive clocking

Select the **Adaptive Clocking** function to synchronize the clock to the processor clock outside the core. This ensures there are no synchronization problems over the JTAG interface.

For a full description of the concept of Adaptive Clocking, see section ***2.4 Adaptive Clocking*** on page 2-10.

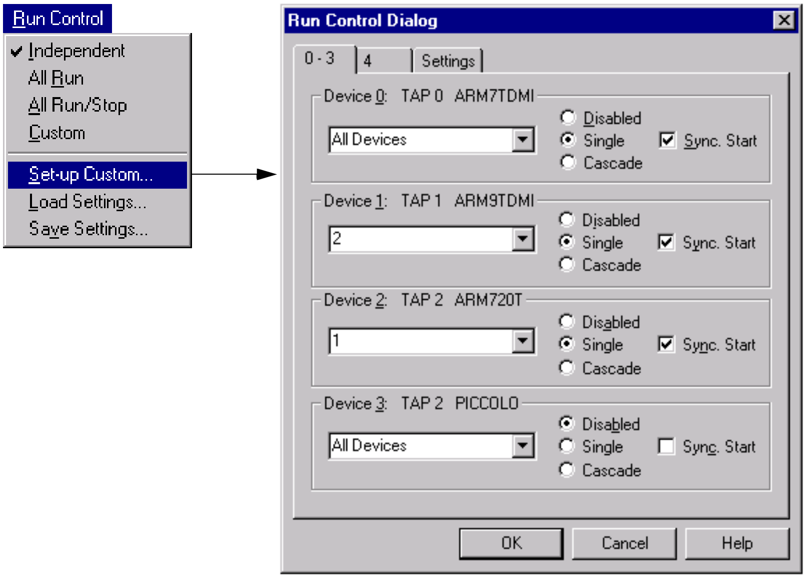
3.9 Run Control

This section describes how to set up Multi-ICE to control the way devices interact. The **Set-up Custom** option on the **Run Control** menu displays a dialog with several pages, as described below.

Note *If Run Control is required while semihosting, the debugger internal variable* `$semihosting_enabled` *should be set to 2 (Debug Comms Channel-based Semihosting) for correct operation.*

3.9.1 Setting up interaction between devices

The pages of device numbers list all the available devices in blocks of four. These pages allow you to set up the interaction between single devices or a range of devices. If you have fewer than four devices on a page, “NOT VALID” is displayed for the unnumbered boxes.



Run Control Dialog

Device 0: TAP 0 ARM7TDMI
All Devices
○ Disabled ● Single ○ Cascade ☑ Sync. Start

Device 1: TAP 1 ARM9TDMI
2
○ Disabled ● Single ○ Cascade ☑ Sync. Start

Device 2: TAP 2 ARM720T
1
○ Disabled ● Single ○ Cascade ☑ Sync. Start

Device 3: TAP 2 PICCOLO
All Devices
● Disabled ○ Single ○ Cascade ☐ Sync. Start

For each device listed on the page, there are a range of control options:

Range field	This is the combo box directly beneath each of the device descriptions, and allows you to select any devices that are to be stopped by the current device. The default option here is **All Devices**. If there are more than two devices available, you can select: • all devices • individual device numbers • a range of devices For example, if you had 10 devices listed, you could stop devices 2, 5, 7, 8, and 9 using the following notation: `2, 5, 7-9`
Disabled	'Stop' events from this device are disabled and have no control over other devices.
Single	Stops any devices in the Range field. If devices in the Range field have Stop events set up, these events will *not* be actioned.
Cascade	Stops any devices in the Range field. If devices in the Range field have Stop events set up, these events will *also* be actioned. When device 1 stops, it will stop devices 2, 4, and 6. If devices 2, 4, and 6 are set up for cascade operation, they will stop devices below them, and so on. For example, ***Figure 3-5: Cascade operation*** shows that device 1 stops three other devices, which in turn stop a further set of devices.

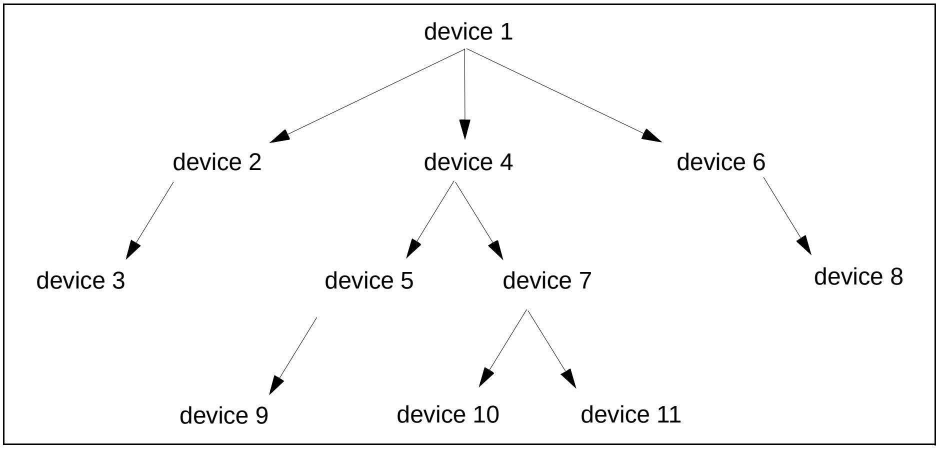


Figure 3-5: Cascade operation

Sync. start This box should be checked if you wish to start a device synchronously with other devices that have their **Sync. Start** boxes checked.

Combining options

To provide finer control, you can combine options to achieve individual results. Using the above diagram as an example:

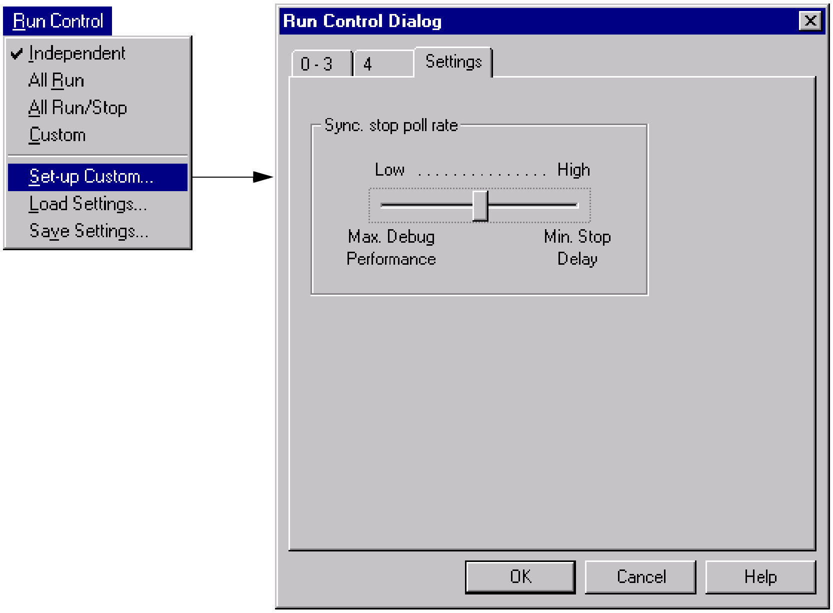
- If device 4 was disabled, devices 5, 7, 9, 10, and 11 would not be affected by the control options from device 1.
- If devices 4 and 5 were set to **Cascade**, but device 7 was disabled, the control options from device 1 would reach device 9, but not devices 10 or 11.
- If device 6 was set to **Single** instead of **Cascade**, device 8 would not be affected by the control options from device 1.

3.9.2 Setting up the poll rate

The **Settings** tab displays a page that allows you to control Multi-ICE's poll rate so you can find a suitable balance between optimal debug performance and the number of times Multi-ICE polls the devices to find out their status (stop delay).

Low	Maximum debug time, with minimal polling to find the status of devices.
High	Maximum polling of devices, with virtually no debug time.

The default setting is midway on the scale.



3.10 Error Messages

The following list gives the error messages that are displayed in dialog boxes, and provides an explanation and recovery from the error, wherever possible.

3.10.1 Server messages

`An error occurred while attempting to set up the TAP configuration.`

The hardware could not be configured, or the application ran out of memory.

`An error occurred while opening the selected port.`

An unexpected software error has occurred. Please contact technical support.

`An error occurred while retrieving the hardware details. This application will now terminate.`

Please contact technical support.

`An error prevented retrieval of hardware details.`

Please contact technical support.

`A fatal RPC error occurred. The server will now terminate.`

The portmap program is not running, or the PC is not running a TCP/IP stack.

`Auto-Configuration failed. Check the target power is on. Your chip may require manual configuration.`

This may be because:

- the power is off
- the Multi-ICE directory does not exist, or could not be written to. Check that the file exists and that you have sufficient access to it.
- the chip may not be an ARM chip—you need to write a manual configuration file. See ***2.2.5 Configuring non-ARM devices*** on page 2-8.

`Could not find the Multi-ICE hardware. Please check that the hardware is properly connected to the parallel port and powered up.`

Check that the Multi-ICE unit has power, and is connected to the PC.

If Multi-ICE is being used with an EmbeddedICE (14-way) connector:

- if power is being supplied by the target, check that the jumper on the adaptor is correctly positioned and that the necessary resistor has been shorted
- if power is being supplied externally, check that the supply voltage is correct and switched on

`Failed to open parallel port. The port may be in use.`

This may be because:

- another device is using the parallel port (for example, a printer)
- you have faulty parallel port hardware

`Multi-ICE driver failure. Check that Multi-ICE device driver is installed and running and there is no other Multi-ICE server running.`

The driver is not present or has not started. On an NT machine, check in the Control Panel›Devices under Multi-ICE that the driver is present and has started. The Multi-ICE driver is started automatically by the Server when running under Windows 95.

`The server could not initialize correctly. Please close down one or more applications and re-start.`

Other applications are using system resources required by the Multi-ICE Server.

`The server installation is incomplete or damaged. You may wish to re-install the software.`

The Server cannot find an entry in the registry.

`The attached device is not compatible with this server.`

The hardware attached is not a Multi-ICE unit.

`The decision to read core information has been re-specified in file` *`filename`* `at line` *`number.`*

It has been specified more than once in the configuration file as to whether core information is automatically read from the connected devices.

`The selected server is not compatible with this debugger.`

The ARM Multi-ICE DLL can only be used with the ARM Multi-ICE Server.

3.10.2 Debugger messages

`[`*`driver`*`] is not an ARM core and cannot be debugged.`

The specified driver name cannot be debugged with ARM tools.

`The driver for [`*`driver`*`] could not be found.`

The `.MUL` file corresponding to the device does not exist in the expected location.

`The driver for [`*`driver`*`] could not be found. This was due to an installation problem. Please re-install the Multi-ICE software.`

The Multi-ICE DLL cannot find an entry in the registry.

`The driver for [`*`driver`*`] could not be read.`

This is because:

- the `.MUL` file corresponding to the device has been opened by another application
- the `.MUL` file does not exist
- the `.MUL` file is corrupted or out of date

`The format of the driver for [`*`driver`*`] is not valid.`

The `.MUL` file corresponding to the device has been corrupted, or the `.MUL` file for the device is out-of-date.

`The Multi-ICE server on [`*`location`*`] cannot be found. If you wish to use this server you should check that it has been started and the location given correctly.`

The Multi-ICE Server is not running. Check that:

- the name of the Server PC is entered correctly
- the network is functioning properly
- `portmap` is running on the Server PC

`The Multi-ICE server on [`*`location`*`] does not have available the previously selected [`*`driver`*`] on TAP [`*`number`*`]. The server configuration may have been altered or the driver may already be connected. Please check the Multi-ICE server if you wish to use this.`

This is because:

- the set of devices connected to the Multi-ICE Server has been altered so that the device previously being debugged is no longer present
- the device previously being debugged is still present, but attached to another debugger

`The Multi-ICE server on [`*`location`*`] has not been initialized.`

The Server was found, but not configured (either by auto-configuration or `.cfg` file).

`The Multi-ICE server on [`*`location`*`] returned an expected error of [`*`number`*`]. The server may be incompatible.`

The DLL and Server versions are incompatible.

`The Multi-ICE server on [`*`location`*`] returned too much driver information. The version of the server software may not be compatible with your current debugger.`

The DLL and Server versions are incompatible.

`The previously selected Multi-ICE Server has no available devices.`

All devices connected to the Multi-ICE Server are attached to other debuggers.

`The server requires the driver for [driver] to be at least version [`*`number`*`].0.`

The Multi-ICE Server needs the debugger to use a later driver version for the intended device. Re-install the `.MUL` files from the Multi-ICE installation CD.

4

Multi-processor Debugger for Windows (MDW)

This chapter gives an overview of the functions provided in the Multi-processor Debugger for Windows. For full details of the debugger, please refer to the *Software Development Toolkit User Guide* (ARM DUI 0040) and *Reference Guide* (ARM DUI 0041).

Multi-processor Debugger for Windows (MDW)

4.1 Introduction

The *Multi-processor Debugger for Windows (MDW)* enables you to debug your ARM targeted image using any of ARM's debugging systems.

MDW works in conjunction with Multi-ICE. You debug your application using a number of windows that give you various views on the application you are debugging.

Refer to the documentation supplied with your target board for more information on development boards.

This chapter provides:

- a basic introduction to MDW
- an overview of the terminology used in ARM debugging
- an overview of the MDW desktop and windows
- a step-by-step guide to debugging a simple application
- ways of displaying information while debugging
- an overview of more advanced debugging functions

4.1.1 Multi-ICE drivers

The following driver files are supplied for the supported ARM cores. These are stored in the Multi-ICE directory on installation:

```
ARM70DI.MUL
ARM710T.MUL
ARM720T.MUL
ARM740T.MUL
ARM7TDMI.MUL
ARM7TDMI-S.MUL
ARM940T.MUL
ARM9TDMI.MUL
Piccolo.MUL
```

4.1.2 CP15

Multi-ICE provides support for coprocessors. The Multi-ICE DLL automatically sets up CP15, and the read/write settings for CP15 registers are held in the relevant `.MUL` file. For further information, see ***Appendix C, CP15 Register Mapping***.

4.1.3 On-line help

When you have started MDW, you can use online Help to find information on the tasks you are performing. You have several options for accessing the Help system:

Contents	Select **Contents** from the **Help** menu to display a Table of Contents.
Search	Select **Index** from the **Help** menu to display an index of all the help topics.
Help	Click the **Help** button to get information on the dialog currently on display.
F1	Press the **F1** key on your keyboard to get help on the currently active window or the dialog box currently on display.

4.2 MDW Concepts

This section describes the terminology used throughout this chapter.

4.2.1 Backtrace

When your program has halted (for example, by your stopping the program, or setting a breakpoint or watchpoint), backtrace information is displayed in the Backtrace Window to give you information about the procedures that are currently active.

The following example shows the backtrace information for a program compiled with debug information and linked with the C library:

```
#DHRY_2:Proc_6 line 42
#DHRY_1:Proc_1 line 315
#DHRY_1:main line 170
PC = 0x0000eb38 (_main + 0x5e0)
PC = 0x0000ae60 (_entry + 0x34)
```

Lines 1–3:

The first line indicates the function you are currently in. The second line indicates the source code line from which this function was called, and similarly the third line indicates the call to the second function.

Lines 4–5:

Line 4 shows the position of the call into your program's main procedure, and the final line indicates the entry point made by the C library's call into your program and

Note *A simple assembler program compiled without debug information and not linked to a C library would show only the PC values.*

4.2.2 Breakpoints

A breakpoint is a point in the code where your program will be halted by MDW. Once you have set a breakpoint it will appear as a red marker in the window.

There are two types of breakpoints:

- a simple breakpoint that stops at a particular point in your code
- a complex breakpoint that:
  - stops when the program has passed the specified point a number of times

  AND/OR

  - stops at the specified point only when an expression is true

You can choose to set a breakpoint at a point in the source or in the disassembled code if it is currently being displayed, with the interleaved source option or the disassembly view. You can also set breakpoints on individual statements on a line, if that line contains more than one statement.

You can set, edit or delete breakpoints in the following windows:

- Execution
- Disassembly
- Source File
- Low-level symbols
- Command
- Backtrace
- Breakpoints

4.2.3 Disassembled code

Disassembled code is the machine code generated by the disassembly process.

You can display disassembled code in the Execution Window or in the Disassembly Window (select **Disassembly** from the **View** menu).

You can also choose the type of disassembled code to display by accessing the **Disassembly Mode** sub-menu, from the **Options** menu. ARM code, Thumb code or both can be displayed, depending on your program's image type.

4.2.4 High- and low-level symbols

A high-level symbol for a procedure refers to the address of the code generated by the first statement in the procedure, and is denoted by the function name shown in the Function Names Window.

A low-level symbol for a procedure refers to its call address, often the first instruction of the stack frame initialization. You can display a list of the low-level symbols in your program in the Low-level Symbols Window.

To indicate a low-level symbol in a regular expression, precede the symbol with `@`.

To indicate a high-level symbol precede it with `^`.

Refer to the *ARM Software Development Toolkit Reference Guide* (ARM DDI 0041) for information about predefined low-level symbols.

4.2.5 Regular expressions

Regular expressions are the notation for specifying and matching strings, similar to arithmetic expressions. A regular expression is either:

- a single extended ASCII character (other than the special characters described below)
- a regular expression modified by one of the special characters

You can include low-level symbols or high-level symbols in a regular expression.

The following special characters modify the meaning of the previous regular expression, and will not work if no such regular expression is given.

`*`	Any number of the proceeding regular expressions (including no occurrences). For example, `A*B` would match `B`, `AB` and `AAB`.
`?`	Either one copy of the proceeding regular expression, or nothing at all. For example, `AC?B` matches `AB` and `ACB` but not `ACCB`.
`+`	At least one copy of the regular expression. For example, `AC+B` matches `ACB` and `ACCB`, but not `AB`.

The following special characters are regular expressions in themselves:

`\`	Precedes any special character you want to include literally in an expression to form a single regular expression. For example, `\*` matches a single asterisk (`*`) and `\\` matches a single backslash (`\`). The regular expression `\x` is equivalent to `\x` as the character `x` is not a special character.
`()`	Allows grouping of characters. For example, `(202)*` matches `202202202` (as well as nothing at all), and `(AC?B)+` looks for sequences of `AB` or `ACB`, such as `ABACBAB`.
`.`	Exactly one character. This is different to `?` in that the period (`.`) is a regular expression in itself, so `.*` matches all, while `?*` is invalid. **Note:** The period (`.`) does *not* match the end-of-line character.
`[ ]`	A set of characters, any one of which may appear in the search match. For example, the expression `r[23]` would match strings `r2` and `r3`. The expression `[a-z]` would match all characters between `a` and `z`.

Note *Pattern matching is done following the UNIX* `regexp(5)` *format, but without the special symbols,* `^` *and* `$`*.*

4.2.6 Search paths

If you want to view the source for your program's image during the debugging session, then MDW needs to know how to find the files. A search path points to a directory or set of directories that are used to locate files whose locations are not referenced absolutely.

If you are developing your program using the ARM Project Manager, the search paths are set up automatically.

If you are using the ARM command-line tools to build your project, you may need to edit the search paths for your image manually, depending on the options you chose when you built it.

If for some reason the files have moved since the image was built, the search paths for these files must be set up in the MDW, using the Search Paths Window (see ***Search paths*** on page 4-21).

Note *This version of Multi-ICE does not support directories that contain spaces. MDW will not function correctly under some circumstances if it is installed to such directories.*

4.2.7 Watchpoints

In its simplest form, a watchpoint halts a program when a specified register or variable is changed. The watchpoint will halt the program at the next statement or machine instruction after the one that triggered the watchpoint.

There are two types of watchpoints:

- a simple watchpoint that stops when a specified variable changes
- a complex watchpoint that:
  - stops when the variable has changed a specified number of times

  AND/OR

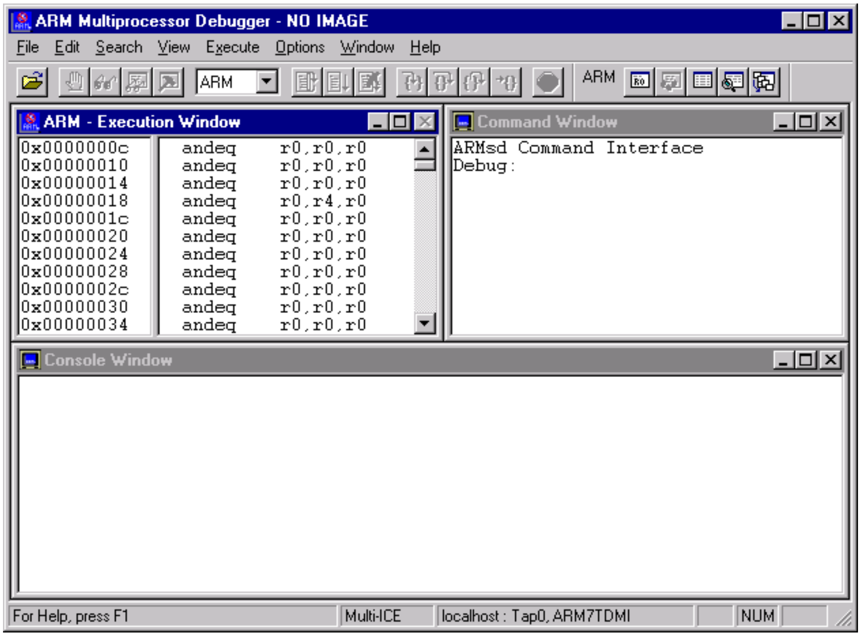
  - stops when the variable is set to a specified value

Note *If you set a watchpoint on a local variable, you will lose the watchpoint as soon as you leave the function which uses the local variable.*

4.3 The MDW Desktop

This section describes MDW and the windows available during your debugging session.

The MDW Desktop consists of a number of windows that are used to display a variety of information as you work through the process of debugging your executable image. Three windows, the Execution Window, the Command Window and the Console Window are opened automatically when you start the debugger and remain. The Execution Window is always open. You can open other windows by selecting the appropriate item from the **View** menu. ***Figure 4-1: MDW at startup*** shows the initial display of the MDW windows:



```
0x0000000c    andeq    r0,r0,r0
0x00000010    andeq    r0,r0,r0
0x00000014    andeq    r0,r0,r0
0x00000018    andeq    r0,r4,r0
0x0000001c    andeq    r0,r0,r0
0x00000020    andeq    r0,r0,r0
0x00000024    andeq    r0,r0,r0
0x00000028    andeq    r0,r0,r0
0x0000002c    andeq    r0,r0,r0
0x00000030    andeq    r0,r0,r0
0x00000034    andeq    r0,r0,r0
```

```
ARMsd Command Interface
Debug:
```

Figure 4-1: MDW at startup

Execution window

This window displays the source code of the program currently executing. You can:

- execute the entire program or step through the program line by line
- examine the contents of variables or registers
- change the display mode to show disassembled machine code interleaved with high-level C source code
- display another area of the code by address
- set, edit or remove breakpoints

Command window

Using the Command Window, you can use `armsd` instructions when you are debugging your image. Type `help` at the Debug prompt for information on the available commands or refer to the *ARM Software Development Toolkit Reference Guide* (ARM DDI 0041).

Console window

The Console Window allows interaction between you and the executing program. Anything printed by the program (for example, a prompt for user input) is displayed in this window and any input required by the program must be entered here. Information remains in the window until you select **Clear** from the **Console Window** menu. You can also save the contents of the Console Window to disk, by selecting **Save** from the **Console Window** menu.

Note *When input is required by your executable image, most MDW functions are disabled until the required information has been entered.*

4.3.1 Other available windows

Additional windows can be displayed from the **View** menu.

Window Type	Description
Backtrace	Displays current backtrace information about your program.
Breakpoints	Displays a list of all breakpoints set in your image.
Debugger Internals	Displays some of MDW's internal variables. Use this window to change the value of editable variables.
Disassembly	Displays disassembled code interpreted from a specified area of memory. The memory addresses are listed in the left-hand pane and the disassembly code is displayed in the right-hand pane. ARM code, Thumb code or both can be displayed.
Expression	Displays the values of selected variables and/or registers.
Function Names	Displays a list of the functions that are part of your program.
Locals/Globals	Locals displays a list of variables currently in scope, and the Globals Window displays a list of global variables. The variable name is displayed in the left-hand pane, the value is displayed in the right-hand pane.
Low Level Symbols	Displays a list of all the low-level symbols in your program.

Table 4-1: List of additional MDW windows

Window Type	Description
Memory	Displays the contents of memory at a specified address. Addresses are listed in the left-hand pane, and the memory content is displayed in the right-hand pane.
Registers	Displays the registers corresponding to the mode named at the top of the window, with the contents displayed in the right-hand pane. You can double-click on an item to modify the register's value.
RDI Log	Displays remote debug information—for example, the low-level communication messages between the MDW and the target processor.
Search Paths	Displays the search paths of the image currently being debugged. You can remove a search path from this window using the delete key.
Source Files List	Displays a list of all source files that are part of the loaded image. From this window you can select a source file that will be displayed in its own Source File Window.
Source File	Displays the contents of the source file named at the top of the window. The line number is displayed in the left-hand pane, the code is in the right-hand pane.
Watchpoints	Displays a list of all watchpoints.

Table 4-1: List of additional MDW windows (Continued)

4.3.2 Status bar

At the bottom of the Desktop is the status bar, which provides current status information or describes the currently selected user interface component. The Server location and the TAP position are also displayed in the MDW status bar.

4.4 Getting Started

This section details the basics of debugging an executable image, including the following debugging tasks:

- Loading an image
- Executing an image
- Stepping through an image
- Setting breakpoints and watchpoints
- Removing a breakpoint or watchpoint
- Examining and setting variables and registers
- Examining memory
- Reloading the image
- Exiting the debugger

For information on more advanced features, refer to the following:

- ***4.5 Debugger Configuration*** on page 4-18
- ***4.6 Displaying Image Information*** on page 4-21
- ***4.7 Setting and Editing Complex Breakpoints and Watchpoints*** on page 4-31
- ***4.8 Other Debugging Functions*** on page 4-34

4.4.1 Loading an image

When you load a program image, the program is displayed in the Execution Window as disassembly code (the Command and Console Windows are also displayed), and a breakpoint is automatically set at the entry point of the image, usually the first line of source after the `main()` function. The current execution marker (a green bar indicating the current line) is located at the entry point of the program.

Note *Once you have executed your program, you must reload it to execute again. See **4.4.8 Reloading the image** on page 4-17.*

To load an image:

1. Select **Load Image** from the **File** menu or click the **Load Image** button. The Open File dialog is displayed.
2. Select the **File Name** of the executable image you wish to debug, using the **Browse** option if necessary.
3. Enter any command-line **Arguments** expected by your image.
4. Click **OK**.

Alternatively, if you have recently loaded your required image, your file appears as a recently-used file on the **File** menu.

Note *If you load your image from the recently used file list, MDW automatically loads the image using the command-line arguments that you specified in the previous run.*

4.4.2 Executing an image

You can run your program in MDW in one of two ways. First, you can execute the entire program; MDW will halt execution at any breakpoints or watchpoints encountered. Second, you can step through the code one line at a time, stepping into or over procedures as needed. Stepping through your image is covered in the next section. To execute your image:

- Select **Go** from the **Execute** menu, or
- Click the **Go** button.

While the program executes, the Console Window is activated and the program code is displayed in the Execution Window.

Execution continues until:

- a breakpoint halts the program at a specified point
- a watchpoint halts the program when a specified variable or register changes
- the image requires input
- the Server stops the core because of a run control option. (See ***3.9 Run Control*** on page 3-22 for more information)
- you stop the program by clicking the **Stop** button. You can then continue execution from the point where the program stopped using **Go** or **Step**.

If the program is failing to respond, you can also abort program execution.

Notes *(1) If you wish to execute your program again, you must first reload it.*

(2) If you are in the ARM Project Manager (APM), you can click the **Execute** *button: the image will be built if necessary, the Debugger will be started, and your image will be loaded and executed.*

The executable used for debugging can be selected by changing the appropriate variable—in other words, from the **Project** *menu select* **Edit Variables for [projectname].apj** *. For further details, refer to* ***4.4.9 Configuring the APM to invoke MDW*** *on page 4-17.*

4.4.3 Stepping through an image

If you want to follow your image's execution more closely, you can step through the code in the following ways:

Step to the next line of code:

- Select **Step** from the **Execute** menu, or
- Click the **Step** button.

The program moves to the next line of code, which is highlighted in the Execution Window. Function calls will be treated as one statement.

If only C code is displayed, **Step** moves to the next line of C. If disassembled code is interleaved with C source, **Step** moves to the next line of disassembled code.

Step In to a function call:

- Select **Step In** from the **Execute** menu, or
- Click the **Step In** button.

The program moves to the next line of code. If the code is in a called function, the function source is displayed in the Execution Window, and the current code line is highlighted.

Step Out of a function

- Select **Step Out** from the **Execute** menu, or
- Click the **Step Out** button.

The program completes execution of the function and halts at the line immediately following the function call.

Run execution to the cursor:

1. Position the cursor in the line where execution should stop.
2. Select **Run to Cursor** from the **Execute** menu or click the **Run to Cursor** button.

This executes the code between the current execution and the position of the cursor.

Note *Be sure that the execution path includes the statement selected with the cursor.*

4.4.4 Setting breakpoints and watchpoints

Breakpoints and watchpoints are used to stop program execution when a selected line of code is about to be executed or when a specified condition occurs. There are two types of breakpoints and watchpoints; simple and complex. This section discusses simple breakpoints and watchpoints; complex breakpoints and watchpoints are discussed in ***4.7 Setting and Editing Complex Breakpoints and Watchpoints*** on page 4-31.

Breakpoints

To set a simple breakpoint on a line of code:

- Double-click on the line where you want to set the breakpoint.

OR

1. Position the cursor in the line where the breakpoint is to be placed.
2. Select **Toggle Breakpoint** from the **Execute** menu or click the **Toggle Breakpoint** button.

A new breakpoint is displayed as a red marker in the left-hand pane of the Execution Window, the Disassembly Window or the Source File Window. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function that you clicked on in step 1.

In a line with several statements, it is possible to set a breakpoint on an individual statement, as demonstrated in the following example:

```
int main()
{
   hello(); world();
}
```

If you position the cursor on the word `world` and click the **Toggle Breakpoint** button, `hello` will be executed, but execution will halt before `world` is executed.

If you want to see all of the breakpoints set in your executable image, open the Breakpoints Window by selecting **Breakpoints** from the **View** menu.

To set a simple breakpoint on a function:

1. Display a list of function names in the Function Names Window, by selecting **Function Names** from the **View** menu.
2. Select **Toggle Breakpoint** from the **Function Names Window** menu or click the **Toggle Breakpoint** button.

The breakpoint is set at the first statement of the function. This method also works for the Low Level Symbols Window, but the breakpoint will be set to the first machine instruction of the function (the beginning of its entry sequence).

Watchpoints

To set a simple watchpoint:

1. Select the variable, area of memory or register you want to watch.
2. Select one of the following:
   - Select **Toggle Watchpoint** from the **Execute** menu.
   - Select the **Toggle Watchpoint** option from the window's menu.
   - Click the **Watchpoint** button.

If you want to see all of the watchpoints set in your executable image, open the Watchpoints Window by selecting **Watchpoints** from the **View** menu.

4.4.5 Removing a breakpoint or watchpoint

To remove a breakpoint:

1. Double-click on a line containing a breakpoint (highlighted in red) in the Execution Window.
2. Select **Toggle Watchpoint** from the menu.

OR

1. Select **Breakpoints** from the **View** menu to display a list of breakpoints in the Breakpoint Window.
2. Select the breakpoint you wish to remove.
3. Click the **Toggle breakpoint** button or press the **Delete** key.

To remove a watchpoint:

1. Select **Watchpoints** from the **View** menu to display a list of watchpoints in the Watchpoint Window.
2. Select the watchpoint you wish to remove.
    a) Click the **Toggle Watchpoint** button or press the **Delete** key, or
    b) Position the cursor on a variable or register containing a watchpoint and right-click and select **Toggle Watchpoint** from the menu.

Note *If you set a watchpoint on a local variable, you will lose the watchpoint as soon as you leave the function which uses the local variable.*

4.4.6 Examining and setting variables and registers

Using the Debugger, you can display and modify the contents of the variables and registers used by your executable image. This section briefly introduces the display and modification features. For more information on variables and registers, see ***4.6.2 Debugger internal variables*** on page 4-22.

Variables

To display a list of variables

- Select **Local** from the **Variables** sub-menu off the **View** menu or click the **Locals** button, or
- Select **Global** from the **Variables** sub-menu off the **View** menu.

A Locals or Globals Window is displayed listing the variables currently active.

To modify a variable's value:

1. Select **Global** or **Local** from the **Variables** sub-menu off the **View** Menu.
   A Locals/Globals Window is displayed with the currently active variables.
2. Double-click on the value of the variable in the right-hand pane of the window.
   The Modify Item dialog is displayed.
3. Type in the new value for the variable.
4. Click **OK**.

Registers

To display a list of registers:

- Select a mode from the **Registers** sub-menu off the **View** menu. The registers are displayed in the appropriate Registers Window.
- To display a list of registers for User mode, click the **User Regs** button.

To modify a register's value:

1. Select a register mode from the **Register** sub-menu off the **View** menu. The registers for that mode are displayed in a Registers Window.
2. Double-click on the register to be modified. The Modify Item dialog is displayed.
3. Type in the new value for the register.
4. Click **OK**.

4.4.7 Examining memory

Using the Debugger, you can display memory locations.

To display a particular area of memory:

1. Select **Memory** from the **View** menu or click on the **Memory** button. The Memory Address dialog is displayed.
2. Enter the address as hex (prefixed by `0x`) or decimal.
3. Click **OK**.

The Memory Window is opened to display the area of memory requested.

Once you have opened the Memory Window you can display other areas of memory by using the scroll bars or by entering another address.

To enter another address:

1. Select **Goto** from the **Search** menu or select **Goto Address** from the **Memory Window** menu. The **Goto Address** dialog is displayed
2. Enter an address.
3. Click **OK**.

See ***4.8.1 Saving or changing an area of memory*** on page 4-34 for more information on working with areas of memory.

4.4.8 Reloading the image

Once you have executed your image, if you want to execute it again, you must reload it. To reload your executable image, select **Reload Current image** from the **File** menu or click the **Reload** button.

4.4.9 Configuring the APM to invoke MDW

To configure the APM to invoke MDW:

1. Open a project (settings are remembered on a project by project basis).
2. Select **Project**, then **Edit Variables for [projectname].apj**
3. Select **adw** from the variables list.
4. Set its value to **mdw**.
5. Press OK.

Note *There are several APM project templates installed with MDW. These templates are identical to those supplied with the ARM 2.11a toolkit, but invoke MDW rather than ADW. An 'M' has been appended to the name to indicate this.*

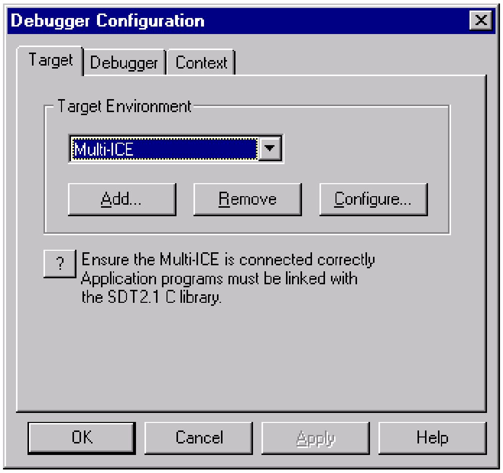
4.4.10 Exiting the Debugger

To close the Debugger, select **Exit** from the **File** menu.

4.5 Debugger Configuration

4.5.1 Target

Use this dialog to change the configuration used by the target environments that will be used during debugging. Accessed by selecting **Configure Debugger** from the **Options** menu.



Ensure the Multi-ICE is connected correctly
Application programs must be linked with
the SDT2.1 C library.

Target Environment	The target environment for the image being debugged.
Add	Display an Open dialog to add a new environment to the debugger configuration. (In order to see all `.dll` files, you must ensure that in Windows Explorer, the **Show all files** option is selected.)
Remove	Remove a target environment.
Configure	Display a configuration dialog for the selected environment.
?	Display a more detailed description of the selected environment.

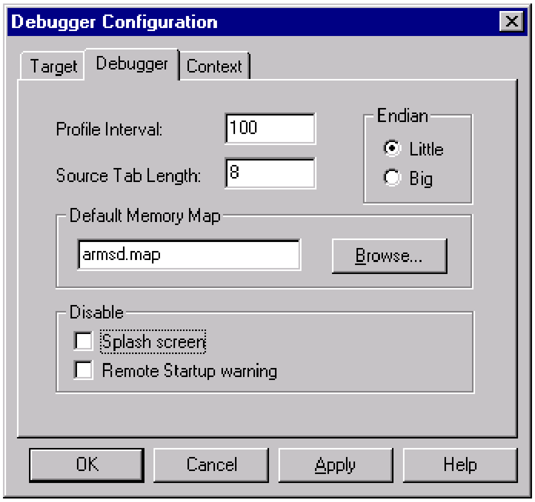
When your changes are complete:

- click **OK** to save and exit
- click **Cancel** to ignore all changes not applied and exit

Note **Apply** *is disabled for the Target page because a successful RDI connection has to be made first. When you click* **OK** *an attempt is made to make your selected RDI connection, if this is not successful, a warning is displayed.*

4.5.2 Debugger

Use this dialog to change the configuration used by the Debugger. Accessed by selecting **Configure Debugger** from the **Options** menu and clicking the **Debugger** tab.



Profile Interval: 100

Source Tab Length: 8

Endian
- ● Little
- ○ Big

Default Memory Map: armsd.map

Disable
- ☐ Splash screen
- ☐ Remote Startup warning

Profile Interval	This is not supported.	
Source Tab Length	When a source file is displayed this specifies the length of tab used in characters.	
Default Memory Map	The default memory map, the file that describes your memory layout**.** This is not required for Multi-ICE.	
Endian	Determines byte sex.	
	Little-endian	LOW addresses have the least significant bytes.
	Big-endian	HIGH addresses have the least significant bytes.
Disable Splash screen	When check-marked, stops display of the splash screen (the MDW startup box) when MDW is first loaded.	
Remote Startup warning	This is not supported for Multi-ICE. The remote startup warning is never given.	

When your changes are complete:

- click **OK** to save and exit
- click **Apply** to save
- click **Cancel** to ignore all changes not applied and exit

Notes *(1) When you make changes to the Debugger configuration, the current execution is ended and your program is reloaded.*

(2) For information on the **Context** *tab, refer to* ***4.9.1 Processor context*** *on page 4-38.*

4.5.3 ARMulator

MDW does not support the ARMulator. If you need to use the ARMulator, it is available in the ARM Software Development Toolkit.

4.6 Displaying Image Information

Certain information can be displayed by selecting the appropriate item from the View menu:

- Breakpoints
- Watchpoints
- Backtrace
- Functions
- Debugger Internals
- Registers

The windows used to display this information are described in ***4.3 The MDW Desktop*** on page 4-8.

Information not available directly from the **View** menu is discussed in this section.

4.6.1 Source files

Search paths

If you want to view the source for your program's image during the debugging session, the MDW needs to know how to find the files. A search path points to a directory or set of directories that are used to locate files whose locations are not referenced absolutely.

If you are developing your program using the ARM Project Manager, the search path for a newly-loaded image is added to the list of paths by reading the build directory from the image file.

If you are using the ARM command-line tools to build your project, you may need to edit the search paths for your image manually, depending on the options you chose when you built it.

If for some reason the files have moved since the image was built, the search paths for these files must be set up in the MDW, using the Add Path dialog (see below).

To display source file search paths, select **Search Paths** from the **View** menu.

The current search paths are displayed in the Search Paths window.

To add a source file search path:

1. Select **Add a Search Path** from the **Options** menu.
   The Select a file from the required directory dialog is displayed.
2. **Browse** for the directory you wish to add and highlight any file in that directory.
3. Click **OK**.

To delete a search path:

1. Select **Search Paths** from the **View** menu. The Search Paths Window is displayed.
2. Select the path to delete.
3. Press the **Delete** key.

Listing source files

To display a list of the current program's source files, select **Source Files** from the **View** menu.

The Source Files List Window is displayed.

Source files

Once you have a listing of source files in the Source Files List Window, you can select a source file to be displayed by double-clicking on a file name.

The file is opened in its own Source File Window.

Note *You can have more than one source file open at a time.*

4.6.2 Debugger internal variables

Debugger internal variables specific to Multi-ICE

The following debugger internal variables are specific to Multi-ICE:

`user_input_bit1` and `user_input_bit2`

These variables show the state of the two user input bits. These are not polled—they show the state at the time the Debugger Internals window was displayed.

`user_output_bit1` and `user_output_bit2`

These variables allow the user to alter the state of the user output bits. The user can only change these if the output bits are assigned to the TAP position to which the debugger is connected, and if the **Set by Driver** option is enabled. These are set on the Server using **User Output Bits**, found under the **Settings** menu—see ***3.6 User Output Bits*** on page 3-17. Again, these are not polled—they show the state at the time the Debugger Internals window was displayed.

`vector_address`

This provides support for WinCE-capable processors. It defaults to the Vector address that has been set up for the current processor, by reading the V bit from CP15 Register 1 on a WinCE-capable processor. It can be set explicitly to either 0 or 0xFFFF0000. For correct operation, it is essential that there is readable memory at the address to which it is sent.

`safe_non_vector_address`

This provides support for WinCE-capable processors, and defaults to 64KB. This variable must be set to the base address of a 64KB area of memory that is distinct from the 64KB block of memory starting at `$vector_address`.
The block of memory to which it is pointed should be "safe", in that the Multi-ICE DLL may cause some reads from this area to occur, and these reads should be harmless. In other words, they should not cause data aborts and should not do anything to I/O devices.

`arm9_restart_code_address`

Applicable only for ARM9T revision 0-based targets. This specifies an area of memory 32 bytes in size that Multi-ICE can use during execution restart requests. This area of program memory must be readable and writable, and must not be used for any other purpose.

`system_reset`

When read, this will always be 0. If written with a non-zero value, however, the target board will be reset using system reset. Multi-ICE will cause a system reset pulse of 250ms to occur.

`cp15_cache_selected (0 or 1: 0=D-Cache, 1=I-Cache)`

Only valid on Harvard-Architecture Processors. It indicates which 'version' of a CP15 register is read/written. For further information on accessing the CP15 registers, refer to ***Appendix C, CP15 Register Mapping***.

`cp15_current_memory_area (0—7: 0=Memory area 0 &ct)`

Selects the memory area to be used with accesses to register 6 on ARM740T and ARM940T processors. In the case of ARM940T processors, the `$cp15_cache_selected` variable decides if this is going to be a Data/Instruction area. For further information on the CP15 register, refer to ***Appendix C, CP15 Register Mapping***.

`cp_access_code_address`

This specifies an area of memory of at least 40 bytes in size, that can be used by Multi-ICE during read/write coprocessor operations. Multi-ICE will ensure that this memory is restarted to its original values after use. This area of memory must be readable and writable.

`semihosting_enabled` and `semihosting_dcchandler_address`

The first of these two is not a new variable, but there is now a new valid value which you can set it to. The 0 and 1 retain the same meaning as with other debug systems. The new value is 2, which requests that Debug Comms Channel-based semihosting is used. If this is selected, the value of `$semihosting_dcchandler_address` becomes significant, as explained below. Standard semihosting remains the default.

Semihosting off (`semihosting_enabled=0`)

This switches semihosting off.

Standard semihosting (`semihosting_enabled=1`)

Standard semihosting involves setting a breakpoint either on the SWI Vector itself or somewhere else in cooperation with the user's own SWI Handler (depending on the value of `$semihosting_vector`). There is a significant drawback to this mechanism: in a JTAG-based debugging system the processor must be halted in order to perform a semihosting operation, and remains halted for long enough to cause problems to realtime systems (with interrupt driven devices), often making semihosting not feasible for such systems.

Debug Comms Channel semihosting (`semihosting_enabled=2`)

Debug Comms Channel semihosting, however, does not cause the target processor to stop. Instead, a SWI Handler is installed in the target's memory which intercepts semihosting SWIs and sends a request for a semihosting operation to be sent up the processor's Debug Comms Channel. The Multi-ICE DLL notices this request and communicates with the SWI handler on the target, requesting that memory is read and written as necessary, and when the operation is completed, it informs the SWI handler that the operation has completed. The SWI Handler then returns to the user's program, and execution restarts from the instruction after the Semihosting SWI.

Debug Comms Channel-based Semihosting does not cause the processor to halt, and the SWI Handler resets the interrupt enabled state to that of its caller, and so this method of semihosting should be entirely suitable for realtime systems. It is also more useful for those systems that include more than one processor on a chip connected to a single Multi-ICE unit, because Debug Comms Channel-based semihosting does not interfere with automatic starting and stopping of processors, as can be requested from the Multi-ICE Server.

One disadvantage, however, is that the Debug Comms Channel cannot be used for other purposes if Debug Comms Channel-based semihosting is enabled.

The address at which the SWI Handler is installed in the target's memory is `$semihosting_dcchandler_address`, and the SWI handler is approximately 0.75Kb in size. The SWI handler is written to memory whenever Debug Comms Channel-based semihosting is enabled (setting `$semihosting_enabled` to 2) or `$semihosting_dcchandler_address` is changed and Debug Comms Channel-based semihosting is already enabled. It is vital that there is otherwise-unused RAM in the region where the debugger will attempt to install this SWI Handler. The default value for `$semihosting_dcchandler_address` is 0x10000 less than the top of memory (`$top_of_memory`).

It is possible to use the Debug Comms Channel SWI Handler together with a user's own SWI Handler. This is done by installing the user's own SWI Handler on the SWI Vector, and setting `$semihosting_vector` to be the address of a NOP instruction in the user's SWI Handler, which will be executed if the SWI is a semihosting SWI (or at least not one that the user's code handles). The registers at that point should be exactly as they were on entry to SVC mode when the SWI was first executed. The Debug Comms Channel SWI Handler will then deal with returning to the caller's code and reporting unknown SWIs.

Problems with `semihosting_enabled=2`

On some devices that utilize AMBA wrappers, coprocessor accesses do not work properly. In these devices, DCC-hosted semihosting (`semihosting_enabled=2`) will not work. For these devices, use `semihosting_enabled=1` (stop/start semihosting).

Debugger internal variables non-specific to Multi-ICE

The following debugger internal variables are non-specific to Multi-ICE:

`vector_catch`

Indicates whether or not execution should be caught when various conditions arise. The default value is `%RUsPDAifE`. Capital letters indicate that the condition is to be intercepted:

R	reset
U	undefined instruction
S	SWI
P	prefetch abort
D	data abort
A	address
I	normal interrupt
F	fast interrupt
E	unused

You can also set the variable to a numeric value which is to be interpreted as a bitmap, in the order set above.

`cmdline`

Argument string for the image being debugged.

`rdi_log`

RDI logging:

Bit 1	Bit 0	
0	0	Off
0	1	RDI on
1	0	Device Driver Logging on
1	1	RDI and Device Logging on

Table 4-2: RDI Logging

The remaining bits should be set to zero.

`semihosting_vector`

The address of SWI handler used for semihosting.

`clock`

The number of microseconds since simulation started. This is not supported by Multi-ICE.

`top_of_memory`

Under Angel, this variable gives the total amount of memory normally on the board. If more memory is added to the board, change this variable to reflect the new amount of memory.

`pr_linelength`

The default number of characters per line (initially set to 72).

`inputbase`

The base for input of integer constants (initially set to 10).

`format`

The default format for printing integer values (initially set to `%`d).

`sourcedir`

The directory containing source code for the program being debugged. This is initially set to the current directory unless your application was built by the ARM Project Manager, in which case this variable points to the source directory known by APM (initially set to NULL).

`result`

The integer result returned by the last called function (invalid if none, or if an integer result was not returned). This variable is read-only.

`fpresult`

The floating-point value returned by the last called function (invalid if none, or if a floating point value was not returned). This variable is read-only.

`type_lines`

The default number of lines for the type command.

`list_lines`

The default number of lines for list command (initially set to 10).

`examine_lines`

The default number of lines for examine command (initially set to 8).

`echo`

This is non-zero if commands from obeyed files are to be echoed (initially set to 0).

`icebreaker_lockedpoints`

If the user writes to a breakpoint register in the EmbeddedICE macrocell, Multi-ICE will set the appropriate bit to indicate that this breakpoint has been locked, and will not be used by Multi-ICE for breakpoints set through the debugger. The user can then clear this bit to indicate that Multi-ICE can again make use of this breakpoint.

Bit 1 set	The user has claimed breakpoint 1
Bit 2 set	The user has claimed breakpoint 2

4.6.3 Local and global variables

A list of local or global variables can be displayed by selecting the appropriate item from the **View** menu; a Locals/Globals Window is displayed. You can also display the value of a single variable or you can display additional variable information from the Locals/Globals Window.

To display the value of a single variable:

1. Select **Expression** from the **Variables** sub-menu off the **View** menu.
2. Enter the name of the variable in the View Expression dialog.
3. Click **OK**.

Alternatively:

1. Highlight the name of the variable.
2. Select **Immediate Evaluation** from **Variables** sub-menu off the **View** menu or click the **Evaluate Expression** button.

In both cases, the value of the variable is displayed in an Expression Value information box and is recorded in the Command Window.

Note *If you select a local variable that is not in the current context, an error message is displayed.*

Display formats

If you are in the Locals or the Globals Window, Expressions Window or the Debugger Internals Window, you can change the format of a variable. The format of values displayed for variables can be modified using the same syntax as a `printf` format string in C. Format descriptors include those listed in ***Table 4-3: Display Formats*** on page 4-28.

Type	Format	Description
`int`	Only use this if the expression being printed yields an integer:	
	%d	Signed decimal integer (default for integers)
	%u	Unsigned integer
	%x	Hexadecimal (lowercase letters)
`char`	Only use this if the expression being printed yields an integer:	
	%c	Character
`char*`	%s	Pointer to character. Only use this for expressions which yield a pointer to a zero terminated string.
`void*`	%p	Pointer (same as `%.8x`)—for example, `00018abc`. This is safe with any kind of pointer.
`float`	Only use this for floating-point results:	
	%e	Exponent notation, for example, `9.999999e+00`
	%f	Fixed point notation—for example, `9.999999`
	%g	General floating-point notation, for example `1.1, 1.2e+06`

Table 4-3: Display Formats

To change the format of a variable:

1. Right click on the variable and select the **Change line format** from the Locals or Globals Window menu. The Display Format dialog is displayed.
2. Enter the display format.
3. Click **OK**.

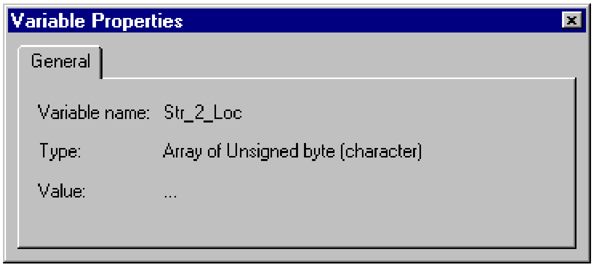
Note *If you change a single line, that line will not be effected by global changes.*

Tip *Leaving the Display Format dialog empty and clicking* **OK** *restores the default display format. This is the method to revert a line format change to the global format.*

Note *The initial display format of a variable declared as* `char[]=` *is special; the whole string is displayed, whereas normally arrays are displayed as ellipsis. If the format is changed it will revert to the standard array representation.*

Variable properties

If you have a list of variables displayed in a Locals/Globals Window, you can display additional information on a variable by selecting **Properties** from the window's menu (right-click on an item to display the window menu). The information is displayed in a dialog. HIGH addresses have the least significant bytes.



Variable name: Str_2_Loc
Type: Array of Unsigned byte (character)
Value: ...

Indirection

By selecting **Indirect through item** from the **Variables** menu you can display other areas of memory.

If you select a variable of integer type, the value is converted to a pointer (using sign extension where applicable) and the memory at that location is displayed. If you select a pointer variable, the memory at the location pointed to is displayed. You cannot select a void pointer for indirection.

4.6.4 Disassembly code

Disassembled code is a textual form of the machine code generated by the ARM C compiler or assembler.

You can display disassembled code in the Execution Window or in the Disassembly Window (select **Disassembly** from the **View** menu).

You can also choose the type of disassembled code to display by accessing the **Disassembly mode** sub-menu, which is on the **Options** menu. ARM code, Thumb code or both can be displayed, depending on your image.

To display an area of memory as disassembled code:

1. Select **Disassembly** from the **View** menu, or click the **Display Disassembly** button. The Disassembly Address dialog is displayed.
2. Enter an address.
3. Click **OK**.

The Disassembly Window is opened to interpret the memory as disassembly code.

To display or hide disassembled code in the Execution Window, select **Toggle Interleaving** from the **Options** menu.

Disassembled code is displayed in gray, the C code in black.

Once you have opened the Disassembly Window, you can display another address as disassembled code by using the scroll bars to search for an address by value or:

1. Select **Goto** from the **Search** menu.
2. Enter an address.
3. Click **OK**.

Specifying a disassembly mode

The MDW tries to interpret whether disassembled code is ARM code or Thumb code, but sometimes this is not possible, for example, if you have copied the contents of a file on disk to memory. To specify the type of code (ARM, Thumb or both) that you want to see when you display disassembly code in the Execution Window, select **Disassembly mode** from the **Options** menu.

4.6.5 Remote debug information

The RDI Log Window displays remote debug information; this comprises the low-level communication messages between the MDW and the target processor.

To display remote debug information (RDI) select **RDI Protocol Log** from the **View** menu. The RDI Log Window is displayed.

Using the RDI Log Level dialog (select **Set RDI Log Level** from the **Options** menu) you can select the information that will be displayed in the RDI Log Window:

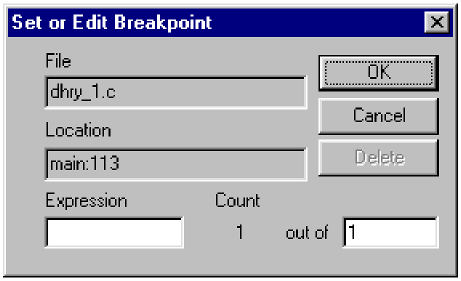
Bit 0	RDI level logging on/off
Bit 1	Device driver logging on/off

4.7 Setting and Editing Complex Breakpoints and Watchpoints

4.7.1 Breakpoints

A breakpoint is a point in the code where your program will be halted by the MDW. Once you have set a breakpoint it will appear as a red marker in the left-hand pane of a Source or Execution window.

When you set a complex breakpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Breakpoint dialog.



File: dhry_1.c
Location: main:113
Expression:
Count 1 out of 1

This dialog contains the following fields:

File	The source file that contains the breakpoint. This field is read-only.
Location	The position of the breakpoint within the source file. This field is read-only.

Expression	An expression that must be true for the program to halt, in addition to any other breakpoint conditions. Use C-like operators such as:
	`i < 10`
	`i != j`
	`i != j + k`

Count	The program halts when all the breakpoint conditions apply for the number of times specified.

To set or edit a complex breakpoint on a line of code:

1. Double-click on the line where you want to set a breakpoint or on an existing breakpoint position.

   The Set or Edit Breakpoint dialog is displayed.
2. Enter or alter the details of the breakpoint.
3. Click **OK**.

The breakpoint is displayed as a red marker in the left-hand pane of the Execution, Source File or Disassembly Window. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function that you highlighted in step 1.

To set or edit a complex breakpoint on a function:

1. Display a list of function names in the Function Names Window.
2. **Select Set or Edit Breakpoint** from the **Function Names Window** menu.
3. The Set or Edit Breakpoint dialog is displayed. Complete or alter the details of the breakpoint.
4. Click **OK**.

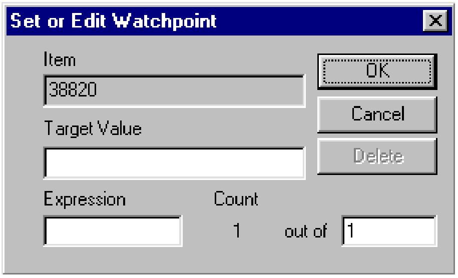
To set a breakpoint on a low-level symbol:

- Type `break@`*`symbolname`* in the Command Window, or
- Display the Low Level Symbols Window and set a breakpoint on the required symbol.

4.7.2 Watchpoints

A watchpoint halts a program when a specified register or variable is changed.

When you set a complex watchpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Watchpoint dialog.



Item: 38820

Target Value:

Expression:

Count: 1 out of 1

This dialog contains the following fields:

Item	The variable or register to be watched.
Target Value	The value of the variable or register that will cause the program to halt. If this value is not specified, any change in the item's value will cause the program to halt, dependent on the other watchpoint conditions.

Expression	Any expression which must be true for the program to halt, in addition to any other watchpoint conditions. As with breakpoints, use C-like operators such as:
	`i < 10`
	`i != j`
	`i != j + k`

Count	The program halts when all the watchpoint conditions apply for the number of times specified.

To set a complex watchpoint:

1 Select the variable or register you want to watch.

2 Select **Set or Edit Watchpoint** from the **Execute** menu.
The Set or Edit Watchpoint dialog is displayed.

3 Specify the details of the watchpoint.

4 Click **OK**.

To edit a complex watchpoint:

1 Display current watchpoints by selecting **Watchpoints** from the **View** menu.

2 Select the watchpoint you want to edit.

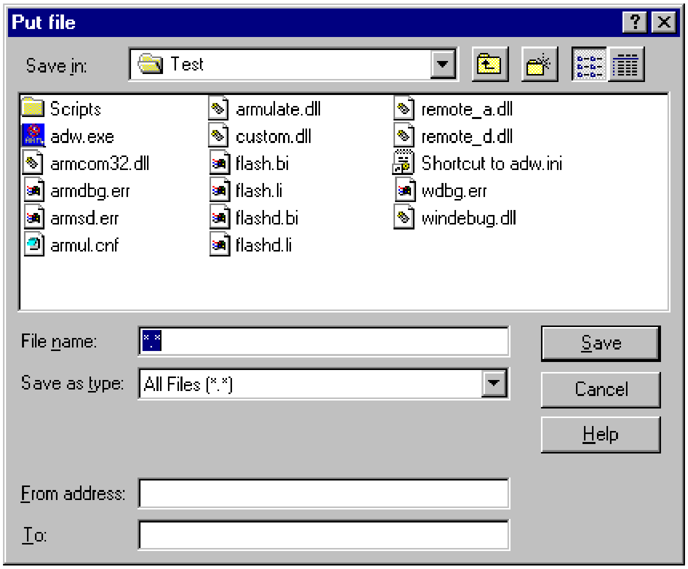
3 Modify the details as required.

4.8 Other Debugging Functions

4.8.1 Saving or changing an area of memory

To save an area of memory to a file on disk:

1 Select **Put File** from the **File** menu. The Put file dialog is displayed.



Save in: Test

File name: *.*

Save as type: All Files (*.*)

2 Select the file to write to.

3 Enter a memory area (in hexadecimal) in the **From address** and **To** fields.

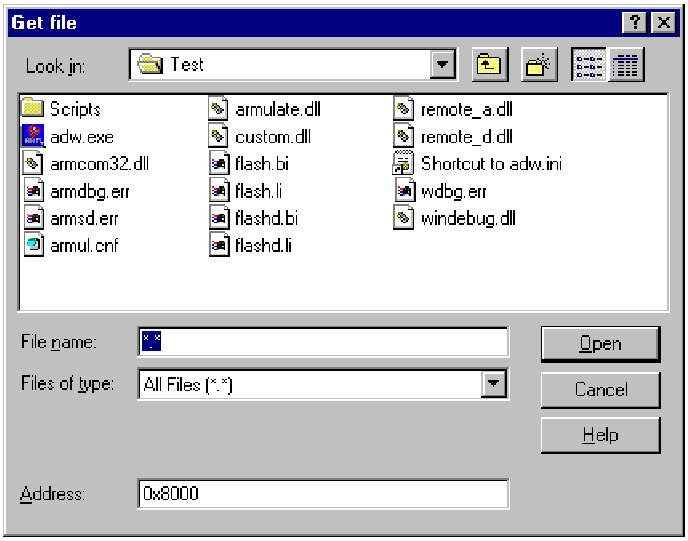
4 Click **Save**.

5 Click **OK**.

Note *The output is saved as a binary data file.*

To load a file on disk to memory:

1. Select **Get File** from the **File** menu. The Get file dialog displayed.



File name: *.*

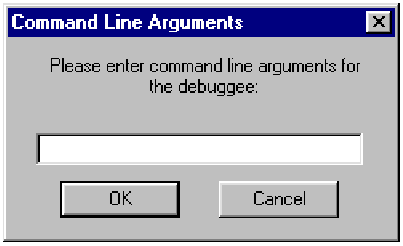
Files of type: All Files (*.*)

Address: 0x8000

2. Select the file you want to download.
3. Enter the start address (in hexadecimal) where the file is to be loaded.
4. Click **Open**.

4.8.2 Specifying command line arguments for your program

1. Select **Set Command Line Args** from the **Options** menu. The Command Line Arguments dialog is displayed.



2. Enter the command line arguments for your program.
3. Click **OK**.

Note *You can also specify command line arguments when you load your program in the Open File dialog or by changing the Debugger internal variable,* `$cmdline`.

4.8.3 Using command line debugger instructions

If you are used to using the ARM Command Line Debugger you may prefer to use the same set of commands from the Command Window.

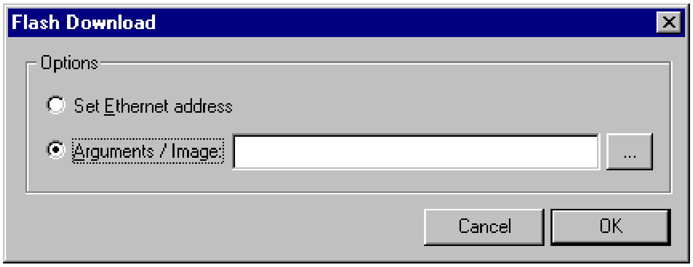
To open this window select **Command** from the **View** menu.

The Command Window displays a `Debug:` command line. You can enter ARM Command Line Debugger commands at this prompt. The syntax used is the same as that used for armsd. Type `help` for information on the available commands.

Refer to *ARM Software Development Toolkit Reference Guide* (ARM DDI 0041) for more information on the Command Line Debugger.

4.8.4 Flash download

The Flash Download dialog is used to write an image to the flash memory chip on an ARM Development Board or any suitably equipped hardware.



Set Ethernet Address

After writing an appropriate image to the flash memory (for example, Angel with ethernet support), this option sets the ethernet address (not necessary if you have built your own Angel with address compiled in). When you click **OK**, you are prompted for the IP address and netmask (for example, 193.145.156.78).

Arguments/Image

Specifies the arguments or image to write to flash. Use the **Browse** button to select the image.

Note *You can build your own flash image using the example in the Target Development System User Guide* (ARM DUI 0061). *MDW looks for one of the following images—*`flash.li` and `flash.bi`*—depending on whether you are using big- or little-endian. It looks first in the current working directory and then in the Toolkit* `bin` *directory.*

4.8.5 Channel viewers

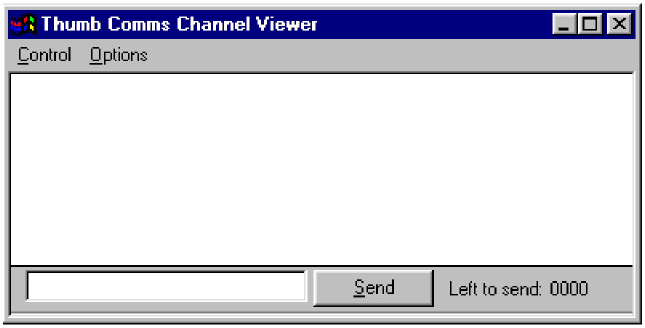
Debug communication channels can be accessed using a Channel Viewer. An example channel viewer is supplied with MDW (`MTHUMBCV.DLL`) or you can provide your own viewer.

To use the channel viewer supplied with MDW, semihosting has to be set to 0. This can be done from the Debugger Internals window (see ***Debugger Internals*** on page 4-9).

To select a Channel Viewer:

1. Ensure that Debug Comms Channel semihosting is disabled. (See ***semihosting_enabled and semihosting_dcchandler_address*** on page 4-23.)
2. Select **Configure Debugger** from the **Options** menu.
3. Select the **Channel Viewer Enabled** option. The **Add** and **Delete** buttons are activated.
4. Click the **Add** button and a list of `.DLL`s will be displayed.
5. Select the appropriate `.DLL` and click the **Open** button.

`MTHUMBCV.DLL` provides the following viewer:



The window has a dockable dialog bar at the bottom, this is used to send information down the channel. Typing information in the edit box and clicking the **Send** button will store the information in a buffer, the information is then sent when requested by the target. The **Left to send** counter displays the number of bytes that are left in the buffer.

Sending information

To send information to the target, type a string into the edit box on the dialog bar and click the **Send** button. The information is sent when requested by the target, in ASCII character codes.

Receiving information

The information received by the Channel Viewer is converted into ASCII character codes and, if the channel viewers are active, are displayed in the window. If, however, `0xffffffff` is received, the following word is treated and displayed as a number.

4.9 Using MDW with Piccolo

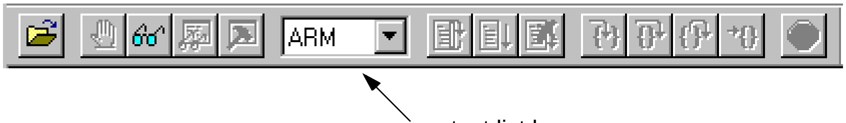
4.9.1 Processor context

When you are using the MDW with the Piccolo coprocessor, you can change the user interface mode, or *context*, to reflect the processor on which you are focussing.

For example, if you are in the ARM context, the pulldown menu option **View Registers** shows the ARM registers; if you are in the Piccolo context, the same option shows the Piccolo registers.

Some pulldown menu options and toolbar options are not available for all processors. These options are grayed when they are unavailable.

The current context is displayed in the context list box on the main toolbar.



context list box

If you are using the default setup, you can change the context by:

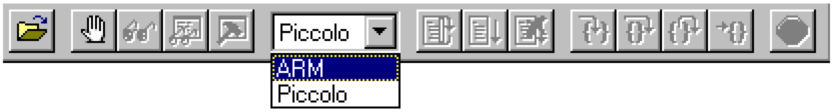
- selecting from the list box on the main toolbar, or
- selecting the Execution Window, or
- displaying an Execution Window menu

These methods are described below.

Selecting context using the toolbar

The list box on the main menu bar shows the current context.

1 Click the down arrow on the context list box.



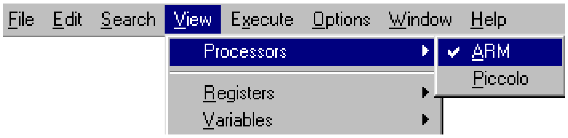
2 Select the processor you require from the list.
When you select a processor, the main menu and toolbar options are changed.

Selecting context by execution window

To select the Execution Window using the **View** menu:

1. Select **Processors** from the **View** menu.

   The processor with the current context is check-marked.



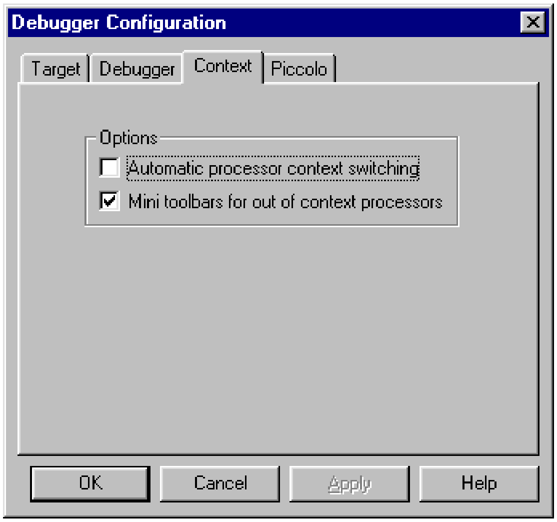
2. Select the required processor from the **Processors** menu.

   Note that:

   - the processor's Execution Window has the focus
   - the context list box on the toolbar shows the selected processor
   - the main menu and toolbar options are changed to reflect the processor

Alternatively, you can change context by selecting the appropriate Execution Window.

You can use the **Configure Debugger** option on the **Options** menu to disable context change by selecting the Execution Window. For example, you may want to do this if you switch between the ARM and Piccolo Execution Windows frequently, but you want to remain in the ARM context all the time. Refer to the screen shot below.



Options

☐ Automatic processor context switching

☑ Mini toolbars for out of context processors

1. Select **Configure Debugger** from the **Options** menu.
2. Select the **Context** tab.
3. Deselect the **Automatic processor context switching** check box.
4. Select **OK**.

Selecting context by execution window menu

When you right-click the mouse button to display an Execution Window menu, the context is automatically changed to match that of the Execution Window:

- the context list box on the toolbar shows the selected processor
- the main menu and toolbar options are changed

Mini toolbars and context

Each processor type has its own mini toolbar. (See ***4.10.3 The Piccolo mini toolbar*** on page 4-47 for a description of the Piccolo mini toolbar.) To display a mini toolbar for a particular processor:

1. Select the required processor context.
2. Select **Mini toolbar** from the **View** menu.

   The option is check-marked, and the mini toolbar is displayed.

To remove the mini toolbar, select the **Mini toolbar** option again.

By default, the mini toolbar remains on screen until you remove it. You can, however, configure the MDW so that only the mini toolbar for the current context is displayed:

1. Select **Configure Debugger** from the **Options** menu.
2. Select the **Context** tab.
3. Deselect the **Mini toolbars for out of context processors** check box.
4. Select **OK**.

4.9.2 Running an image

When you load an image containing Piccolo code, you can run the code in the usual way. The program will run until either ARM or Piccolo encounters a stop condition.

4.9.3 Stepping through an image

When you have loaded an image that contains Piccolo code, you can step through code by using the toolbar buttons, pulldown menu options, or Execution Window menu options in the usual way. However, the step options available to you are different depending upon the current context. These are summarized in ***Table 4-4: Piccolo and ARM step options***.

Processor context	Step In	Step Over	Step Out	Run to cursor
ARM	✔	✔	✔	✔
Piccolo	✘	✔	✘	✔

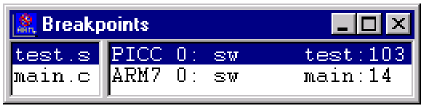
Table 4-4: Piccolo and ARM step options

Notes *(1) If you use the Execution Window context-sensitive menu for stepping, the context is automatically changed when you display the menu.*

(2) After stepping the ARM, Piccolo may or may not be on an instruction boundary; similarly, after stepping Piccolo the ARM may or may not be on an instruction boundary. In instances where any processor is not on an instruction boundary—that is, it is mid-step—it is not possible to change the contents of any of its registers. A processor can be moved to the next instruction boundary by selecting the appropriate context and executing a single step.

4.9.4 Breakpoints

The Breakpoints Window indicates the type of code in which the selected breakpoint appears. The debugger assigns a short, unique name to each processor in the target. The ARM is referred to by ARM7 0, and Piccolo by PICC 0. The screen shot below shows a Piccolo breakpoint on line 103 of file `test.s`, with an ARM breakpoint on line 14 of file `main.c`



```
test.s    PICC 0:  sw        test:103
main.c    ARM7 0:  sw        main:14
```

Note *In execution windows, breakpoints that are “out of context” appear in brown—for example, Piccolo breakpoints that are visible in the ARM execution window.*

4.9.5 Synchronous stopping with Piccolo

If the Piccolo PC is at a breakpoint, then a step is automatically performed before ‘go’ and ‘Run to Cursor’. This is not noticed during normal usage, but will cause the debugger to stop other processors executing if synchronous stopping is enabled.

4.9.6 Piccolo connection problems

If MDW is connected to the ARM (only) of an ARM + Piccolo core, and a second MDW attempts to connect to ARM + Piccolo of that core, then it is possible that both connections will fail.

4.10 Working with Piccolo

This section describes the features available when using the Piccolo context.

4.10.1 The Piccolo execution window

The Piccolo Execution Window is displayed automatically if you have connected to Piccolo.

You can also activate the Window manually by selecting from the **Processor** menu on the **View** pulldown menu.

Execution Window menu

The Piccolo Execution Window menu is slightly different from the ARM menu, in that the **Step In**, **Step Out** and **Disassembly Mode** are not available.

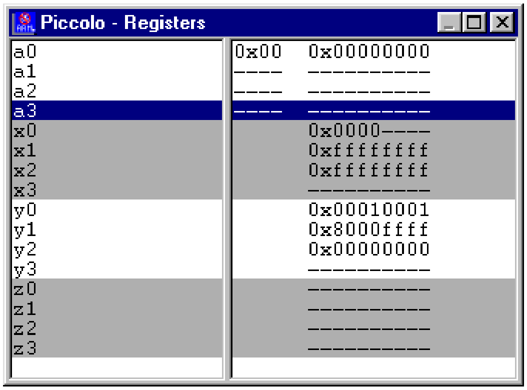
4.10.2 The Piccolo registers windows

To access the Piccolo Registers Windows:

1. Select the Piccolo context.
2. Select **Registers** from the **View** menu.
3. Select the register set you require from the **Registers** menu. You can choose from:
    - General Registers
    - Reorder Buffer
    - Output FIFO
    - Special Registers

For example, the Piccolo Registers Window is shown below.

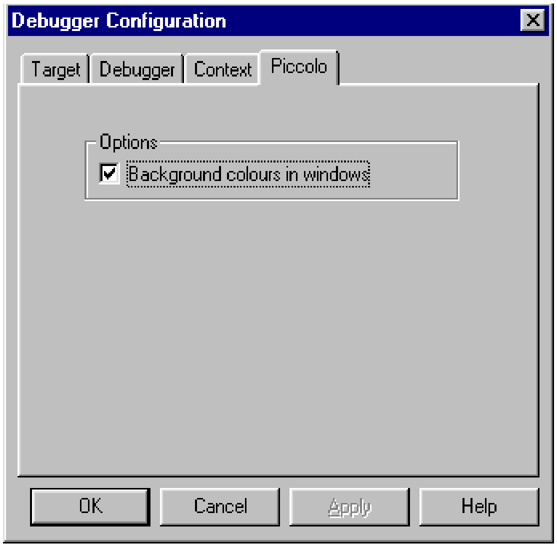
Alternatively, you can select the appropriate button on the Piccolo mini toolbar (see ***4.10.3 The Piccolo mini toolbar*** on page 4-47).



```
a0    0x00   0x00000000
a1    ----   ----------
a2    ----   ----------
a3    ----   ----------
x0           0x0000----
x1           0xffffffff
x2           0xffffffff
x3           ----------
y0           0x00010001
y1           0x8000ffff
y2           0x00000000
y3           ----------
z0           ----------
z1           ----------
z2           ----------
z3           ----------
```

By default, the Piccolo Registers Windows have background shading to group similar banks of registers together so that the Window is easier to read. If required, you can disable the background shading:

1. Select **Configure Debugger** from the **Options** menu.
2. Select the **Piccolo** tab.



Options

☑ Background colours in windows

3. Deselect the **Background colors in windows** check box.
4. Select **OK**.

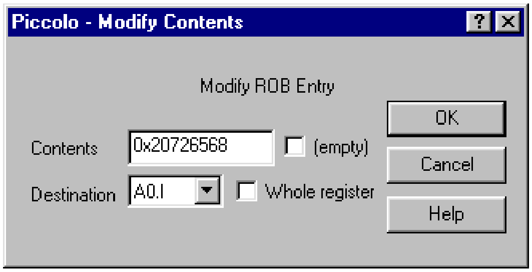
Registers window menu

The Piccolo Registers Windows menu is similar to the ARM menu, except:

- the **Indirect through item** option is not available
- the **Edit item contents** option displays different dialog boxes (see ***Editing register contents*** below)
- the **Change line format** and **Change window format** options lead to a submenu (see ***Changing register formats*** on page 4-46).

Editing register contents

When you double-click on a Piccolo register or select the **Edit item contents** option from the Registers Window menu, the appropriate Piccolo - Modify Contents dialog is displayed. For example, the dialog for the Reorder Buffer is shown below.


Piccolo - Modify Contents

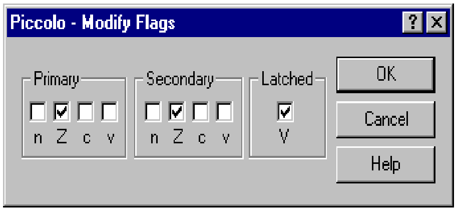
Modify ROB Entry

Contents 0x20726568 ☐ (empty)

Destination A0.l ☐ Whole register

The displayed dialog may depend on the formatting of the current line. For example, if you are viewing the register as halfwords, it may be edited as halfwords. Refer to ***Changing register formats*** on page 4-46.

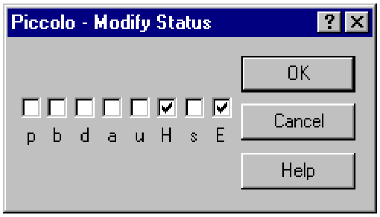
If you have selected a Flag or Status register in the Special Registers Window, the following dialogs are displayed:


Piccolo - Modify Flags

Primary: ☐ n ☑ Z ☐ c ☐ v

Secondary: ☐ n ☑ Z ☐ c ☐ v

Latched: ☑ V

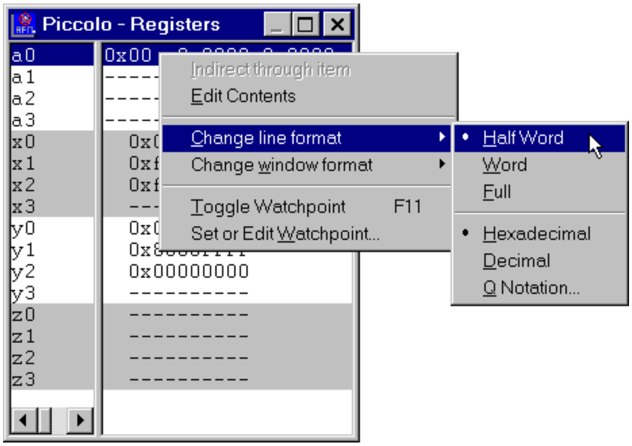

Piccolo - Modify Status

☐ p ☐ b ☐ d ☐ a ☐ u ☑ H ☐ s ☑ E

For details about how to use these dialogs, refer to the online Help text.

Changing register formats

When you select the **Change line format** or **Change window format** option from the Piccolo Registers Window menu, a submenu is displayed, as shown below.



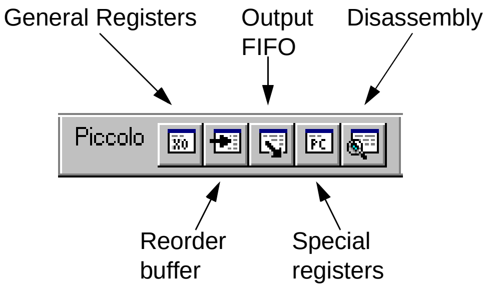
The selected register's current format and base are respectively denoted by radio buttons. Select the new word format or base as required. To change both, repeat the procedure.

Half word formatting of 32-bit registers may be allowed when the register is considered in two 16-bit halves. When a register is displayed in two halves, the Modify Contents dialog allows independent modification of the two halves. In the case of Piccolo accumulators, guard bits may also be modified independently if the register is formatted as halfwords. Full formatting only applies to Piccolo 40-bit accumulators.

Register contents are viewable in hexadecimal, decimal, or fractional form using Q notation. To view in fractional form, select the **Q Notation...** option from the submenu (see above), and enter the number of fractional bits in the dialog box provided.

4.10.3 The Piccolo mini toolbar

The Piccolo mini toolbar provides shortcuts to viewing registers, and to disassembly. To display the mini toolbar if it is not already present, select **Mini toolbar** from the **View** menu when you are in the Piccolo context.



You can also access these options from the **View** pulldown menu when you are in the Piccolo context.

See ***4.10.2 The Piccolo registers windows*** on page 4-43 for a description of the Piccolo Registers Windows.

4.11 Command Line Options for MDW

This section lists and describes the command line options available for MDW.

4.11.1 Command line

The basic command line is:

```
mdw_pathname -switch filename
```

where:

mdw_pathname	specifies the full pathname of the MDW executable
switch	is one of the options listed below
filename	is the full pathname of the file to use with the switch

4.11.2 MDW-specific options

`-debug` *image_name*	loads *image_name* for debugging
`-exec` *image_name*	loads and executes *image_name*
`-reset`	resets the MDW registry to the original default settings
`-nologo`	stops the splash screen being displayed on startup
`-nomainbreak`	stops MDW from trying to set a breakpoint at `Main()` when loading an image
`-script` *script_name*	Obeys the *script_name* on startup. This is equivalent to typing the following line as soon as the debugger starts up: `obey` *script_name*

4.11.3 Piccolo-specific options

`-noshading`	turns off shading on the Piccolo register windows
`-noautocontext`	turns off automatic switching between processor contexts
`-nootherminis`	hides mini toolbars when not in context

4.11.4 MDW support option

This option allows you to run multiple instances of MDW so that you can have a debugger running for each chip.

`-session` *session_name*	starts a session with the name *session_name*

Using a batch file

To run multiple instances of MDW from a batch file, you need to add the word `start` to the beginning of each command line. For example:

```
start mdw -session debug1
start mdw -session debug2
start mdw -session debug3
```

This allows each instance of the debugger to run in parallel. If you do not add `start` to each line, you must quit from the first debugger before the next listed debug session can start.

Session switches

These can be used to start multiple sessions from within a batch file. For example:

```
start c:\mdw\mdw.exe -session Smith -debug c:\images\dhry.axf
start c:\mdw\mdw.exe -session Jones -debug c:\images\flash.axf
```

Quitting from batch files

Under Windows 95, you must use the `exit` command to quit from the batch file. Under Windows NT, the batch file quits automatically when it has executed all its commands.

4.11.5 Examples

This example loads and executes the file `dhry.axf`:

```
c:\mdw\mdw.exe -exec c:\mdw\examples\dhry\dhry.axf
```

This example loads the image `dhry.axf` for debugging:

```
c:\mdw\mdw.exe -debug c:\mdw\examples\dhry\dhry.axf
```

This example stops MDW from trying to set a breakpoint at `Main()` when loading `dhry.axf` and does not display the MDW startup banner:

```
c:\mdw\mdw.exe -nologo -nomainbreak -debug c:\mdw\examples\dhry\dhry.axf
```

This is equivalent to typing the command `obey dccsemi.txt` as soon as the debugger starts up:

```
c:\mdw\mdw.exe -script c:\mdw\examples\dccsemi.txt
```

5

TAPOp Procedure Calls

This chapter defines the software interface between a Multi-ICE Server and a debug client as used by the ARM Multi-ICE DLL for ARM’s debugger. It provides a complete function reference and programming guidelines for writing client side drivers to interface with hardware connected to Multi-ICE.

TAPOp Procedure Calls

5.1 Introduction

The TAPOp interface allows ARM processors or third-party devices on an ASIC to be accessed via a Multi-ICE Server. This allows users to attach applications to any ARM and/or non-ARM elements on an ASIC. For example:

- DSP debuggers
- cache reader
- communications channel drivers
- test applications

Test Access Port (TAP) controller state transitions

Figure 5-1: Test access port (TAP) controller state transitions shows the TAP controller state transitions.

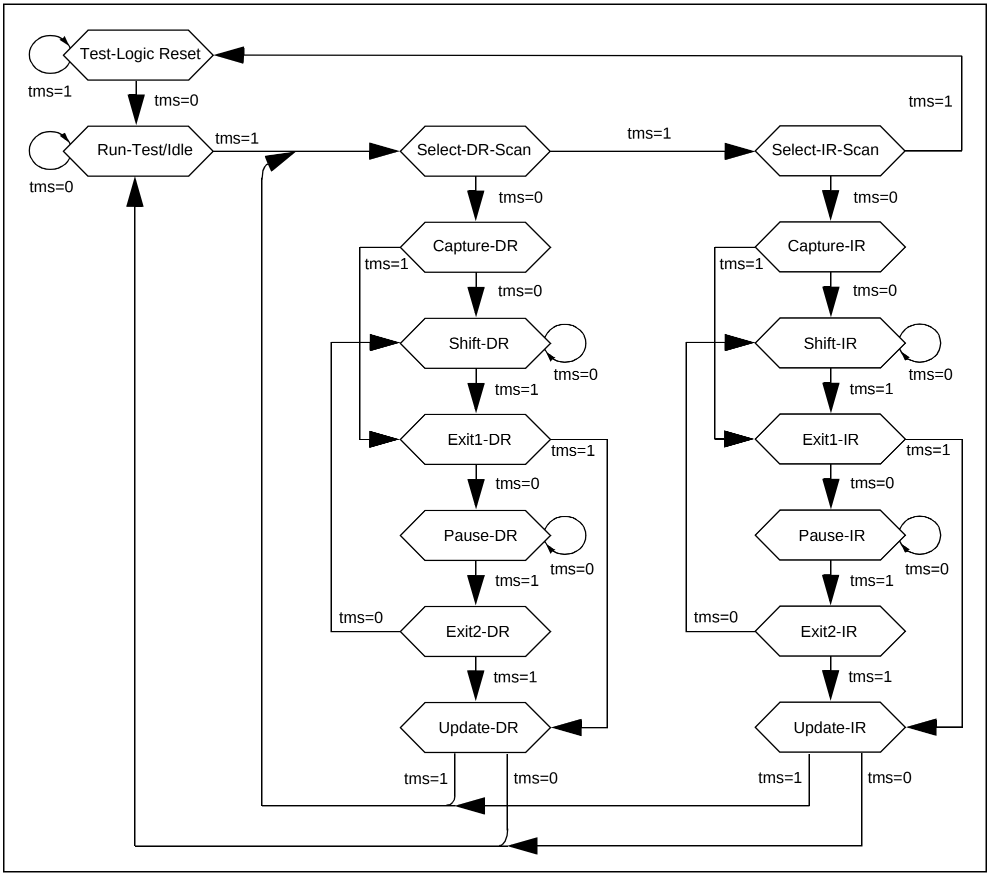


Figure 5-1: Test access port (TAP) controller state transitions

5.2 Accessing the Multi-ICE Server at the TAPOp Level

A client communicates with a Multi-ICE Server using remote procedure calls (RPCs). Multi-ICE uses ONC RPC in the TCP mode; this is a TCP/IP connection for making procedure calls on the Server. Because RPC uses TCP/IP as its transport mechanism, it is just as easy to connect to a Multi-ICE Server over a network or modem as it is to connect locally. Sources for the client end of the RPC-based TAPOp interface can be found on the installation disk.

In overview, a client opens a connection to the required device, makes RPC calls to the Server which scans data through that device's scan chains and, when finished, disconnects and allows another client to connect.

Each RPC call performs one *Test Access Port (TAP)* operation, hence the name TAPOp interface. For example, there are TAP operations to write a value to the *Instruction Register (IR)* or read a device scan chain. Because of the low level of the interface and the high overhead of RPC calls, it is possible to batch up multiple RPC calls into macros that are run by the Server. This is similar to JAVA applets downloaded from a web Server to a client (browser), because the browser is faster than the link. In Multi-ICE, the client downloads macros to the Server because the Server is faster than the link. This gives a significant performance improvement.

5.2.1 Connections

Opening a TCP connection

If a client wishes to communicate with a Server, `rpc_initialise` must be called to open a connection to the transport layer (TCP). The Server location is identified by a callback function `GetServerName` that the client must supply. This opens a two-way channel between the client and Server through which procedure calls can be made to the Server.

More than one TCP connection to the Server can be opened at the same time from the same client. The standard distribution of `rpcclient.c` opens two TCP connections by default and this can be used to overlap RPC calls to improve performance (this is done in the ARM debugger during downloads where multiple threads are used to pipeline RPC calls). When the client has finished, the TCP connection is closed using `rpc_finalise`.

Opening a TAPOp connection

After a TCP connection has been made to a Multi-ICE Server, the client must indicate to the Server which device to use. This is known as opening a TAPOp connection, and at any one time there is a single connection between the client and a single device on the Server. The TAPOp connection is identified by a connection ID; this ID is used in all subsequent calls to the Server. The client should close this connection when finishing a debug session.

At the same time, another TAPOp connection can be present to another device (even on the same TAP controller) using a different connection ID, but a single device can only be connected to a single connection ID. For example, if the client opens two TCP connections to the same device (as in the standard distribution `rpcclient.c`), calls from the client using both TCP connections must use the same TAPOp connection ID.

The connection ID is a logical identifier that the Server uses to recognize which client it is talking to, and it identifies a particular device on a particular TAP controller. It is allocated by the Server when the client makes a `TAPOp_OpenConnection` call to the Server.

Closing a TAPOp connection

To close the TAPOp connection, call `TAPOp_CloseConnection`. All the macros defined by the client are deleted and storage is freed.

5.2.2 Multiple clients of the TAPOp layer

There may be several simultaneous clients to the TAPOp layer, each one connected to a different TAP controller. Alternatively, clients can connect to the same TAP controller but only if they do not share any resources other than:

- the TAP controller IR
- the use of a scan chain select register

Two debuggers that access distinct sets of scan chains can both be clients (for example, a DSP scan chain hung off an extra scan chain of the ARM TAP controller). However, two debuggers that access the same scan chain cannot rely on the TAPOp interface to separate their accesses, particularly in the case of potentially shareable resources such as EmbeddedICE breakpoint registers. For example, two debuggers that talk to the same processor must co-operate at a higher level (the Remote Debugger Interface, RDI, is the most obvious place for an ARM processor).

In order to manage several clients using this interface simultaneously, most of the operations in this interface implicitly request that the client becomes the sole user of the Multi-ICE hardware, and also pass in a parameter indicating whether the client is ready to give up ownership of the Multi-ICE hardware after the operation has completed (this is known as deselecting the connection). If another client still has ownership of the Multi-ICE hardware, a call fails and returns an error code, and the operation is not performed. It is then the client's responsibility to try again.

When a client finally gives up ownership of the Multi-ICE hardware, the implementation of this interface guarantees that the next time that client successfully asks for ownership, the TAP controller will be in the same TAP state (for example, Run-Test/Idle) as it was when ownership was relinquished, and the same instruction will be in that client's TAP IR register. It also guarantees that the same scan chain will be selected in that TAP controller. In return, the implementation insists that ownership is relinquished only when the TAP controller is in either Run-Test/Idle or Select-DR-Scan state. This is only an issue for the `AnySequence` operations, because all other operations leave the TAP controller in one of these two states.

The Multi-ICE Server keeps track of the following for each TAPOp connection:

- the last value written to the IR for each TAP controller
- the state the TAP controller was in when ownership is relinquished, so that this state can be restored when ownership reverts to that client.
- the last scan chain selected using a `SCAN_N` instruction

5.2.3 TAP controller identification

The Multi-ICE Server can automatically detect the number of TAP controllers and any details required for each TAP controller—for example, the length of the IR register. There is also the option to manually load a configuration file.

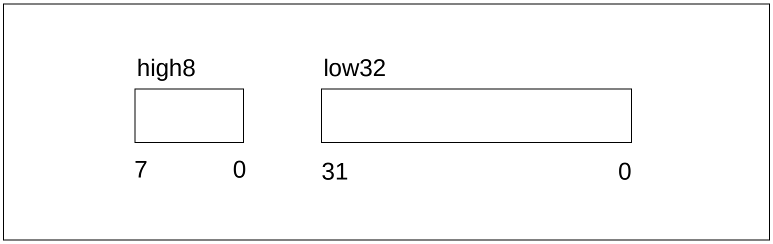
All incoming requests are labelled with the connection ID, which can be used to look up the position of the TAP controller in the scan chain, where 0 is nearest TDI. It is therefore necessary for a TAPOp client to inform the Server of the TAP controller position and the scan chains it needs when opening a TAPOp connection.

To get a list of devices for a particular Server, call `TAPOp_GetDriverDetails`. This returns a list of device names (for example, ARM7TDMI), their TAP positions (TAP 0 is nearest TDI) and flags indicating if the devices are connected.

5.2.4 Order of output of TDI/TMS bits passed over tapop.h

TDI, TDO and TMS data is passed over `tapop.h` as a 40-bit type called ScanData40 constructed from a 32-bit word containing the least significant bits and a byte containing the most significant bits. This type is defined in `tapop.h` .

```
typedef struct ScanData40 {
       unsigned32 low32;
       unsigned8  high8;
} ScanData40;
```



The bits are output as follows:

Bit 0 of low32—Bit 31 of low32

Bit 0 of high8—Bit 7 of high8

Similarly, for an output (TDO) block the first TDO bit input is placed in Bit 0 of low32, and the last in Bit 7 of high8.

If a data field is specified as reversed then the same data will be presented to (or read from for TDO) the Multi-ICE Reversed Data register, which will result in it being bit-reversed.

When scanning less than 40 bits, the data must be left justified so that, after bit-reversal, the data is at the correct end of the register.

5.2.5 Accessing long scan chains/using mask and offset parameters

`AccessDR_W` calls contain `WRoffset` and `WRmask` parameters. In addition `AccessDR_RW` calls contain an `RDoffset` parameter. The purpose of these parameters is to reduce the amount of bits that get written to less than 40 bits using `WRmask`, or increase the amount of bits that can be read/written to more than 40 bits by supplying a read/write offset (`RDoffset` and `WRoffset` parameters). For example, to access a 50-bit scan chain use the following steps:

1. Access 40 bits using `WRmask` = all ones. Since the parameter is a pointer to an array of bits, a special mechanism is built in for all ’1’ masks. If using a non-macro call, use NULL for all ones. For macroized calls, all the bits must be provided in two parameters (see `macrostruct.h` for details).
2. Access the remaining 10 bits using a 10-bit `WRmask` and a 40-bit offset.

5.2.6 Efficiency considerations

The TAPOp layer is similar to the functional interface used by EmbeddedICE. In a Multi-ICE system, however, the function is executed across an RPC layer to a PC that may be remote. As a result, when a large number of calls are made across this interface, there is a reduction in performance. In order to provide a way around this, you can use TAPOp macros to batch up TAPOp operations and make a single RPC call perform multiple TAPOp operations.

In general, there are limits to the number of TAPOp operations that can be grouped together because of the nature of the operation being performed. If TDO data causes a decision to be made in the TAPOp client, any macro must be finished before the next operation can be decided upon.

Large continuous data transfers prevent other TAPOp clients accessing the Server. It is recommended that data size should be limited to chunks of 10kB when communicating over a network to the Server.

5.2.7 Error detection and automatic connection de-selection

Whenever a TAPOp call is made, if a `TAPOp_Error` is returned which is not `TAPOp_NoError` or `TAPOp_UnableToSelect`, that error is considered fatal. This means that the TAPOp client may not be able to recover its session without losing data or at least aborting the operation.

Because the TAPOp interface can be used by several client debuggers at once, the connection that has an error is automatically deselected; this ensures that one TAPOp client receiving a fatal error does not block out others.

When a TAPOp call is made, it is likely that network traffic is present. To allow calls to be reliably passed over a network connection, a `TAPCheck` macro has been provided. This macro waits until the network is available before making a function call. This is defined in `macros.h`.

The `TAP_Check` macro should be used around all TAPOp and ARMTAP calls. It makes the call and performs error checking on the return value as follows:

1. If the call returns `TAPOp_NoError` then it does nothing.
2. If the call returns `TAPOp_UnableToSelect`, it will retry the same call.
3. If the call returns anything else, a call to `GetMICEflags` will be made in an attempt to diagnose the failure. If the flags indicate that the target power is off or has been off or the target has been reset then the returned error from the called procedure is overwritten with a more appropriate code.

You must define the following function to allow a fatal error to be dealt with cleanly. This function may be empty.

```
give_up(void)
```

Sample usage of `TAPCheck` can be seen in ***Example 2*** on page 5-15.

5.2.8 TAPSHARE.H header file

The following functions can be used to read and write data which is held by the Server on behalf of the various TAPOp clients attached to it, allowing these applications to communicate with each other in a limited manner.

There are two sets of data:

1. Data which is private to each TAP controller (processor).
   There are flags held for each processor; some are read by the debugger and some are written. The TAPOp module manages how these are passed on to other TAPs' (processor's) read flags in a way set up by the Multi-ICE Server.

   Note: Use of these flags is optional for a TAPOp client, but if they are used, they provide a way to start and stop processors almost synchronously when several applications are involved.
2. Common data.
   The Server does nothing with this data; it just maintains the data so the TAPOp clients can use it to communicate between themselves in the manner defined. The size of this data is arbitrary, and is currently four words (16 bytes).

Unlike most of the TAPOp operations, it is not necessary for a TAPOp client to have a selected connection in order to use the Private Data functions, because they do not affect the TAP controller in any way. However, to allow atomic Read-Modify-Write of the common data, the connection does have to be selected for accesses to the common data, so a deselect parameter is available.

Private flags

`TAPOp_ProcRunning`

A TAPOp client should set this flag when the processor starts executing code. It should be cleared when the processor halts. This can be used by the Multi-ICE Server to indicate whether or not other processors should be stopped according to the Multi-ICE user's requirements. Setting this bit causes the processor state display to change to [R]. This is a write-only flag for the TAPOp client.

`TAPOp_ProcHasStopped`
`TAPOp_ProcStoppedByServer`

These two flags are used to determine if and why a processor has stopped. A client should poll the `TAPOp_ProcHasStopped` flag when the processor is running. If it set, the `TAPOp_ProcStoppedByServer` flag will indicate why. If it is set, the Multi-ICE Server has stopped the processor because some other processor has stopped and a synchronized stop condition was set up. If `TAPOp_ProcStoppedByServer` is not set, the processor has stopped of its own accord—for example, because it hit a breakpoint. These flags are read-only for a client.

`TAPOp_DownloadingCode`

A debugger should set this flag immediately before starting to download code to the target processor. This allows a user output bit to be set when this occurs, which is potentially useful on a system that can switch between very slow and very fast clocks, as fast clocking will speed up download considerably. Similarly, when the download has completed, this bit should be cleared. Setting this bit causes the processor state display to change to [D] If `set on download` has been selected for user output bit1, setting this bit will also set the user output bit1.This flag is read-only for the Server.

`TAPOp_ProcStartREQ`
`TAPOp_ProcStartACK`

These two flags control synchronized starting of processors. If `TAPOp_UserWantsSyncStart` is set, the debugger should set `TAPOp_ProcStartREQ` to request the Server to start the processor. When all the debuggers have set their `TAPOp_ProcStartREQ` flags, the Server will start all processors together, and set the `TAPOp_ProcStartACK` flag. `TAPOp_ProcStartREQ` is read-only for the Server. `TAPOp_ProcStartACK` is read-only for a client.

`TAPOp_UserWantsSyncStart`
`TAPOp_UserWantsSyncStop`

These two flags are read-only, and will be set by the Server if the user has selected sync start and/or stop from the dialog. These flags are read-only for a client.

5.2.9 Flags returned by ReadMICEFlags

`TAPOp_FL_TargetPowerOffNow`	The target's power is off. This is an error condition.
`TAPOp_FL_TargetPowerHasBeenOff`	The target's power has been off since the last `TAPOp_OpenConnection` call was made. This is also an error condition; turning the target on and off in the middle of a work session is likely to cause problems.
`TAPOp_FL_InResetNow`	The target is currently in Reset; the target's reset signal is currently set. This is generally an error condition.
`TAPOp_FL_TargetHasBeenReset`	The target has been reset since the last `TAPOp_OpenConnection` call was made. This is generally an error condition.
`TAPOp_FL_UserIn1`	The state of the user-defined input signal 1 from Multi-ICE.
`TAPOp_FL_UserIn2`	The state of the user-defined input signal 2 from Multi-ICE.
`TAPOp_FL_UserOut1`	Current state of user-defined output 1 from Multi-ICE
`TAPOp_FL_UserOut2`	Current state of user-defined output 2 from Multi-ICE

5.2.10 Building a driver

The files required to build such applications are optionally copied to the following directory during installation of the Multi-ICE software:

```
C:\Multi-ICE\source
```

The pre-processor definitions WIN32 and MULTI_ICE need to be made.

The source files needed are:

```
mice_clnt.c
mice_xdr.c
oncrpc.lib
rpcclient.c
nonrpcclient.c
```

In addition, there are a number of header files required, which are copied into the same area as source files. These header files are listed in the ***Multi-ICE Installation Guide*** (ARM DSI0005).

Compiler differences

The source files supplied are for use with Microsoft Visual C++. If an alternative compiler is used, the definitions in `basetype.h` must be changed. It is imperative that `unsigned8`, `unsigned16` etc. are exactly 8, 16 bits in size.

5.3 Using TAPOp Macros

This section describes how to write and run TAPOp macros. The following list gives a functional summary of the macro procedure calls.

Creating macros	`TAPOp_DefineMacro`
Destroying macros	`TAPOp_DeleteMacro` `TAPOp_DeleteAllMacros`
Displaying macros (for debug)	`TAPOp_DisplayMacro`

Running macros	`TAPOp_RunMacro`
	`TAPOp_RunBufferedMacro`
	`TAPOp_FillMacroBuffer`

Synchronized stop / start macros	`TAPOp_SetControlMacros`

The macro procedure calls are given in full in the alphabetical listing of all procedure calls in ***5.5 TAPOp Procedure Call Alphabetic Reference***.

5.3.1 Writing a macro

The first step in writing a macro is to decide how instructions should be grouped together to optimize the speed of transfer. Better performance results from macros containing a large number of operations.

The second step is to 'convert' the parameters of the normal TAPOp operations to the macro versions. The structure for the data required for the macro versions of the instructions can be found in the header file `macstruct.h`.

In general, it is a good policy to get the non-macro version of a client working before attempting to turn it into a macro, as it is harder to debug once in macro format.

As an example, the prototype for the non-macro `ARMTAP_AccessDR_W` operation is:

```
extern TAPOp_Error ARMTAP_AccessDR_W(
       unsigned8   connectId,
       ScanData40  *TDIbits,
       unsigned8   TDIrev,
       unsigned8   len,
       unsigned8   WRoffset,
       ScanData40  *WRmask,
       unsigned8   nclks,
       unsigned8   deselect
);
```

For the `ARMTAP_AccessDR_W` instruction above, the macro version requires the parameters shown on the next page. The differences to note are:

1. The `connectId` is not present, this parameter is passed to the `TAPOp_RunMacro` function.
2. The `deselect` parameter is not present, this parameter is passed to the `TAPOp_RunMacro` function.

3 `TDIbits` and `WRmask` are passed as two parameters rather than as one `ScanData40` type

```
typedef struct MAC_ARMTAP_AccessDR_WIn {
      unsigned32  TDIbits1;
      unsigned8   TDIbits2;
      unsigned8   TDIrev;
      unsigned8   len;
      unsigned8   WRoffset;
      unsigned32  WRmask1;
      unsigned8   WRmask2;
      unsigned8   nclks;
} MAC_ARMTAP_AccessDR_WIn;
```

Next decide which parameters are fixed (define time) and which are variable (runtime). This depends on the specific programming task. During the implementation of the macro definition, the fixed parameters are easily recognized.

The process of defining and using macros is illustrated in the following examples using the `ARMTAP_AccessDR_W` instruction; other instructions are used in macros in a similar manner.

As with all TAPOp and ARMTAP instructions, the macro operations must be performed within the `TAPCheck` macro to ensure that appropriate error checking is performed. This is shown in the following examples.

5.3.2 Passing fixed and variable parameters to TAPOp macros

To simplify the passing of both fixed and variable parameters to TAPOp macros, a set of predefined C macros is provided in `macros.h`.

The macros that enter parameters use the following variables, which must be defined:

```
      int         ValPtr;
      unsigned8   Values[MACRO_ARGUMENT_AREA_SIZE];
```

The value of `MACRO_ARGUMENT_AREA_SIZE` is defined in `macros.h`.

C macros for passing fixed and variable parameters

The following macros return `IErr_NotEnoughMacroArgumentSpace` if:

`ValPtr > MACRO_ARGUMENT_AREA_SIZE`

`InitParams`

Resets `ValPtr`, required before the first fixed parameter of each macro ‘line’ at define-time, and before the first variable parameter at runtime.

`EnterParamBytes(void *byte_ptr, int nbytes)`

Enters a number of bytes into the `Values` array.

`EnterParamU8(unsigned8 byte)`

Enters a single byte into the `Values` array.

`EnterParamU16(unsigned16 halfword)`

Enters a 16-bit halfword as two bytes into the `Values` array.

`EnterParamU32(unsigned32 word)`

Enters a 32-bit word as 4 bytes into the `Values` array.

The following macros do not return (NR) an error if:

```
ValPtr > MACRO_ARGUMENT_AREA_SIZE
```

but an error message is displayed and the array boundary is protected. This uses `fprintf(stderr, “...”);`.

`NREnterParamBytes(void *byte_ptr, int nbytes)`

Enters a number of bytes into the `Values` array.

`NREnterParamU8(unsigned8 byte)`

Enters a single byte into the `Values` array.

`NREnterParamU16(unsigned16 halfword)`

Enters a 16-bit halfword as two bytes into the `Values` array.

`NREnterParamU32(unsigned32 word)`

Enters a 32-bit word as 4 bytes into the `Values` array.

5.3.3 Example 1

Example1 sends an LDMIA op-code to an ARM using `ARMTAP_AccessDR_W`. A `connectId` has previously been obtained with a `TAPOp_OpenConnection`. The instruction can be sent to the Data Register using the non-macro method:

```
ScanData40   opcode = { 0xE89E3FFF, 0};    /* op-code for LDMIA instr */
ScanData40   WRmask = { 0xFFFFFFFF, 0x1};  /* Write mask */
unsigned8    TDIrev = 1, len = 32, nclks = 1, deselect = 1;
TAPCheck (ARMTAP_AccessDR_W (connectId, &opcode, TDIrev, len, 0, &WRmask,
                              nclks, deselect));
```

To define the macro

To illustrate a simple macro, a macro ‘line’ of the above instruction is entered with all parameters fixed. Before you can use the `TAPOp_DefineMacro` instruction, the fixed parameters (see `MAC_ARMTAP_AccessDR_WIn`) must be put into an array, with attention to the type of the parameter.

```
#define MACRO1 1
#define LDMIA  (unsigned32) 0xE89E3FFF
                             /* op-code for LDMIA instruction */
#define SC_DATABUS 33        /* length of the scan chain */

int            ValPtr;
unsigned8      Values[MACRO_ARGUMENT_AREA_SIZE];
                             /* const def in macros.h */

/*All parameters are fixed.*/
  InitParams;                /* Reset ValPtr */
  NREnterParamU32(LDMIA);    /* Place param1 (TDIbits1) in Values array */
  NREnterParamU8(0);         /* Place param2 (TDIbits2) in Values array */
  NREnterParamU8(1);         /* Place param3 (TDIrev) in Values array */
  NREnterParamU8(SC_DATABUS);/* Place param4 (len) in Values array */
  NREnterParamU8(0);         /* Place param5 (WRoffset) in Values array */
  NREnterParamU32(0);        /* Place param6 (WRmask1) in Values array */
  NREnterParamU8(0);         /* Place param7 (WRmask2) in Values array */
  NREnterParamU8(1);         /* Place param8 (nclks) in Values array */

/* ‘line’ can now be added */

TAPCheck(TAPOp_DefineMacro(connectId, MACRO1,"ARMTAP_AccessDR_W:12345678",
                             1, Values, ValPtr));
```

where `12345678` means that parameters 1 to 8 are fixed.

To run the macro

```
int  lnerr,lperr,     /* Variables for error position detecting */
     resultvalues,    /* In this example, no data is returned, but */
     resultsize = 0;  /* variables need to be defined for correct operation */

InitParams;           /* Reset ValPtr */

TAPCheck(TAPOp_RunMacro(connectId, MACRO1, Values, ValPtr, &lnerr,
                          &lperr, &resultvalues, &resultsize, 1, 1));
```

Since all the parameters are fixed, there is no need to load any parameters for the `RunMacro`. It is still necessary, however, to use `InitParams` to indicate this.

5.3.4 Example 2

To demonstrate sending one of the parameters at runtime, a macro line is entered to accept the least-significant 32 TDIbits at runtime for three instructions. The purpose of this macro is to send the following instructions to the Data Register. The following are entered at runtime:

- LDMIA instruction
- two NOPs

To define the macro

```
#define MACRO2     2
#define LDMIA (unsigned32) 0xE89E3FFF/* op-code for LDMIA instr */
#define SC_DATBUS   (unsigned8) 33
int          ValPtr;
unsigned8    Values[MACRO_ARGUMENT_AREA_SIZE];

/* Send 1 data word out.*/
/*Parameters 2 to 5 are fixed, parameter 1 sent at run-time */
InitParams;               /* Reset ValPtr */
NREnterParamU8(0);        /* Place param2 (TDIbits2) in Values array */
NREnterParamU8(1);        /* Place param3 (TDIrev) in Values array */
NREnterParamU8(SC_DATBUS);/* Place param4 (len) in Values array */
NREnterParamU8(0);        /* Place param5 (WRoffset) in Values array */
NREnterParamU32(0);       /* Place param6 (WRmask1) in Values array */
NREnterParamU8(0);        /* Place param7 (WRmask2) in Values array */
NREnterParamU8(1);        /* Place param8 (nclks) in Values array */

/* ‘line’ can now be added */

TAPCheck(TAPOp_DefineMacro(connectId, MACRO2,
             "ARMTAP_AccessDR_W:2345678", 3,Values, ValPtr));
```

To run the macro

```
int   lnerr,lperr,   /* Variables for error position detecting */
      resultvalues,  /* In this example, no data is returned, but */
      resultsize = 0;/* variables need to be defined for correct operation */
InitParams;          /* Reset ValPtr */
NREnterParamU32(LDMIA);/* Place param1 (TDIbits1) for first iteration */
NREnterParamU32(NOP);/* Place param1 (TDIbits1) for second iteration */
NREnterParamU32(NOP);/* Place param1 (TDIbits1) for third iteration */

TAPCheck(TAPOp_RunMacro(connectId, MACRO2, Values, ValPtr, &lnerr,
                           &resultvalues, &resultsize, 1, 1));
```

5.3.5 Example 3

The following example shows how three ‘lines’ are added to a macro. It uses a combination of fixed and variable parameters and also uses multiple iterations of the instruction in a single line. This macro sends the following instructions and data to the Data Register.

The following are all entered at runtime:

- LDMIA instruction
- two NOPs
- 14 x 32-bit data words
- two NOPs
- STMIA instruction

The following is fixed at define time:

- NOP with breakpoint bit set

To define the macro

```
#define MACRO3     3                   /* macro number 3 */
#define NOP  (unsigned32) 0xE1A00000   /* A no-op for ARM7TDMI */
#define SC_DATBUS (unsigned8) 33

int          ValPtr;
unsigned8    Values[MACRO_ARGUMENT_AREA_SIZE];

void define_send14_macro(void)
{
/* Parameters 2 to 8 are fixed, parameter 1 sent at run-time. */
   InitParams;/* Reset ValPtr */
   NREnterParamU8(0);          /* Place param2 (TDIbits2) in Values array */
   NREnterParamU8(1);          /* Place param3 (TDIrev) in Values array */
   NREnterParamU8(SC_DATABUS); /* Place param4 (len) in Values array */
   NREnterParamU8(0);          /* Place param5 (WRoffset) in Values array */
   NREnterParamU32(0);         /* Place param6 (WRmask1) in Values array */
   NREnterParamU8(0);          /* Place param7 (WRmask2) in Values array */
   NREnterParamU8(1);          /* Place param8 (nclks) in Values array */

/* ‘line’ 1 can now be added - but it will be entered 19 times */
   TAPCheck(TAPOp_DefineMacro(connectId,MACRO3,
                               "ARMTAP_AccessDR_W:2345678",19, Values,
                                ValPtr));
/* send a NOP with the breakpoint (b32) bit set.*/
/* All 8 parameters are fixed.*/
   InitParams;                 /* Reset ValPtr */
   NREnterParamU32(NOP);       /* Place param1 (TDIbits1) in Values array */
   NREnterParamU8(1);          /* Place param2 (TDIbits2) in Values array */
   NREnterParamU8(1);          /* Place param3 (TDIrev) in Values array */
   NREnterParamU8(SC_DATABUS); /* Place param4 (len) in Values array */
   NREnterParamU8(0);          /* Place param5 (WRoffset) in Values array */
   NREnterParamU32(0);         /* Place param6 (WRmask1) in Values array */
   NREnterParamU8(0);          /* Place param7 (WRmask2) in Values array */
   NREnterParamU8(1);          /* Place param8 (nclks) in Values array */
/* ‘line’ 2 can now be added */
   TAPCheck(TAPOp_DefineMacro(connectId,MACRO3,
                            "ARMTAP_AccessDR_W:12345678",1,Values,
                            ValPtr));
/* Send 1 data word out.*/
/*Parameters 2 to 8 are fixed, parameter 1 sent at run-time */
   InitParams;                 /* Reset ValPtr */
   NREnterParamU8(0);          /* Place param2 (TDIbits2) in Values array */
   NREnterParamU8(1);          /* Place param3 (TDIrev) in Values array */
   NREnterParamU8(SC_DATABUS); /* Place param4 (len) in Values array */
   NREnterParamU8(0);          /* Place param5 (WRoffset) in Values array */
   NREnterParamU32(0);         /* Place param6 (WRmask1) in Values array */
   NREnterParamU8(0);          /* Place param7 (WRmask2) in Values array */
   NREnterParamU8(1);          /* Place param8 (nclks) in Values array */

/* ‘line’ 3 can now be added */
   TAPCheck(TAPOp_DefineMacro(connectId,MACRO3,
                              "ARMTAP_AccessDR_W:2345678",1,Values,
                              ValPtr));
/* check that macro has been entered OK */
   TAPCheck(TAPOp_DisplayMacro(connectId, MACRO3));
}
```

To run the macro

```
#define LDMIA  (unsigned32)   0xE89E3FFF
#define STMIA  (unsigned32)   0xE8AE3FFF
#define NOP    (unsigned32)   0xE1A00000

TAPOp_Error run_send14_macro(unsigned32 *data, int *lnerr, int *lperr)
{
  intj,                /* loop counter */
     lnerr, lperr,     /* Variables for error position detecting*/
     resultvalues,     /* In this example, no data is returned, but*/
     resultsize = 0;   /* variables need to be defined for correct operation */

  InitParams;          /* reset ValPtr */
/* Send parameters for ‘line’ 1, 19 unsigned32 words required */
  NREnterParamU32(LDMIA);                       /* 1 */
  NREnterParamU32(NOP);                         /* 2 */
  NREnterParamU32(NOP);                         /* 3 */
  for (j=0;j<14;j++) {
       NREnterParamU32(data[j]);/* 4 to 17 */
}
  NREnterParamU32(NOP);                         /* 18 */
  NREnterParamU32(NOP);                         /* 19 */
/* ‘line’ 2 is now ‘skipped’ as it requires no further parameters */
/* Send parameter for ‘line’ 3, 1 unsigned32 word required */
  NREnterParamU32(STMIA);

  TAPCheck(TAPOp_RunMacro(connectId, MACRO3, Values, ValPtr, lnerr, lperr,
       &resultvalues,&resultsize, 1, 1));
  return t_err;
}
```

5.4 TAPOp Calls Listed by Function

This section lists TAPOp calls according to their general function. The categories are:

- TAP controller/scan chain access procedures
- Macro usage procedures
- Connection control procedures
- User I/O procedures
- Data read/write procedures
- Debugging procedures

Each procedure is listed in ***TAPOp Procedure Call Alphabetic Reference*** on page 5-21.

5.4.1 TAP controller/scan chain access

Type of TAP operation	Generic TAP operations (use Run-Test/Idle as idle)	ARM-specific operations (use Select-DR-Scan as idle)
IR access (write only)	`TAPOp_AccessIR`	`ARMTAP_AccessIR` `ARMTAP_AccessIR_1Clk`

DR (scan chain) write	`TAPOp_AccessDR_W`	`ARMTAP_AccessDR_W`
		`ARMTAP_AccessDR_NoClk_W`
		`ARMTAP_AccessDR_1Clk_W`

DR (scan chain) read/write	`TAPOp_AccessDR_RW`	`ARMTAP_AccessDR_RW` `ARMTAP_AccessDR_RW_And_Test`
TAP controller manual control	`TAPOp_AnySequence_W` `TAPOp_AnySequence_RW`	
ARM clock control		`ARMTAP_ClockARM`

Table 5-1: **TAP controller/scan chain access**

5.4.2 Data read/write

Data type	Function
Private data	`TAPOp_ReadPrivateFlags` `TAPOp_WritePrivateFlags`
Common data	`TAPOp_ReadCommonData` `TAPOp_WriteCommonData`

Table 5-2: Data read/write

5.4.3 Connection control

	RPC connection control	TAPOp connection control
Opening Connections	`rpc_initialise` `GetServerName`	`TAPOp_OpenConnection` `TAPOp_GetDriverDetails`
Closing Connections	`rpc_finalise`	`TAPOp_CloseConnection`
Connection Heartbeat		`TAPOp_PingServer`

Table 5-3: Connection control

5.4.4 Debugging

Debug facility	Function
Displaying macros	`TAPOp_DisplayMacro`
RPC logging	`TAPOp_SetLogging`

Table 5-4: Debugging

5.4.5 Macro usage

	Normal macro usage	Advanced macro usage
Creating macros	`TAPOp_DefineMacro`	
Destroying macros [1]	`TAPOp_DeleteMacro`	
	`TAPOp_DeleteAllMacros`	
Displaying macros (for debug)	`TAPOp_DisplayMacro`	
Running macros	`TAPOp_RunMacro`	`TAPOp_RunBufferedMacro` `TAPOp_FillMacroBuffer`
Synchronized stop/start macros		`TAPOp_SetControlMacros`

Table 5-5: Macro usage

[1] *TAPOp_CloseConnection calls TAPOp_DeleteAllMacros automatically.*

5.4.6 User I/O

Signal	Function
Output bits:	`TAPOp_WriteMICEUser1` `TAPOp_WriteMICEUser2`
System reset:	`TAPOp_SetSysResetSignal`

Table 5-6: User I/O

5.5 TAPOp Procedure Call Alphabetic Reference

This section lists all available TAPOp procedure calls. Each argument is shown on a new line for clarity, and the **`bold`** text describes the type of data required for each argument. The prototypes for these procedure calls are held in the file named at the top of each procedure call listing—for example, `(armtapop.h)`.

The TAPOp procedure calls described are:

TAPOp_SetLogging on page 5-73.

TAPOp_SetSysResetSignal on page 5-74.

TAPOp_WriteCommonData on page 5-75.

TAPOp_WriteMICEUser1 on page 5-76.

TAPOp_WriteMICEUser2 on page 5-77.

TAPOp_WritePrivateFlags on page 5-78.

5.5.1 ARMTAP_AccessDR_1Clk_W

Writes data to a TAP data register (scan chain) and performs one DCLK. Data coming out of TDO is discarded.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessDR_1Clk_W(
        unsigned8   connectId,
        ScanData40  *TDIbits,
        unsigned8   TDIrev,
        unsigned8   len,
        unsigned8   WRoffset,
        ScanData40  *WRmask,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data read is from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made. You should try again later.

7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.

9	`TAPOp_BadParameter`	Failed because of one of the following:
		`TDIbits = NULL`
		`len = 0`
		`Wroffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.2 ARMTAP_AccessDR_NoClk_W

Writes data to a TAP data register (scan chain). No DCLKs are performed (in other words, does not go through Run-Test/Idle). Data coming out of TDO is discarded.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessDR_NoClk_W(
        unsigned8   connectId,
        ScanData40  *TDIbits,
        unsigned8   TDIrev,
        unsigned8   len,
        unsigned8   WRoffset,
        ScanData40  *WRmask,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data read is from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan

9	`TAPOp_BadParameter`	Failed because of one of the following:
		`TDIbits = NULL`
		`len = 0`
		`Wroffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.3 ARMTAP_AccessDR_RW

Reads and writes data from a TAP data register. The TAP does *not* go through Run-Test/Idle if `nclks=0`, otherwise it does, and (`nclks-1`) transitions from Run-Test/Idle are made to itself before returning to Select-DR-Scan; this has the effect of producing `nclks` DCLKs.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessDR_RW(
       unsigned8   connectId,
       ScanData40  *TDIbits,
       unsigned8   TDIrev,
       ScanData40  *TDObits,
       unsigned8   TDOrev,
       unsigned8   len,
       unsigned8   WRoffset,
       ScanData40  *WRmask,
       unsigned8   RDoffset,
       unsigned8   nclks,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`TDOrev`	If 0, you read TDO via the Multi-ICE Data Register; otherwise, read TDO via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`RDoffset`	Offset of the selected data register (scan chain) to start reading data from.

	`nclks`	Number of ARM DClks which are to be produced. This is the number of transitions out of Run-Test/Idle which are made. If 0, Run-Test/Idle is avoided altogether. **Note:** `0 <= n < 32`
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.
Output	`TDObits`	TDO data is put here. This must not be NULL.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan

9	`TAPOp_BadParameter`	Failed because of one of the following:
		`TDIbits = NULL`
		`len = 0`
		`Wroffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.
2. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.4 ARMTAP_AccessDR_RW_And_Test

Writes and reads data from a TAP data register (scan chain). TDO is marked with `maskBits` and compared with `resBits`. If a match is found, `TAPOp_NoError` is returned, but if no match is found, the write and read is performed `nTries` times before returning `TAPOp_MaskAndTestFailed`.The TAP does *not* go through Run-Test/Idle, so if an ARM Data Bus is selected, no DCLK happens.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessDR_RW_And_Test(
        unsigned8    connectId,
        ScanData40   *TDIbits,
        unsigned8    TDIrev,
        unsigned8    TDOrev,
        unsigned8    len,
        ScanData40   *maskBits,
        ScanData40   *resBits,
        unsigned8    WRoffset,
        ScanData40   *WRmask,
        unsigned8    RDoffset,
        unsigned32   nTries,
        unsigned8    deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`TDOrev`	If 0, you read TDO via the Multi-ICE Data Register; otherwise, read TDO via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`maskBits`	Mask bits used to test TDO.
	`resBits`	Result bits used to test TDO.
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.

`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
`RDoffset`	Offset of the selected data register (scan chain) where you start reading data.
`nTries`	Number of unsuccessful attempts before returning an error.
`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.

9	`TAPOp_BadParameter`	Failed because of one of the following:
		`TDIbits = NULL`
		`maskBits  = NULL`
		`resBits = NULL`
		`nTries = 0`
		`len = 0`
		`WRoffset >= len`
		`RDoffset  >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
25	`TAPOp_MaskAndTestFailed`	After `nTries`, mask and test still did not match `resBits`.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.5 ARMTAP_AccessDR_W

Writes data to a TAP data register (could be the data bus scan chain on an ARM) and performs one or more DCLKs. Data coming out of TDO is discarded.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessDR_W(
       unsigned8   connectId,
       ScanData40  *TDIbits,
       unsigned8   TDIrev,
       unsigned8   len,
       unsigned8   WRoffset,
       ScanData40  *WRmask,
       unsigned8   nclks,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection.`
	`TDIbits`	TDI is data read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`nclks`	The number of DCLKs to perform after the write (how times to enter Run-Test/Idle). **Note:** `0 < n <32`.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.

9	`TAPOp_BadParameter`	Failed because of one of the following:
		`TDIbits = NULL`
		`len = 0`
		`Wroffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.6 ARMTAP_AccessIR

Writes data to the TAP instruction register. Data coming out of TDO is discarded.

The length of the Instruction Register is already known by the Multi-ICE application because this is part of the configuration data for each processor.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_AccessIR(
       unsigned8   connectId,
       unsigned16  TDIbits,
       unsigned8   TDIrev,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP does NOT go through Run-Test/Idle, so if the ARM Data Bus scan chain is selected no DCLK happens. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.7 ARMTAP_AccessIR_1Clk

Writes data to the TAP instruction register, and performs one DCLK (goes through the Run-Test/Idle state). Data coming out of TDO is discarded.

The length of the Instruction Register is already known by the Multi-ICE application as this is part of the configuration data for each processor.

`(armtapop.h)`

```
extern TAPOp_Error ARMTAP_AccessIR_1Clk(
       unsigned8   connectId,
       unsigned16  TDIbits,
       unsigned8   TDIrev,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.8 ARMTAP_ClockARM

Performs a number of ARM DCLKs. This function writes INTEST into the instruction register, as it has to go through either the DR or IR states to get to Run-Test/Idle.

In general, it is always more efficient to call one of the `ARMTAP_AccessDR` functions which combines DCLKs with the DR access, but sometimes this is not possible.

```
(armtapop.h)
extern TAPOp_Error ARMTAP_ClockARM(
        unsigned8   connectId,
        unsigned8   nclocks,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`nclocks`	The number of DCLKs to perform. **Note:** `0 <= nclocks <= 31`
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Select-DR-Scan.
9	`TAPOp_BadParameter`	Failed because `nclks >= 32`.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. The TAP Controller must be in Select-DR state before the function is used, and is left in this state after the function has been performed.

5.5.9 GetServerName

Returns the name of the selected RPC Server. This procedure needs to be provided by the user. The file `rpcclient.c` calls this procedure to find out the location of the Multi-ICE Server.

```
(tapop.h)
int GetServerName(
       char        **ServerName
);
```

Arguments

Output	`ServerName`	The name/location of the RPC Server. For example: `"pc25"`

Returns

0	No error.
Non-zero	Failure (user-supplied values).

5.5.10 rpc_finalise

Finalizes the RPC transport layer. You call this to close the TCP/IP connection. See also ***rpc_initialise*** on page 5-39.

```
(tapop.h)
void rpc_finalise(void);
```

Arguments

None.

Returns

None.

5.5.11 rpc_initialise

Initializes the RPC transport layer. You call this before using any other RPC calls.See also ***rpc_finalise*** on page 5-38.

```
(tapop.h)
int rpc_initialise(void);
```

Arguments

None.

Returns

0	No error.
Non-zero	Failed to open RPC connection.

5.5.12 TAPOp_AccessDR_RW

Reads and writes data from a TAP data register (scan chain).

```
(tapop.h)
extern TAPOp_Error TAPOp_AccessDR_RW(
      unsigned8   connectId,
      ScanData40  *TDIbits,
      unsigned8   TDIrev,
      ScanData40  *TDObits,
      unsigned8   TDOrev,
      unsigned8   len,
      unsigned8   WRoffset,
      ScanData40  *WRmask,
      unsigned8   RDoffset,
      unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	The TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`TDOrev`	If 0, you read TDO via the Multi-ICE Data Register; otherwise, read TDO via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`RDoffset`	Offset of the selected data register (scan chain) where you start reading data.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Output	`TDObits`	TDO data is put here. This must not be NULL.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Run-Test/Idle.

9	`TAPOp_BadParameter`	Failed for one of the following reasons:
		`TDIbit = NULL`
		`TDObit = NULL`
		`len = 0`
		`RDoffset >= len`
		`WRoffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. The TAP Controller must be in Run-Test/Idle state before the function is used, and is left in this state after the function has been performed.
2. If a connection to this TAP controller has been selected already then this operation happens automatically. If not, this call attempts to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.13 TAPOp_AccessDR_W

Writes data to a TAP data register (scan chain). Data coming out of TDO is discarded.

```
(tapop.h)
extern TAPOp_Error TAPOp_AccessDR_W(
        unsigned8   connectId,
        ScanData40  *TDIbits,
        unsigned8   TDIrev,
        unsigned8   len,
        unsigned8   WRoffset,
        ScanData40  *WRmask,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`len`	Length of the selected data register (scan chain).
	`WRoffset`	Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated (whatever comes out of the TDO pin is put back into TDI during the scan). To write less than 40 bits, use `WRmask`.
	`WRmask`	A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan. 1 Bit is written. 0 Bit is recirculated, so that whatever comes out of the TDO pin is put back into TDI during the scan. To write all 40 bits, `WRmask` can be set to NULL.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Run-Test/Idle.

9	`TAPOp_BadParameter`	Failed for one of the following reasons:
		`TDIbit = NULL`
		`WRoffset >= len`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. The TAP Controller must be in Run-Test/Idle state before this function is used, and is left in this state after the function has been performed.
2. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call attempts to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.14 TAPOp_AccessIR

Writes data to the TAP instruction register. Data coming out of TDO is discarded.

The length of the Instruction register is already known by the Multi-ICE Application as this is part of the configuration data for each processor.

```
(tapop.h)          extern TAPOp_Error TAPOp_AccessIR(
       unsigned8   connectId,
       unsigned16  TDIbits,
       unsigned8   TDIrev,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TDIrev`	If 0, you present TDI via the Multi-ICE Data Register; otherwise, present TDI via the Reversed Data Register.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	The TAP controller is not in Run-Test/Idle.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. The TAP Controller must be in Run-Test/Idle state before the function is used, and is left in this state after the function has been performed.
2. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.15 TAPOp_AnySequence_RW

Perform a sequence of TCKs, with explicitly specified TMS and TDI inputs. TDO values are read out.

```
(tapop.h)
extern TAPOp_Error TAPOp_AnySequence_RW(
        unsigned8   connectId,
        unsigned8   numTCKs,
        unsigned32  *TDIbits,
        unsigned32  *TMSbits,
        unsigned32  *TDObits,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`numTCKs`	Number of TCKs to perform (max 255).
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TMSbits`	TMS data is read from here. This must not be NULL.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.
Output	`TDObits`	TDO data is written to here. This must not be NULL.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	Deselected when not in Select-DR-Scan or Run-Test/Idle.

9	`TAPOp_BadParameter`	Failed for one of the following reasons:
		`TDIbits = NULL`
		`TDObits = NULL`
		`TMSbits = NULL`
		`numTCKs = 0`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
32	`TAPOp_AnySeqUsedBadPath`	An Exit2-IR/DR -> Shift-IR/DR transition was performed.

33	`TAPOp_AnySeqWrongIRLength`	The wrong number of Shift-IR TCKs was detected.

This interface is a “virtual TAP controller” interface—in other words, if there are actually several TAP controllers connected in series, the Server will deal with having to insert extra bits into the IR and DR scan chains transparently to the caller of this function. For this to work, however, some restrictions are imposed on what sequences can be performed. These restrictions are outlined below.

Note *If a connection to this TAP controller has already been selected, this operation happens automatically. If not, this call will attempt to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and* `TAPOp_UnableToSelect` *is returned. The caller should try again later.*

Post conditions

When the connection is deselected, the TAP controller must be left in one of Select-DR-Scan or Run-Test/Idle. If this is not done, an error will result.

Restrictions

1. When shifting data into an IR or DR register, all the shifts (performed when the TAP Controller start state is Shift-IR/DR) must occur together—in other words, after performing some such shifts, Shift-IR/DR should not be left and then re-entered without going through Capture-IR/DR. If the user does this, the Server will return an error indicating that the transition Exit2-IR/DR -> Shift-IR/DR was performed. The client code should be recoded to avoid this case.
2. The Multi-ICE Server knows the length of the IR register. If the number of shifts into this register is not the same as the length according to the Server, the Server will ignore the data shifted into IR, and will instead put in a BYPASS instruction, causing an error to be returned. This ensures that other TAP controllers cannot be accidentally put into strange states by a bug in this TAP controller’s client.
3. As a consequence of IR length checking, the data to be shifted into the IR must be in a single `TAPOp_AnySequence_RW` call—it must not be split over two calls. This is necessary so that the call can ensure that the correct number of IR shifts are performed before processing any of them.
4. The number of shifts performed in Shift-DR must be the same as the length of the scan chain. If it is not, then in a multi-processor system the operation is undefined due to the insertion of extra `TCKS` by the Server to bypass other TAP controllers.
5. If used in a macro, all 256 bits must be passed. Use fixed / variable U32s; `numTCKs` determine how many are actually used. For a single `AnySequence_RW` call, a pointer to an array is passed.

5.5.16 TAPOp_AnySequence_W

Performs a sequence of TCK's, with explicitly specified TMS and TDI inputs. TDO is ignored.

```
(tapop.h)
extern TAPOp_Error TAPOp_AnySequence_W(
        unsigned8   connectId,
        unsigned8   numTCKs,
        unsigned32  *TDIbits,
        unsigned32  *TMSbits,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`numTCKs`	Number of TCKs to perform (maximum 255).
	`TDIbits`	TDI data is read from here. This must not be NULL.
	`TMSbits`	TMS data is read from here. This must not be NULL.
	`deselect`	0 indicates that the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	Deselected when not in Select-DR-Scan or Run-Test/Idle.

9	`TAPOp_BadParameter`	Failed for one of the following reasons:
		`TDIbits = NULL`
		`TMSbits = NULL`
		`numTCKs = 0`

19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
32	`TAPOp_AnySeqUsedBadPath`	An Exit2-IR/DR -> Shift-IR/DR transition was performed.
33	`TAPOp_AnySeqWrongIRLength`	The wrong number of Shift-IR TCKs was detected.

TAPOp Procedure Calls

This interface is a “virtual TAP controller” interface—in other words, if there are actually several TAP controllers connected in series, the Server will deal with having to insert extra bits into the IR and DR scan chains transparently to the caller of this function. For this to work, however, some restrictions are imposed on what sequences can be performed. These restrictions are outlined below.

Note *If a connection to this TAP controller has already been selected, this operation happens automatically. If not, this call will attempt to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and* `TAPOp_UnableToSelect` *is returned. The caller should try again later.*

Post conditions

When the connection is deselected, the TAP controller must be left in one of Select-DR-Scan or Run-Test/Idle. If this is not done, an error will result.

Restrictions

1. When shifting data into an IR or DR register, all the shifts (performed when the TAP Controller start state is Shift-IR/DR) must occur together—in other words, after performing some such shifts, Shift-IR/DR should not be left and then re-entered without going through Capture-IR/DR. If the user does this, the Server will return an error indicating that the transition Exit2-IR/DR -> Shift-IR/DR was performed. The client code should be recoded to avoid this case.
2. The Multi-ICE Server knows the length of the IR register. If the number of shifts into this register is not the same as the length according to the Server, the Server will ignore the data shifted into IR, and will instead put in a BYPASS instruction, causing an error to be returned. This ensures that other TAP controllers cannot be accidentally put into strange states by a bug in this TAP controller’s client.
3. As a consequence of IR length checking, the data to be shifted into the IR must be in a single `TAPOp_AnySequence_W` call—it must not be split over two calls. This is necessary so that the call can ensure that the correct number of IR shifts are performed before processing any of them.
4. The number of shifts performed in Shift-DR must be the same as the length of the scan chain. If it is not, then in a multi-processor system the operation is undefined due to the insertion of extra `TCKS` by the Server to bypass other TAP controllers.
5. If used in a macro, all 256 bits must be passed. Use fixed / variable U32s; `numTCKs` determine how many are actually used. For a single `AnySequence_W` call, a pointer to an array is passed.

5.5.17 TAPOp_CloseConnection

At the end of a debug session, this function kills a connection to a TAP controller for the TAPOp client. After this call, no further TAPOp functions may be called with this connection ID.

This should not be confused with deselecting a connection, which occurs when a TAPOp function is called and the deselect parameter is non-zero.

See also ***TAPOp_OpenConnection*** on page 5-58.

```
(tapop.h)
extern TAPOp_Error TAPOp_CloseConnection(
        unsigned8 connectId
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.18 TAPOp_DefineMacro

Adds a ‘line’ (one TAPOp / ARMTAP function call) to a macro. Some parameters can be passed at definition time, and these are fixed so that they need not be passed in at runtime.

This function should be called both for the first ‘line’ of a macro and for subsequent ‘lines’. You can have one line executed a number of times; this allows the Server to optimize some operations providing better performance. When this facility is used, only one block of fixed parameter data is used each time the line is executed. However, each time the line is executed, a new set of variable data is used and new results are output.

See also the examples in ***Using TAPOp Macros*** on page 5-11.

```
(tapmacro.h)
extern TAPOp_Error TAPOp_DefineMacro(
      unsigned8   connectId,
      unsigned8   macroNo,
      char        *callDetails,
      unsigned8   nTimes,
      void        *fixedParamValues,
      int         paramSize
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.	
	`macroNo`	The number of the macro to run (local to this connection).	
	`callDetails`	*`fn-name`*	is used if there are no fixed parameters for this invocation of *`fn-name`*
		`fn-name:list`	is used if there are fixed parameters. *`fn-name`* should be the name of one of the exported TAPOp or ARMTAPOp functions—for example, `ARMTAP_AccessDR_W`
		The list of fixed parameters consists of a series of single characters—for example, “1256”, which must be in numerically increasing order. Each digit refers to the corresponding (starting at 1) structure element in the corresponding `MAC_xxx` structure defined in `macstruct.h`. Any parameters listed are fixed, and the fixed values are passed in `fixedParamValues`. **Note:** *Parameters 10,11,12 and upwards are specified with ‘A’, ‘B’, ‘C’ and so on. This is not hex because it does not stop at ‘F’. Currently, however, there are no definitions which need more than 15 parameters, so it can be regarded as hexadecimal. There is no parameter ‘0’.*	

`nTimes`	The number of times the line of this macro should be executed. Each time it is executed, the same fixed parameters are used, but a new set of variable parameters is used, and results are added to the result block.
`fixedParamValues`	A block of data holding the fixed parameters for this invocation of this macro.
`paramSize`	The size of the block of fixed parameter data. This is needed so it can be easily sent over RPC.

Returns

0	`TAPOp_NoError`	No error
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
8	`TAPOp_InBadTAPState`	TAP controller is not in Run-Test/Idle.
9	`TAPOp_BadParameter`	Unrecognized *`fn-name`* in `callDetails`.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
21	`TAPOp_TooManyMacros`	Maximum number of macros exceeded. The maximum is 1000.
22	`TAPOp_TooManyMacroLines`	Too many lines in the macro. The maximum is 1000.
23	`TAPOp_BadFixedParamNo`	Bad Parameter Number specified.
34	`TAPOp_UnknownProcedureName`	Unknown procedure name in `DefineMacro` call.
35	`TAPOp_CantUseProcInMacro`	Procedure specified in `DefineMacro` call cannot be run in a macro.

5.5.19 TAPOp_DeleteAllMacros

Deletes all macros for a particular connection.

```
(tapmacro.h)
extern TAPOp_Error TAPOp_DeleteAllMacros(
      unsigned8   connectId
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.

Returns

0	`TAPOp_NoError`	No error.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.20 TAPOp_DeleteMacro

Deletes a currently defined macro for a particular connection.

```
(tapmacro.h)
extern TAPOp_Error TAPOp_DeleteMacro(
        unsigned8   connectId,
        unsigned8   macroNo
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`macroNo`	The number of the macro to delete.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
20	`TAPOp_UndefinedMacro`	`macroNo` is not defined for that `connectId`.

5.5.21 TAPOp_DisplayMacro

Sends a request to the Multi-ICE Server to display the lines of a macro. This is intended solely for use when debugging TAPOp clients.

```
(tapmacro.h)
extern TAPOp_Error TAPOp_DisplayMacro(
        unsigned8   connectId,
        unsigned8   macroNo
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`macroNo`	The number of the macro to display (local to this connection).

Returns

0	`TAPOp_NoError`	No error.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Note

If a macro has not been defined, this is reported on the Server console. No error is returned.

Output format

The macro information is displayed on the Multi-ICE Server in the following format:

```
line_no : TAPOp/ARMTAP_instr, fixed_params, macro_line_executions
```

For example:

```
    -------------------------- Macro no. 1 -----------------------
    0: ARMTAP_AccessDR_W , plist=0x1F , nTimes=1
    1: ARMTAP_AccessDR_W , plist=0x1E , nTimes=14
```

In line 0:

```
    plist=0x1F = bin 0001 1111
```

showing that parameters 1,2,3,4, and 5 have been fixed.

In line 1:

```
    plist=0x1E = bin 0001 1110
```

showing parameters 2,3,4, and 5 have been fixed.

5.5.22 TAPOp_FillMacroBuffer

Loads a buffer in the Server with variable parameters, before running a macro from the buffer. See ***TAPOp_RunBufferedMacro*** on page 5-66 for an explanation of how to use this function.

```
(tapmacro.h)
TAPOp_Error TAPOp_FillMacroBuffer(
        unsigned8   connectId,
        unsigned8   bufferNo,
        void        *variableParamValues,
        int         paramSize
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection.`
	`bufferNo`	The number of the buffer to load with the variable data block. This must be 0 or 1.
	`variableParamValues`	A block of data holding the variable parameters to be written to the buffer.
	`paramSize`	The size of the block of variable parameter data. This is needed so the data can be sent easily over RPC.

Returns

0	`TAPOp_NoError`	No error.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
9	`TAPOp_BadParameter`	The `bufferNo` is not 0 or 1.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.23 TAPOp_GetDriverDetails

Before a debugger opens a connection to a device on the Multi-ICE Server, it must find out from the Multi-ICE Server which devices are available. `TAPOp_GetDriverDetails` performs this function by getting a list of drivers (devices) from the Server. You use this information to find out which drivers (and therefore which TAP controllers) are available.

```
(tapop.h)
extern TAPOp_Error TAPOp_GetDriverDetails(
       MultiICE_DriverDetails    *DriverDets,
       unsigned32                *NumDrivers,
       unsigned32                *VersionNo
);
```

Arguments

In/Out	`DriverDets`	The structures where you put the driver details. See the `MultiICE_DriverDetails` structure definition for details. This can be found in `mice.h` .
	`NumDrivers`	The number of driver structures pointed to by `DriverDets`. Before you call this procedure, set `NumDrivers` to the maximum number of drivers you have allocated space for. The procedure returns the actual number in `DriverDets`.
Output	`VersionNo`	Version number of the Multi-ICE application.

Returns

0	`TAPOp_NoError`	No error.
4	`TAPOp_NotInitialised`	The Multi-ICE Server is not initialized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.24 TAPOp_LogString

This function inserts a user-supplied string into the log file.

```
(tapop.h)
extern TAPOp_Error TAPOp_LogString(
       unsigned8  connectId,
       char       *message
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`message`	The string to output to the log file. The string length is limited to 255 characters.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Note

This call does not require a selected connection to a TAP controller, as it is just providing debug output.

5.5.25 TAPOp_OpenConnection

At the beginning of a debug session, this function creates a connection to a TAP controller for the TAPOp client, and returns a connection ID to be used when calling other TAPOp functions.

This function should not be confused with selecting a connection that occurs when a TAPOp function is called and a different connection was previously selected.

The set of scan chains connected to the TAP specified is listed, and the implementation checks that two connections to the same TAP do not conflict.

See also ***TAPOp_CloseConnection*** on page 5-49.

```
(tapop.h)
extern TAPOp_Error TAPOp_OpenConnection(
       unsigned8       TAPPos,
       unsigned8       *connectId,
       unsigned8       numScanChains,
       unsigned8       *scanChains,
       unsigned8       startState,
       unsigned8       IRlen,
       unsigned32      intest,
       unsigned8       SCSRlen,
       unsigned32      scan_n,
       unsigned8       allowAutoDisconnect,
       char            *debuggerName,
       char            *driverName
);
```

Arguments

Input	`TAPPos`	The position in the scan chain of the TAP controller to connect to. Position 0 is closest to TDI.
	`numScanChains`	The number of scan chains claimed by this connection.
	`scanChains`	An array containing the numbers of the scan chains claimed by this connection.
	`startState`	The startup state of TAP controller for this connection: 1 Start the TAP controller(s) in Select-DR-Scan. 2 Start the TAP controller(s) in Run-Test/Idle. Other values are invalid.
	`IRlen`	This is the length of the Instruction Register of the TAP being connected. This is compared with the length of the device stored in `IRlength.arm`.

	`intest`	INTEST instruction bit pattern for the TAP that this connection is talking to. If it is 0, the TAP does not support multiple scan chains.
	`SCSRlen`	This is the length of the scan chain select register for the TAP being connected.
	`scan_n`	SCAN_N instruction bit pattern for the TAP that this connection is talking to. It is only valid if `SCSRlen != 0`.
	`allowAutoDisconnect`	This flag is set if the server is allowed to disconnect this client due to another client attempting to connect. If this client implements the standard heartbeat mechanism (automatic for Win32 clients), this flag should be set. If this is a non-Win32 client and there is no special heartbeat set up, this flag should be 0.
	`debuggerName`	A null-terminated string that gives the identity of the debugger.
	`driverName`	A null-terminated string which contains the driver name from `GetDriverDetails` for this connection.
Output	`connectId`	The connection ID to be used when calling other TAPOp functions.

Returns

0	`TAPOp_NoError`	No error.
3	`TAPOp_TAPNotPresent`	Bad TAP controller position.
4	`TAPOp_NotInitialised`	The Multi-ICE Server is not initialized.
5	`TAPOp_TooManyConnections`	There are no free connections.
9	`TAPOp_BadParameter`	Bad parameter value passed.
13	`TAPOp_ScanChainAlreadyClaimed`	One of the required scan chains has already been claimed by another connection.
18	`TAPOp_ParameterConflicts`	A parameter conflicts with the configuration data.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.26 TAPOp_PingServer

This provides a heartbeat function. It is used to poll the Server so that the Server knows the client is still connected during periods of inactivity. Although this function is part of the `TAPOp` public interface, it is not normally used, as `rpcclient.c` normally calls this function once per second when a connection is active.

```
(tapop.h)
extern TAPOp_Error TAPOp_PingServer(
      unsigned8      connectId
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.

Returns

0	`TAPOp_NoError`	No error.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

5.5.27 TAPOp_ReadCommonData

Reads the data block which is common to all the debuggers connected to Multi-ICE.

```
(tapshare.h)
extern TAPOp_Error TAPOp_ReadCommonData(
      unsigned8   connectId,
      unsigned32  *commonBlk,
      unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise the connection is deselected, giving other connections a chance to perform operations.
Output	`commonBlk`	Four words of common data whose meaning is determined by the debugger. This buffer is allocated by the caller.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
9	`TAPOp_BadParameter`	Failed because `commonBlk = NULL`.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Note

If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.28 TAPOp_ReadMICEFlags

Reads various flags stored in the Server. These hold information relating to the state of the target power and reset condition and the state of the user input bits.

```
(tapshare.h)
extern TAPOp_Error TAPOp_ReadMICEFlags(
       unsigned8   connectId,
       unsigned8   *flags
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
Output	`flags`	This function writes the state of the flags into the variable passed by the caller. The bits set are defined in ***Flags*** below.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
9	`TAPOp_BadParameter`	Failed because `flags = NULL`.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Flags

`TAPOp_FL_TargetPowerOffNow`

The target's power is currently off. This is an error condition.

`TAPOp_FL_TargetPowerHasBeenOff`

The target's power has been off since the last `TAPOp_OpenConnection` call was made. This is also an error condition, because turning the target on and off in the middle of a session will cause problems.

`TAPOp_FL_InResetNow`

The target is currently in Reset (the target's reset signal is currently set). This is generally an error condition.

`TAPOp_FL_TargetHasBeenReset`

The target has been reset since the last `TAPOp_OpenConnection` call was made. This is generally an error condition.

`TAPOp_FL_UserIn1`

The state of the user-defined input signal 1 from Multi-ICE.

`TAPOp_FL_UserIn2`

The state of the user-defined input signal 2 from Multi-ICE.

`TAPOp_FL_UserOut1`

Current state of user-defined output 1 from Multi-ICE.

`TAPOp_FL_UserOut2`

Current state of user-defined output 2 from Multi-ICE.

5.5.29 TAPOp_ReadPrivateFlags

Reads the private word of flags for this processor.

```
(tapshare.h)
extern TAPOp_Error TAPOp_ReadPrivateFlags(
       unsigned8   connectId,
       unsigned32  *flags
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
Output	`flags`	Private word of flags. The flag bits are the `TAPOp_` flags described in ***Flags*** below.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
9	`TAPOp_BadParameter`	Failed because `flags = NULL`.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Flags

`TAPOp_ProcRunning`

A TAPOp client should set this flag when the processor starts executing code. It should be cleared when the processor halts. This can be used by the Multi-ICE Server to indicate whether or not other processors should be started or stopped according to the Multi-ICE user's requirements. This is a write-only flag for the TAPOp client.

`TAPOp_ProcHasStopped`
`TAPOp_ProcStoppedByServer`

These two flags are used to determine if and why a processor has stopped. A client should poll the `TAPOp_ProcHasStopped` flag when the processor is running. If it set, the `TAPOp_ProcStoppedByServer` flag will indicate why. If it is set, the Multi-ICE Server has stopped the processor because some other processor has stopped and a synchronized stop condition was set up. If `TAPOp_ProcStoppedByServer` is not set, the processor has stopped of its own accord—for example, because it hit a breakpoint. These flags are read-only for a client.

`TAPOp_DownloadingCode`

A debugger should set this flag immediately before starting to download code to the target processor. This allows a user output bit to be set when this occurs, which is potentially useful on a system that can switch between very slow and very fast clocks, as fast clocking will speed up download considerably. Similarly, when the download has completed, this bit should be cleared. This flag is read-only for the Server.

`TAPOp_ProcStartREQ`
`TAPOp_ProcStartACK`

These two flags control synchronized starting of processors.
If `TAPOp_UserWantsSyncStart` is set, the debugger should set `TAPOp_ProcStartREQ` to request the Server to start the processor. When all the debuggers have set their `TAPOp_ProcStartREQ` flags, the Server will start all processors together, and set the `TAPOp_ProcStartACK` flag. TAPOp_ProcStartREQ is read-only for the Server. `TAPOp_ProcStartACK` is read-only for a client.

`TAPOp_UserWantsSyncStart`
`TAPOp_UserWantsSyncStop`

These two flags are read-only, and will be set by the Server if the user has selected sync start and/or stop from the dialog. These flags are read-only for a client.

5.5.30 TAPOp_RunBufferedMacro

This is the same as `TAPOp_RunMacro` except the variable data is not passed in this call; it has already been downloaded into one of two buffers using `TAPOp_FillMacroBuffer`. This is useful for two reasons:

- `TAPOp_FillMacroBuffer` cannot return a BUSY (`TAPOp_UnableToSelect`) error, so under normal circumstances it does not fail. This means that the variable data for a `RunMacro` call does not need to be re-sent if the Server is busy. Because the data transfer takes most of the time and processor bandwidth, this improves performance on multi-processor systems where the Server often gives BUSY replies.
- As there are two buffers, a multi-threaded client can be loading one buffer while running a macro from the other. This overlap of the data transfer and execute portions of the `RunMacro` calls improves performance and ensures that the interface between client and Server is fully utilized. This is particularly important on slow interfaces (for example, Ethernet).

Like `TAPOp_RunMacro`, this call returns `TAPOp_UnableToSelect` when the Server is busy and it should be handled in the same way. The overhead of re-sending this call is small as it does not contain the variable data block.

```
(tapmacro.h)
TAPOp_Error TAPOp_RunBufferedMacro(
        unsigned8   connectId,
        unsigned8   macroNo,
        unsigned8   bufferNo,
        int         *lineno_error,
        int         *loopno_error,
        void        *resultValues,
        int         *resultSize,
        int         nTimes,
        unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection.`
	`macroNo`	The number of the macro to run (local to this connection).
	`bufferNo`	The number of the buffer that contains the variable data block. This must be 0 or 1 and the data must have been loaded before this call (see above).
	`nTimes`	The number of times the macro is run (parameters are read and results are written cumulatively from/to the data arrays).

	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.
In/Out	`resultSize`	On entry, this indicates the maximum amount of data that can fit in `resultValues`. On exit, the actual amount of data in `resultValues` is returned.
Output	`lineno_error`	The macro line number that caused the error. This is only valid if `TAPOp_RunBufferedMacro` returned an error. `0` indicates the call to `TAPOp_RunBufferedMacro` failed—for example, an undefined macro number was specified `1,2,3..` indicate failure on line 1, 2, 3 and so on of the macro
	`loopno_error`	Macro loop number which caused the error. This is only valid if `TAPOp_RunBufferedMacro` actually returned an error. For example, if a macro call has `nTimes=3` and it fails on the third time the macro is run, `loopno_error` will be 3.
	`resultValues`	A block of data into which the results of the functions called by this macro are placed. This is allocated by the caller.

Return

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
36	`TAPOp_CouldNotBuildCompleteParameterList`	Parameters were missing during either the macro define or execution.
*	`Other`	Any other errors that the components of the macro can return.

5.5.31 TAPOp_RunMacro

Runs a previously defined macro, passing in the variable data and reading out any results from running the macro.

```
(tapmacro.h)
extern TAPOp_Error TAPOp_RunMacro(
      unsigned8   connectId,
      unsigned8   macroNo,
      void        *variableParamValues,
      int         paramSize,
      int         *lineno_error,
      int         *loopno_error,
      void        *resultValues,
      int         *resultSize,
      int         nTimes,
      unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`macroNo`	The number of the macro to run (local to this connection).
	`variableParamValues`	A block of data holding the variable parameters for this invocation of this macro.
	`paramSize`	The size of the block of variable parameter data. This is needed so the data can be easily sent over RPC.
	`nTimes`	The number of times the macro is run (parameters are read and results are written cumulatively from/to the data arrays).
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise the connection is deselected, giving other connections a chance to perform operations.
In/Out	`resultSize`	On entry, this indicates the maximum amount of data that will fit in `resultValues`. On exit, the actual amount of data in `resultValues` is returned.
Output	`lineno_error`	Gives the macro line number which caused the error. This is only valid if `TAPOp_RunMacro` returned an error. The numbers start from 0.

`loopno_error`	Macro loop number which caused the error. This is only valid if `TAPOp_RunMacro` actually returned an error. For example, if a macro call has `nTimes=3` 3 and it fails on the third time the macro is run, `loopno_error` will be 3.
`resultValues`	A block of data where the results of the functions called by this macro are put. This is allocated by the caller.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
36	`TAPOp_CouldNotBuildCompleteParameterList`	Parameters were missing during either the macro define or execution.
*	`Other`	Any other errors the components of the macro can return.

Note

Some parameters may have been fixed at define time. These do not need to be passed in at runtime.

5.5.32 TAPOp_SetControlMacros

Tells the Multi-ICE Server which macros to use for synchronized starting/stopping of processors. Before calling this function, the client must define three macros that perform the following functions:

`eventMacro`

This macro is periodically run by the Server while the processor is running (indicated by the state of the private flags set by the debugger). The last line of this macro must be `TAPOp_AccessDR_RW` or `ARMTAP_AccessDR_RW`. The result of the data read from this function is AND'ed with `eventMask` and XOR'ed with `eventXOR`. If the result is zero, an event is recognized. When the Server recognizes an event, it runs the other processors' `stopMacros` (if marked for synchronized stopping on the Server).

The typical use of `eventMacro` is to recognize a breakpoint condition. This is not the only possible use; any event that can be recognized using a data chain read and mask can be set up as a trigger for `stopMacro` (which does not have to be written to stop the processor).

`preExecMacro`

If the `TAPOp_PreExecMacroRequired` flag is set, this macro will be run before the `executeMacro`; it is used to set up the processor state before actually starting it. The Server runs the execute macros in the following order:

1. All the `preExec` macros are run one by one for each processor.
2. All the mergable execute macros are run in one pass of Run-Test/Idle as described below.
3. Any non-mergable execute macros are run.
4. All the `postExec` macros are run one by one for each processor.

All the above steps are completed in one block—that is, no other `TAPOp` operations take place during these steps. The sequence above will commence when all the processors are ready to start—in other words, all the processors that sync start is required for have had a `TAPOp_ProcStartREQ` from their respective debuggers.

`executeMacro`

This macro is run by the Server to start execution of the processor. If the last line of the macro is an IR write and go through Run-Test/Idle (as it is on an ARM processor), the IR writes from all the processors are merged. The whole set of IR registers in the scan chain is written in one operation, and the resulting pass through Run-Test/Idle starts all the processors on the same TCK.

`postExecMacro`

If the `TAPOp_PostExecMacroRequired` flag is set, this macro will be run by the Server immediately after the `executeMacro`; it should be used to tidy up after `executeMacro`. Since the last line of `executeMacro` has some constraints (as described above), this extra macro can be used to sort out any loose ends—for example, re-selecting scan chains and putting INTEST in the IR.

`StopMacro`

This macro is run to stop the processor as described above.

Note

The event, stop and combined execute macros should all start and stop in a known state, as they can be run in any order by the Server.

Any `TAPOp` operations that a client makes while the processor is running with synchronous stopping enabled must also start and stop in this same state, and should also not deselect until these have finished (to ensure that running one of these macros cannot get in at an inappropriate point).

```
(tapmacro.h)
TAPOp_Error TAPOp_SetControlMacros(
       unsigned8   connectId,
       unsigned8   flags,
       unsigned8   eventMacroNo,
       unsigned8   preExecMacroNo,
       unsigned8   executeMacroNo,
       unsigned8   postExecMacroNo,
       unsigned8   stopMacroNo,
       ScanData40  *eventMask,
       ScanData40  *eventXOR
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`flags`	Various flags (defined in ***Flags*** on page 5-72).
	`eventMacroNo`	The number of the macro to use as `eventMacro`.
	`preExecMacroNo`	The number of the macro to use as `preExecMacro`.
	`executeMacroNo`	The number of the macro to use as `executeMacro`.
	`postExecMacroNo`	The number of the macro to use as `postExecMacro`.
	`stopMacroNo`	The number of the macro to use as `stopMacro`.
	`eventMask`	The mask for detecting an event (see above).
	`eventXOR`	XOR for detecting an event (see above).

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
20	`TAPOp_UndefinedMacro`	The macro does not exist.

Flags

`TAPOp_SyncStopSupported`

Set this flag to indicate that the client supports synchronized stopping. The Server will then run the event macro periodically, and events will cause the stop macros to be run as described below. Only set this flag if the client has defined a suitable event macro, stop macro, `eventMask` and `eventXOR`.

`TAPOp_SyncStartSupported`

Set this following flag to indicate that the client supports synchronized starting. The Server will then wait for this client's `TAPOp_ProcStartREQ` private flag before starting the processor. Only set this flag if the client has defined a suitable execute macro.

`TAPOp_PreExecMacroUsed`

Set this flag if a `preExec` macro is required.

`TAPOp_PostExecMacroUsed`

Set this flag if a `postExec` macro is required.

5.5.33 TAPOp_SetLogging

Switches debug logging for TAPOp functions on or off, for a particular TAP controller.

This call does not require a selected connection to a TAP controller, as it is just changing the level of debugging output.

```
(tapop.h)
extern TAPOp_Error TAPOp_SetLogging(
       unsigned8   connectId,
       unsigned32  flags
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`flags`	If 0, this switches logging off; otherwise, it is switched on.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
27	`TAPOp_CannotEnableLogging`	The Multi-Ice Server could not enable logging.

5.5.34 TAPOp_SetSysResetSignal

Sets or clears the System Reset Signal.

```
(tapop.h)
extern TAPOp_Error TAPOp_SetSysResetSignal(
        unsigned8 connectId,
        unsigned8 level,
        unsigned8 deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.	
	`level`	Specifies whether to set or clear System Reset:	
		`0`	clears the System Reset Signal
		`other`	sets the System Reset Signal
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.	

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Notes

1. If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call attempts to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.
2. When setting and then clearing System Reset, the Multi-ICE Server automatically clears the sticky reset bit in the status register. This makes it possible for applications to perform a system reset and then continue with the same connection without the Server forcing a disconnect and reconnect to take place.

   If you want the Server to take account of the system reset and force this (and other) connections to disconnect and reconnect, call `TAPOp_ReadMICEFlags` while system reset is asserted.

5.5.35 TAPOp_WriteCommonData

Writes to the data block that is common to all the debuggers connected to Multi-ICE.

In order to safely write to the common data block, use an atomic Read-Modify-Write sequence. A call of `TAPOp_ReadCommonData` and `TAPOp_WriteCommonData` can be made atomic simply by not ‘deselecting’ after the Read call.

```
(tapshare.h)
extern TAPOp_Error TAPOp_WriteCommonData(
       unsigned8   connectId,
       unsigned32  *commonBlk,
       unsigned8   deselect
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`commonBlk`	Four words of common data, whose meaning is defined by the debugger.
	`deselect`	If 0, the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Note

If a connection to this TAP controller has been selected already, this operation happens automatically. If not, this call tries to select the connection. If the connection cannot be selected (because another TAP controller is being accessed), this operation is not performed and `TAPOp_UnableToSelect` is returned. The caller should try again later.

5.5.36 TAPOp_WriteMICEUser1

Sets the level of the user-defined Multi-ICE User 1 output bit.

```
(tapshare.h)
extern TAPOp_Error TAPOp_WriteMICEUser1(
       unsigned8   connectId,
       unsigned8   user1
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`user1`	Bit 0 indicates the state to which the User 1 signal is set.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
26	`TAPOp_NotAllocatedToThisConnection`	The User Output bit is not allocated to the debugger by the Multi-ICE Server.

Note

For this procedure to have an effect on the state of the output bits, the **Set by Driver** option must be chosen in the Server-User output bit settings. The TAP position for the connection must also be selected.

5.5.37 TAPOp_WriteMICEUser2

Sets the level of the user-defined Multi-ICE User 2 output bit.

```
(tapshare.h)
extern TAPOp_Error TAPOp_WriteMICEUser2(
       unsigned8   connectId,
       unsigned8   user2
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`user2`	Bit 0 indicates the state to which the User 2 signal is set.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.
26	`TAPOp_NotAllocatedToThisConnection`	The User Output bit is not allocated to the debugger by the Multi-ICE Server.

Note

For this procedure to have an effect on the state of the output bits, the **Set by Driver** option must be chosen in the Server-User output bit settings. The TAP position for the connection must also be selected.

5.5.38 TAPOp_WritePrivateFlags

To write the private word of flags for this processor.

```
(tapshare.h)
extern TAPOp_Error TAPOp_WritePrivateFlags(
       unsigned8   connectId,
       unsigned32  flags
);
```

Arguments

Input	`connectId`	Connection ID, as returned by `TAPOp_OpenConnection`.
	`flags`	Private word of flags. The flag bits are described in ***Flags*** below.

Returns

0	`TAPOp_NoError`	No error.
2	`TAPOp_UnableToSelect`	Connection could not be made.
7	`TAPOp_NoSuchConnection`	The `connectId` was not recognized.
19	`TAPOp_RPC_ConnectionFail`	The RPC connection died while processing this request.

Flags

`TAPOp_ProcRunning`

A TAPOp client should set this flag when the processor starts executing code. It should be cleared when the processor halts. This can be used by the Multi-ICE Server to indicate whether or not other processors should be started or stopped according to the Multi-ICE user’s requirements. This is a write-only flag for the TAPOp client.

`TAPOp_ProcHasStopped`
`TAPOp_ProcStoppedByServer`

These two flags are used to determine if and why a processor has stopped. A client should poll the `TAPOp_ProcHasStopped` flag when the processor is running. If it set, the `TAPOp_ProcStoppedByServer` flag will indicate why. If it is set, the Multi-ICE Server has stopped the processor because some other processor has stopped and a synchronized stop condition was set up. If `TAPOp_ProcStoppedByServer` is not set, the processor has stopped of its own accord—for example, because it hit a breakpoint. These flags are read-only for a client.

`TAPOp_DownloadingCode`

A debugger should set this flag immediately before starting to download code to the target processor. This allows a user output bit to be set when this occurs, which is potentially useful on a system that can switch between very slow and very fast clocks, as fast clocking will speed up download considerably. Similarly, when the download has completed, this bit should be cleared. This flag is read-only for the Server.

`TAPOp_ProcStartREQ`
`TAPOp_ProcStartACK`

These two flags control synchronized starting of processors.
If `TAPOp_UserWantsSyncStart` is set, the debugger should set `TAPOp_ProcStartREQ` to request the Server to start the processor. When all the debuggers have set their `TAPOp_ProcStartREQ` flags, the Server will start all processors together, and set the `TAPOp_ProcStartACK` flag. TAPOp_ProcStartREQ is read-only for the Server. `TAPOp_ProcStartACK` is read-only for a client.

`TAPOp_UserWantsSyncStart`
`TAPOp_UserWantsSyncStop`

These two flags are read-only, and will be set by the Server if the user has selected sync start and/or stop from the dialog. These flags are read-only for a client.

5.6 TAPOp Error Codes

This section lists all the TAPOp error codes in numerical order.

0 `TAPOp_NoError`
No error. The operation completed successfully.

1 `TAPOp_OutOfStore`
Ran out of memory. This is a serious error.

2 `TAPOp_UnableToSelect`
Unable to select a connection for this TAP controller because another TAP controller is being accessed. Try again later.

3 `TAPOp_TAPNotPresent`
The specified TAP controller is not present.

4 `TAPOp_NotInitialised`
The Multi-ICE Server has not yet been initialized.

5 `TAPOp_TooManyConnections`
The Multi-ICE Server has no free connections.

6 `TAPOp_ClientsStillConnected`
The Multi-ICE Server cannot finalize or reconfigure while clients are attached.

7 `TAPOp_NoSuchConnection`
Invalid connection ID was presented.

8 `TAPOp_InBadTAPState`
TAP controllers in unknown or incorrect state so Multi-ICE could not perform the request.

9 `TAPOp_BadParameter`
Invalid parameter value specified.

10 `TAPOp_ConnectionStillSelected`
Connection still connected.

11 `TAPOp_IRSCTooLong`
The combined length of all instruction registers is too great for this version of Multi-ICE.

12 `TAPOp_SCSRTooLong`
One or more of the Scan Chain Select registers was too long.

13 `TAPOp_ScanChainAlreadyClaimed`
Connection could not be opened because one of the required scan chains is claimed by another connection.

14 `TAPOp_BadConfigurationData`
The configuration data is unsuitable for this implementation.

15 `TAPOp_DriverLimitExceeded`
On a call to `TAPOp_GetDriverDetails`, the array size allocation was not big enough to return details of all the drivers found by Multi-ICE.

16 `TAPOp_UnknownDriverName`
On a call to `TAPOp_OpenConnection`, the driver name was not one that was passed back to the client by `TAPOp_GetDriverDetails.`

17 `TAPOp_CouldNotOpenPort`
Could not open the requested port.

18 `TAPOp_ParameterConflicts`
A parameter conflicts with the configuration data.

19 `TAPOp_RPC_ConnectionFail`
RPC connection failure during a call.

20 `TAPOp_UndefinedMacro`
Tried to run a macro that has not been defined.

21 `TAPOp_TooManyMacros`
Tried to create more macros than `MAX_MACROS` allows. The maximum is 1000.

22 `TAPOp_TooManyMacroLines`
Tried to add more lines to a macro than `MAX_MACRO_LINES` allows.

23 `TAPOp_BadFixedParamNo`
When defining a macro line, reference to a non-existent parameter was given in the list of fixed parameters.

24 `TAPOp_OutOfMacroResultSpace`
Ran out of macro result space.

25 `TAPOp_MaskAndTestFailed`
The Mask and Test operation did not match.

26 `TAPOp_NotAllocatedToThisConnection`
Resource is not allocated to this connection.

27 `TAPOp_CannotEnableLogging`
The log file is not set up.

28 `TAPOp_TooManyProcessors`
Maximum number of processors has been exceeded.

29 `TAPOp_IncompatibleModel`
The hardware connected is not the correct model version.

30 `TAPOp_CouldNotOpenMulFile`
A MUL file could not be opened.

31 `TAPOp_BadlyFormattedMulFile`
A MUL file was corrupt.

32 `TAPOp_AnySeqUsedBadPath`

An Exit2-IR/DR -> Shift-IR/DR transition was performed.

33 `TAPOp_AnySeqWrongIRLength`

The wrong number of Shift-IR TCKs was detected.

34 `TAPOp_UnknownProcedureName`

Unknown procedure name in DefineMacro call.

35 `TAPOp_CantUseProcInMacro`

Procedure specified in DefineMacro call cannot be run in a macro.

36 `TAPOp_CouldNotBuildCompleteParameterList`

While attempting to run a macro, the Server was unable to build a complete parameter list from the parameters supplied when it was defined and when it was executed.

66 `TAPOp_MultiICEHWNotPoweredUp`

The Server could not connect to the Multi-ICE hardware. It is possibly not powered up.

84 `TAPOp_ParallelInterfaceTimeout`

Parallel port interface timeout occurred.

Appendix A

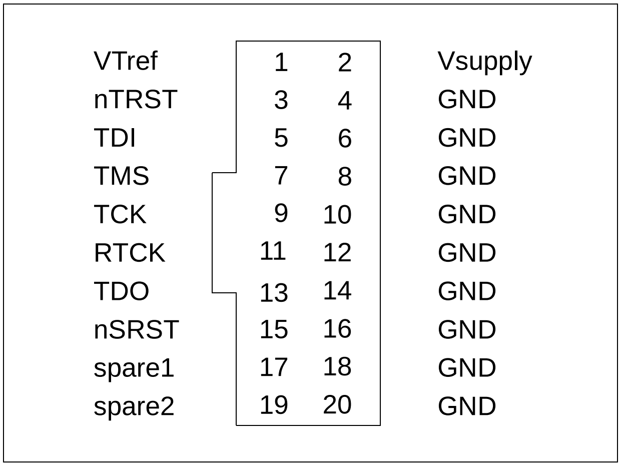
JTAG Interface Connections

This appendix describes and illustrates the JTAG pin connections.

JTAG Interface Connections

A.1 Multi-ICE JTAG Interface Connections

This section displays the JTAG pin connections and lists any applicable notes for each pin.



VTref	1	2	Vsupply
nTRST	3	4	GND
TDI	5	6	GND
TMS	7	8	GND
TCK	9	10	GND
RTCK	11	12	GND
TDO	13	14	GND
nSRST	15	16	GND
spare1	17	18	GND
spare2	19	20	GND

Figure A-1: JTAG pin connections

The connector is a 20-way header which mates with IDC sockets mounted on a ribbon cable.

Note *All GND pins should be connected to 0V on the target board.*

5.6.1 Notes

Pin 1	**VTref**	This is the target reference voltage. It indicates that the target has power and it is also used to create the logic-level reference for the input comparators on **TDO** and **RTCK**. It also controls the output logic levels to the target. It is normally fed from **Vdd** on the target board and may have a series resistor (though this is not recommended).
Pin 2	**Vsupply**	This is the supply voltage to Multi-ICE. It draws its supply current from this pin via a step-up voltage convertor. This is normally fed by the target **Vdd** which *must not* have a series resistor in the feed to this pin. If the target supply voltage or its current capability is too low, this path is broken by an external power jack on an inline header.

Pin 3	**nTRST**	Open collector output from Multi-ICE to the Reset signal on the target JTAG port. This pin should be pulled high on the target to avoid unintentional resets when there is no connection.
Pin 4	**GND**	
Pin 5	**TDI**	Test Data In signal from Multi-ICE to the target JTAG port. It is recommended that this pin is pulled to a defined state.
Pin 6	**GND**	
Pin 7	**TMS**	Test Mode signal from Multi-ICE to the target JTAG port. This pin should be pulled up on the target so that the effect of any spurious **TCK**s when there is no connection is benign.
Pin 8	**GND**	
Pin 9	**TCK**	Test Clock signal from Multi-ICE to the target JTAG port. It is recommended that this pin is pulled to a defined state.
Pin 10	**GND**	
Pin 11	**RTCK**	Return Test Clock signal from the target JTAG port to Multi-ICE. Targets that use TrackingICE technology need to synchronize the JTAG inputs to internal clocks. To assist in meeting this requirement, a returned (and re-timed) **TCK** can be used to dynamically control the **TCK** rate. Multi-ICE provides Adaptive Clock Timing, which waits for **TCK** changes to be echoed correctly before making further changes. Targets that do not need to process **TCK** can simply loop the pins on the PCB, thus at least calibrating out the round-trip delay (ribbon cable and I/O drivers).
Pin 12	**GND**	
Pin 13	**TDO**	Test Data Out from the target JTAG port to Multi-ICE.
Pin 14	**GND**	
Pin 15	**nSRST**	Open collector output from Multi-ICE to the target system reset. This is also an input to Multi-ICE so that a reset initiated on the target may be reported to the debugger. This pin should be pulled up on the target to avoid unintentional resets when there is no connection.
Pin 16	**GND**	
Pin 17	**spare 1**	This pin is not connected in the Multi-ICE unit. It is reserved for compatibility with other equipment to be used as a debug request signal to the target system.
Pin 18	**GND**	

Pin 19	**spare 2**	This pin is not connected in the Multi-ICE unit. It is reserved for compatibility with other equipment to be used as a debug acknowledge signal from the target system.
Pin 20	**GND**	

A.2 Multi-ICE Target Interface Voltage Levels

The graph in ***Figure A-2: Voltage relationships*** shows the relationship between the target voltage and the Multi-ICE input voltage threshold and output voltage.

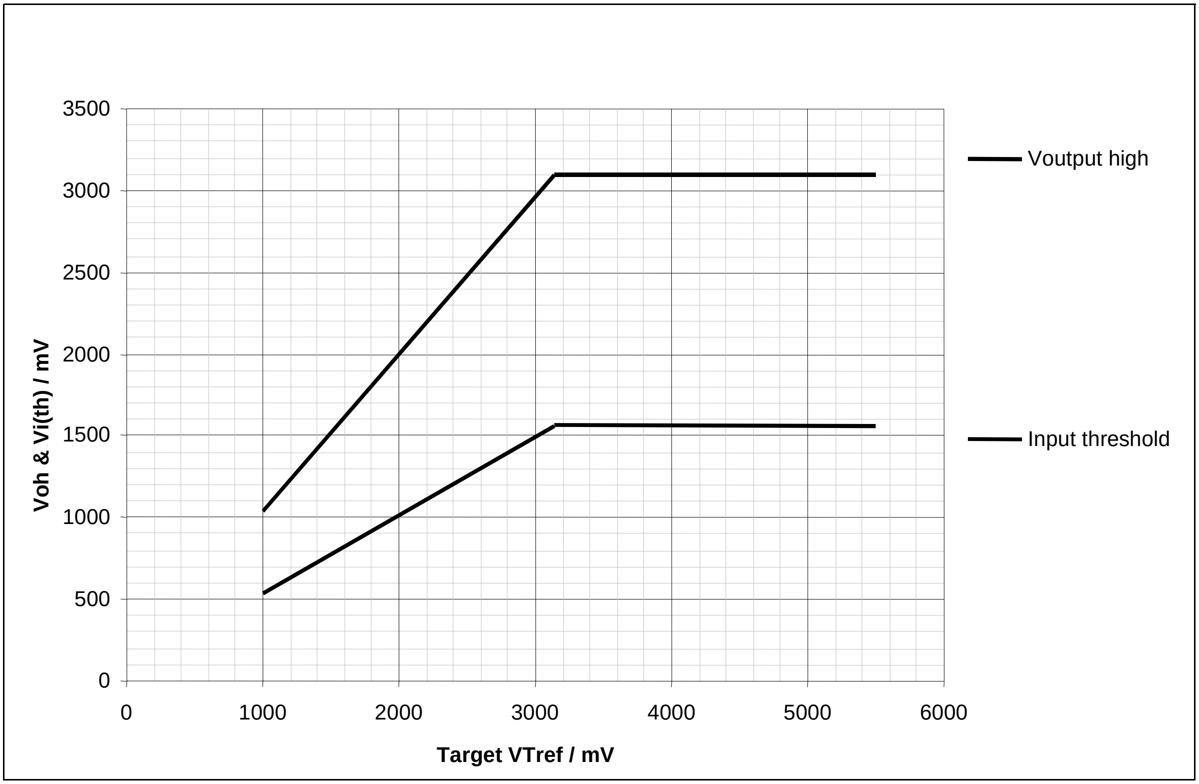


Figure A-2: Voltage relationships

A.3 TCK Frequencies

Table A-1: TCK frequencies gives the values that must be entered into the TCK fields on the JTAG settings dialog for a particular TCK frequency. For example, for a 3.33MHz TCK rate, use a value of 2 for TCK high and TCK low.

Freq (kHz)	Period (ns)	Value
10000.00	50	0
5000.00	100	1
3333.33	150	2
2500.00	200	3
2000.00	250	4
1666.67	300	5
1428.57	350	6
1250.00	400	7
1111.11	450	8
1000.00	500	9
909.09	550	10
833.33	600	11
769.23	650	12
714.29	700	13
666.67	750	14
625.00	800	15
588.24	850	16
555.56	900	17
526.32	950	18
500.00	1000	19
476.19	1050	20
454.55	1100	21
434.78	1150	22
416.67	1200	23
400.00	1250	24
384.62	1300	25
370.37	1350	26
357.14	1400	27
344.83	1450	28
333.33	1500	29
322.58	1550	30
312.50	1600	31
294.12	1700	48
277.78	1800	49
263.16	1900	50
250.00	2000	51
238.10	2100	52
227.27	2200	53
217.39	2300	54
208.33	2400	55
200.00	2500	56
192.31	2600	57

Table A-1: TCK frequencies

Freq (kHz)	Period (ns)	Value
185.19	2700	58
178.57	2800	59
172.41	2900	60
166.67	3000	61
147.06	3400	80
138.89	3600	81
131.58	3800	82
125.00	4000	83
119.05	4200	84
113.64	4400	85
108.70	4600	86
104.17	4800	87
100.00	5000	88
96.15	5200	89
92.59	5400	90
89.29	5600	91
86.21	5800	92
83.33	6000	93
80.65	6200	94
78.13	6400	95
73.53	6800	112
69.44	7200	113
65.79	7600	114
62.50	8000	115
59.52	8400	116
56.82	8800	117
54.35	9200	118
52.08	9600	119
50.00	10000	120
48.08	10400	121
46.30	10800	122
44.64	11200	123
43.10	11600	124
41.67	12000	125
40.32	12400	126
39.06	12800	127
36.76	13600	144
34.72	14400	145
32.89	15200	146
31.25	16000	147
29.76	16800	148
28.41	17600	149
27.17	18400	150
26.04	19200	151
25.00	20000	152
24.04	20800	153
23.15	21600	154
22.32	22400	155

Table A-1: TCK frequencies (Continued)

Freq (kHz)	Period (ns)	Value
21.55	23200	156
20.83	24000	157
20.16	24800	158
19.53	25600	159
18.38	27200	176
17.36	28800	177
16.45	30400	178
15.63	32000	179
14.88	33600	180
14.20	35200	181
13.59	36800	182
13.02	38400	183
12.50	40000	184
12.02	41600	185
11.57	43200	186
11.16	44800	187
10.78	46400	188
10.42	48000	189
10.08	49600	190
9.77	51200	191
9.19	54400	208
8.68	57600	209
8.22	60800	210
7.81	64000	211
7.44	67200	212
7.10	70400	213
6.79	73600	214
6.51	76800	215
6.25	80000	216
6.01	83200	217
5.79	86400	218
5.58	89600	219
5.39	92800	220
5.21	96000	221
5.04	99200	222
4.88	102400	223
4.60	108800	240
4.34	115200	241
4.11	121600	242
3.91	128000	243
3.72	134400	244
3.55	140800	245
3.40	147200	246
3.26	153600	247
3.13	160000	248
3.00	166400	249
2.89	172800	250
2.79	179200	251

Table A-1: TCK frequencies (Continued)

Freq (kHz)	Period (ns)	Value
2.69	185600	252
2.60	192000	253
2.52	198400	254
2.44	204800	255

Table A-1: TCK frequencies (Continued)

A.4 TCK Values

Table A-2: TCK values shows the corresponding frequencies for the TCK fields on the JTAG settings dialog. As an example, for a value of 4 for TCK high and TCK low, the TCK rate is 2MHz.

Value	Period (ns)	Freq (kHz)
0	50	10000.00
1	100	5000.00
2	150	3333.33
3	200	2500.00
4	250	2000.00
5	300	1666.67
6	350	1428.57
7	400	1250.00
8	450	1111.11
9	500	1000.00
10	550	909.09
11	600	833.33
12	650	769.23
13	700	714.29
14	750	666.67
15	800	625.00
16	850	588.24
17	900	555.56
18	950	526.32
19	1000	500.00
20	1050	476.19
21	1100	454.55
22	1150	434.78
23	1200	416.67
24	1250	400.00
25	1300	384.62
26	1350	370.37
27	1400	357.14
28	1450	344.83
29	1500	333.33
30	1550	322.58
31	1600	312.50
32	100	5000.00
33	200	2500.00
34	300	1666.67
35	400	1250.00
36	500	1000.00
37	600	833.33
38	700	714.29
39	800	625.00
40	900	555.56
41	1000	500.00

Table A-2: TCK values

Value	Period (ns)	Freq (kHz)
42	1100	454.55
43	1200	416.67
44	1300	384.62
45	1400	357.14
46	1500	333.33
47	1600	312.50
48	1700	294.12
49	1800	277.78
50	1900	263.16
51	2000	250.00
52	2100	238.10
53	2200	227.27
54	2300	217.39
55	2400	208.33
56	2500	200.00
57	2600	192.31
58	2700	185.19
59	2800	178.57
60	2900	172.41
61	3000	166.67
62	3100	161.29
63	3200	156.25
64	200	2500.00
65	400	1250.00
66	600	833.33
67	800	625.00
68	1000	500.00
69	1200	416.67
70	1400	357.14
71	1600	312.50
72	1800	277.78
73	2000	250.00
74	2200	227.27
75	2400	208.33
76	2600	192.31
77	2800	178.57
78	3000	166.67
79	3200	156.25
80	3400	147.06
81	3600	138.89
82	3800	131.58
83	4000	125.00
84	4200	119.05
85	4400	113.64
86	4600	108.70
87	4800	104.17
88	5000	100.00
89	5200	96.15

Table A-2: TCK values (Continued)

Value	Period (ns)	Freq (kHz)
90	5400	92.59
91	5600	89.29
92	5800	86.21
93	6000	83.33
94	6200	80.65
95	6400	78.13
96	400	1250.00
97	800	625.00
98	1200	416.67
99	1600	312.50
100	2000	250.00
101	2400	208.33
102	2800	178.57
103	3200	156.25
104	3600	138.89
105	4000	125.00
106	4400	113.64
107	4800	104.17
108	5200	96.15
109	5600	89.29
110	6000	83.33
111	6400	78.13
112	6800	73.53
113	7200	69.44
114	7600	65.79
115	8000	62.50
116	8400	59.52
117	8800	56.82
118	9200	54.35
119	9600	52.08
120	10000	50.00
121	10400	48.08
122	10800	46.30
123	11200	44.64
124	11600	43.10
125	12000	41.67
126	12400	40.32
127	12800	39.06
128	800	625.00
129	1600	312.50
130	2400	208.33
131	3200	156.25
132	4000	125.00
133	4800	104.17
134	5600	89.29
135	6400	78.13
136	7200	69.44
137	8000	62.50

Table A-2: TCK values (Continued)

Value	Period (ns)	Freq (kHz)
138	8800	56.82
139	9600	52.08
140	10400	48.08
141	11200	44.64
142	12000	41.67
143	12800	39.06
144	13600	36.76
145	14400	34.72
146	15200	32.89
147	16000	31.25
148	16800	29.76
149	17600	28.41
150	18400	27.17
151	19200	26.04
152	20000	25.00
153	20800	24.04
154	21600	23.15
155	22400	22.32
156	23200	21.55
157	24000	20.83
158	24800	20.16
159	25600	19.53
160	1600	312.50
161	3200	156.25
162	4800	104.17
163	6400	78.13
164	8000	62.50
165	9600	52.08
166	11200	44.64
167	12800	39.06
168	14400	34.72
169	16000	31.25
170	17600	28.41
171	19200	26.04
172	20800	24.04
173	22400	22.32
174	24000	20.83
175	25600	19.53
176	27200	18.38
177	28800	17.36
178	30400	16.45
179	32000	15.63
180	33600	14.88
181	35200	14.20
182	36800	13.59
183	38400	13.02
184	40000	12.50
185	41600	12.02

Table A-2: TCK values (Continued)

Value	Period (ns)	Freq (kHz)
186	43200	11.57
187	44800	11.16
188	46400	10.78
189	48000	10.42
190	49600	10.08
191	51200	9.77
192	3200	156.25
193	6400	78.13
194	9600	52.08
195	12800	39.06
196	16000	31.25
197	19200	26.04
198	22400	22.32
199	25600	19.53
200	28800	17.36
201	32000	15.63
202	35200	14.20
203	38400	13.02
204	41600	12.02
205	44800	11.16
206	48000	10.42
207	51200	9.77
208	54400	9.19
209	57600	8.68

Value	Period (ns)	Freq (kHz)
210	60800	8.22
211	64000	7.81
212	67200	7.44
213	70400	7.10
214	73600	6.79
215	76800	6.51
216	80000	6.25
217	83200	6.01
218	86400	5.79
219	89600	5.58
220	92800	5.39
221	96000	5.21
222	99200	5.04
223	102400	4.88
224	6400	78.13
225	12800	39.06
226	19200	26.04
227	25600	19.53
228	32000	15.63
229	38400	13.02
230	44800	11.16
231	51200	9.77
232	57600	8.68
233	64000	7.81

Table A-2: TCK values (Continued)

Value	Period (ns)	Freq (kHz)
234	70400	7.10
235	76800	6.51
236	83200	6.01
237	89600	5.58
238	96000	5.21
239	102400	4.88
240	108800	4.60
241	115200	4.34
242	121600	4.11
243	128000	3.91
244	134400	3.72

Value	Period (ns)	Freq (kHz)
245	140800	3.55
246	147200	3.40
247	153600	3.26
248	160000	3.13
249	166400	3.00
250	172800	2.89
251	179200	2.79
252	185600	2.69
253	192000	2.60
254	198400	2.52
255	204800	2.44

Table A-2: TCK values (Continued)

B

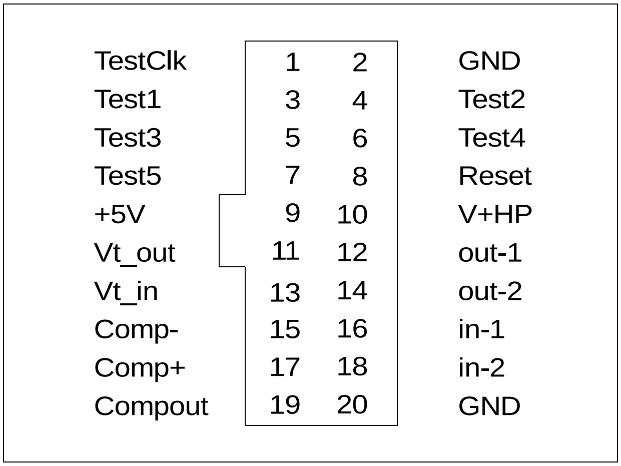
User I/O Pin Connections

This appendix describes and illustrates the additional input and output connections provided in Multi-ICE.

User I/O Pin Connections

B.1 Multi-ICE USER I/O Pin Connections

The User I/O connector is situated under the removable cover on the Multi-ICE unit. The connector is a 20-way header which mates with IDC sockets mounted on a ribbon cable.



TestClk	1	2	GND
Test1	3	4	Test2
Test3	5	6	Test4
Test5	7	8	Reset
+5V	9	10	V+HP
Vt_out	11	12	out-1
Vt_in	13	14	out-2
Comp-	15	16	in-1
Comp+	17	18	in-2
Compout	19	20	GND

Figure B-1: User I/O pin connections

If you need to drive one of the user-defined inputs with a signal operating at the target system's logic levels, see the sample circuit in ***Figure 3-3: Converting user-input signals to TTL levels*** on page 3-18.

B.1.1 Notes

Pin 1	**TestClk**	For production test only. This pin must be left unconnected.
Pin 2	**GND**	
Pin 3	**Test1**	For production test only. This pin must be left unconnected.
Pin 4	**Test2**	For production test only. This pin must be left unconnected.
Pin 5	**Test3**	For production test only. This pin must be left unconnected.
Pin 6	**Test4**	For production test only. This pin must be left unconnected.
Pin 7	**Test5**	For production test only. This pin must be left unconnected.
Pin 8	**Reset**	For production test only. This pin must be left unconnected.

Pin 9	**+5V**	This is intended for use as a supply for a small amount of external logic circuitry. There is a recommended current limit of 20mA from this pin. It should be remembered that due to the DC-DC converter, the additional current taken from the target to supply any external logic will be approximately: `Iout * ( 5V / target voltage)`
Pin 10	**V+HP**	For production test only. This pin must be left unconnected.
Pin 11	**Vt_out**	For production test only. This pin must be left unconnected.
Pin 12	**out-1**	This is a user output bit as described in ***3.6 User Output Bits*** on page 3-17.
Pin 13	**Vt_in**	This is the voltage threshold used for input logic detection, derived from the target logic reference voltage (**VTref** on pin 1 of the target connector). This can be used as one of the inputs to the comparator if a target logic level is being monitored.
Pin 14	**out-2**	This is a user output bit as described in ***3.6 User Output Bits*** on page 3-17.
Pin 15	**Comp-**	This is connected to the inverting input of a spare LM339D comparator for use with the user-defined input/output. Note that there is a 1MΩ pull-down resistor to GND.
Pin 16	**in-1**	This is a user input bit as described in ***3.7 User Input Bits*** on page 3-18. This uses 74ACT family input thresholds and has a 10kΩ pull-up to +5V.
Pin 17	**Comp+**	This is connected to the non-inverting input of a spare LM339D comparator for use with the user-defined input/output. Note that there is a 1MΩ pull-up resistor to +5V.
Pin 18	**in-2**	This is a user input bit as described in ***3.7 User Input Bits*** on page 3-18. This uses 74ACT family input thresholds and has a 10kΩ pull-up to +5V.
Pin 19	**Compout**	This is connected to the output of a spare LM339D comparator for use with the user-defined input/output. Note that this is an open collector so it requires a pull-up resistor (user inputs **in-1** and **in-2** already have suitable pull-ups (10kΩ to 5V)).
Pin 20	**GND**	

C

CP15 Register Mapping

This appendix contains details relating to register mapping information for the ARM7 and ARM9 based processors containing a CP15 device, as well as support information for caches and WinCE features.

CP15 Register Mapping

C.1 Register Mapping

The Multi-ICE Debugger for Windows (MDW) currently only supports the display of Coprocessor 15 registers in a list format where each entry corresponds to a single register and data is viewed/edited as 32 bit hex numbers. The problem with this approach is that when dealing with CP15, some of the physical register numbers are used for more than one function—for example, on the ARM710T device, CP15 r8 is used to either flush the TLB or flush a single TLB entry.

To illustrate this point, the CP15 registers for the relevant processors are laid out in ***Table C-1: ARM710T processor***, ***Table C-2: ARM720T processor*** on page C-4, ***Table C-3: ARM740T*** on page C-5 and ***Table C-4: ARM940T*** on page C-6.

Only the standard registers will appear in the co-processor window, debugger internal variables and/or the data written to a register are used to determine the exact meaning of the access.

The extra debugger internal variables that have been defined are:

- `cp15_cache_selected`
- `cp15_current_memory_area`

Refer to ***4.6.2 Debugger internal variables*** on page 4-22.

C.1.1 ARM710T

Register	Description	Access	Data
r0	ID Register	Read Only	
r1	Configuration Register	Read/Write	Config Value
r2	Translation Table Base Register	Read/Write	Base Address
r3	Domain Access Control Register	Read/Write	Domain Value
r5	Fault Status Register	Read/Write	Fault Value
r6	Fault Address Register	Read/Write	Fault Address
r7	Cache Operations:	Write Only	
	Invalidate ID Cache		SBZ
r8	TLB Operations:	Write Only	
	Invalidate whole TLB		SBZ
	Invalidate Single Entry		Virtual Address

Table C-1: ARM710T processor

The encodings to be used to read/write the registers are as follows:

r0–r3, r5–r6	Data reads/writes occur as expected.
r7	Any value invalidates ID Cache.
r8	Writing 0 causes the whole TLB to be invalidated. Writing {Address} with bit 0 set to 1, causes TLB entry for {Address} to be invalidated.

C.1.2 ARM720T

Register	Description	Access	Data
r0	ID Register	Read Only	
r1	Configuration Register	Read/Write	Config Value
r2	Translation Table Base Register	Read/Write	Base Address
r3	Domain Access Control Register	Read/Write	Domain Value
r5	Fault Status Register	Read/Write	Fault Value
r6	Fault Address Register	Read/Write	Fault Address
r7	Cache Operations: Invalidate ID Cache	Write Only	 SBZ
r8	TLB Operations: Invalidate whole TLB Invalidate Single Entry	Write Only	 SBZ Virtual Address
r13	Process ID Register (WinCE)	Read/Write	Process ID

Table C-2: ARM720T processor

The encodings to be used to read/write the registers are as follows:

r0–r3, r5–r6	Data reads/writes occur as expected.
r7	Any value invalidates ID Cache.
r8	Writing 0 causes the whole TLB to be invalidated. Writing {Address} with bit 0 set to 1, causes TLB entry for {Address} to be invalidated.
r13	Data reads/writes occur as expected.

C.1.3 ARM740T

Register	Description	Access	Data
r0	ID Register	Read Only	
r1	Configuration Register	Read/Write	Config Value
r2	Cacheable Control	Read/Write	Cache Ctrl Flags
r3	Bufferable Control	Read/Write	Buffer Ctrl Flags
r5	Memory Protection	Read/Write	Memory Prtn Data
r6	Memory Area Definition:	Read/Write	
	Memory Region 0		Base/Size/Enable
	Memory Region 1		Base/Size/Enable
	Memory Region 2		Base/Size/Enable
	Memory Region 3		Base/Size/Enable
	Memory Region 4		Base/Size/Enable
	Memory Region 5		Base/Size/Enable
	Memory Region 6		Base/Size/Enable
	Memory Region 7		Base/Size/Enable
r7	Cache Operations:	Write Only	
	Invalidate ID Cache		SBZ

Table C-3: ARM740T

The encodings to be used to read/write the registers are as follows:

r0–r3,r5	Data reads/writes occur as expected.
r6	Data value read/written will set/read a memory area definition consisting of a base address, a size value, and an enable flag for the Memory area defined by the variable `cp15_current_memory_area`.
r7	Any value invalidates ID Cache

C.1.4 ARM940T

Register	Description	Access	Data
r0	ID Register	Read Only	
r1	Configuration Register	Read/Write	Config Flags
r2	Cacheable Control: Data Cache Control Instruction Cache Control	Read/Write	 D-Cache Ctrl Flags I-Cache Ctrl Flags
r3	Bufferable Control	Read/Write	D-Buffer Ctrl Flags
r5	Memory Protection: Data Cache Control Instruction Cache Control	Read/Write	 D-Prtn Ctrl Flags I-Prtn Ctrl Flags
r6	Memory Area Definition: Data Memory Region 0 Data Memory Region 1 Data Memory Region 2 Data Memory Region 3 Data Memory Region 4 Data Memory Region 5 Data Memory Region 6 Data Memory Region 7 Instruction Memory Region 0 Instruction Memory Region 1 Instruction Memory Region 2 Instruction Memory Region 3 Instruction Memory Region 4 Instruction Memory Region 5 Instruction Memory Region 6 Instruction Memory Region 7	Read/Write	 Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable Base/Size/Enable

Table C-4: ARM940T

Register	Description	Access	Data
r7	Cache Operations:	Write Only	
	Flush I-Cache		SBZ
	Flush I-Cache Single Entry		Index/Segment
	Flush D-Cache		SBZ
	Flush D-Cache Single Entry		Index/Segment
	Clean D-Cache Entry		Index/Segment
	Prefetch I-Cache Line		Address
	Clean and Flush D-Cache entry		Index/Segment
r9	Lockdown control:	Read/Write	
	Data Lockdown Control		D-Ctrl Value
	Instruction Lockdown Control		I-Ctrl Value
r15	Test/Debug Register	Read/Write	Map I/D CAM Flags

Table C-4: ARM940T (Continued)

The encodings to be used to read/write the registers are as follows:

r0–r1	Data reads/writes occur as expected.
r2	Data read/write with `cp15_cache_selected` = 0 reads/writes D-Cache Bits. Data read/write with `cp15_cache_selected` = 1 reads/writes I-Cache Bits.
r3	Data reads/writes occur as expected.
r5	Data read/write with `cp15_cache_selected` = 0 reads/writes Data protection Access permissions Data read/write with `cp15_cache_selected` = 1 reads/writes Instruction protection Access permissions
r6	Data value read/written will read/write a memory area definition consisting of a base address, a size and an enable flag for the Memory area defined by the value of the `cp15_current_memory_area` variable. In addition the variable `cp15_cache_selected` selects between the I/D areas.
r7	Bits[1:0] of the data value written in conjunction with the value of `cp15_cache_selected` decide on the function accessed, as shown in ***Table C-5: ARM940T cp15 register 7 accesses*** on page C-8.

cp15_cache_selected	Bit[1]	Bit[0]	Purpose
0	0	0	Flush D-Cache
0	0	1	Flush 1 entry D-Cache
0	1	0	Clean D-Cache entry
0	1	1	Clean & Flush D-Cache Entry
1	0	0	Flush I-Cache
1	0	1	Flush 1 entry I-Cache
1	1	0	Prefetch I-Cache cache line

Table C-5: ARM940T cp15 register 7 accesses

The data value used once the function has been decoded will be Bits[31:2] of the Data value written OR'd with binary 00. So If 0x80000002 is written to r7 and `cp15_cache_selected` =1 then Instruction Data at 0x80000000 will be prefetched into the I-Cache.

r8	Data read/write with `cp15_cache_selected` = 0 reads/writes Data Lockdown control Data read/write with `cp15_cache_selected` = 1 reads/writes Instruction Lockdown control
r15	Data read/write occur as expected.

Index