

CodeScape Light

Overview

Setting up the Development System

Developing your Application

DashScript

Hardware Reference

Legal Notice

IMPORTANT

The information contained in this publication is subject to change without notice. This publication is supplied "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties or conditions of merchantability or fitness for a particular purpose. In no event shall Imagination Technologies be liable for errors contained herein or for incidental or consequential damages, including lost profits, in connection with the performance or use of this material whether based on warranty, contract, or other legal theory.

This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior permission of Imagination Technologies Limited.

CodeScape

CodeScape version 4.0

First release June 2002, Release 1.0 August 2002

CodeScape version 3.0

First release January 2001, Release 1.0 September 2001, Release 2.0 November 2001, Release 3.0 April 2002.

Copyright © 2001 - 2002 Imagination Technologies Ltd. All Rights Reserved.

CodeScape, Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective manufacturers. This publication is for information only. Any contract between Imagination Technologies and its customers will be subject to the terms and conditions of the relevant agreement. Specifications are subject to change without notice.

Imagination Technologies Ltd

23 The Calls, Leeds, West Yorkshire, LS2 7EH

telephone: +44 113 242 9814

facsimile: +44 113 242 6163

www.codescape.com

email sales: enquiry@codescape.com

email support: support@codescape.com

Contents

Overview	8
This Release	10
System requirements	12
 Setting up the Development System	 14
 Connecting the Hardware	 16
Connecting an SH2, SH3, or SH4 development system	18
Connecting an SH2e development system	20
 Configuring the Communications	 24
Configuring the communications	26
Installing CodeScape	27
Configuring the DASH's IP address	30
Installing TCP/IP	30
Assigning an IP address and installing firmware	31
Initializing the DASH for debugging	34
Initializing for 7615, 7622, 7709A, 7729R, 7729, 7727, 7750, 7750S, and 7751	34
Initializing for 7047	42
Initializing for 7055	43
Troubleshooting	44

JTAG Connector Pin Details	46
Generic SH JTAG details	48
Generic connection details for the SH2 processor	49
Generic connection details for the SH2 processor	51
Generic connection details for the SH3	53
Generic connection details for the SH4 processor	55
Supported development boards	58
Connection details for the SH3 (and SH2-7622) Solution Engine	59
Connection details for the SH4 and SH2 Solution Engine	61
Connection details for the SH3 EBX, SH3 MicroEngine and SH4 MicroEngine	62
Connection details for the SH2e EVB (SH7055)	63
 Preparing boot scripts and boot ROMs	 68
Writing a boot ROM for SH2, SH3 or SH4	70
Basic requirements of the boot ROM	70
Writing boot scripts	74
 Developing Your Application	 86
 Creating Projects and Configuring the Interface	 88
Running CodeScape	90
CodeScape Project Manager	94
Overview	95
Setting up a new project: Procedures	99
Working with an existing project: Procedures	102
Use and configure the interface	108
Commands on the menu bar	109
The commands on the toolbars	127
Commands on each toolbar	130

How windows and regions work	142
Using windows	143
Using regions	145
Configuration file: dali.cfg	152
Debugging	158
Debugging	160
Region types	164
Source and Disassembly regions	169
Call Stack region	178
Watch and Local Watch regions	179
Watch region	181
Local Watch region	184
Memory region	187
Register region	196
SuperH target processor register display	200
Edit region	206
Running and stopping programs	216
Stepping into (tracing) code	220
Breakpoints	226
Configuring breakpoints	232
Evaluating expressions	244
C/C++ expressions	247
Assembler expressions	249
Writing Scripts	252
Writing scripts	254
Scripting commands	256

General and control	258
Basic read/write	268
Breakpoints	280
Multiple Target Support	296
 LibCross Fileserver	 308
LibCross Fileserver Library	310
Set the relative path for fileserver operations	312
Fileserver functions	313
 Statistical Profiling	 332
What is profiling?	334
Statistical profiling	334
Function trace profiling	334
 Setting up the Profiler	 336
Prerequisites	336
Loading a program file	337
Setting the Profiler Setup options	338
Setting Profiler breakpoints	341
 Profiling a program and analyzing the results	 344
Running the Profiler	344
Analyzing the profile data	346
Checking for stack overflow	349
Navigating in the Profiler window	350
Saving the profile data	352
 Appendix A: Profiler's shortcut menu	 354
 Appendix B: Profiler file format	 356

DashScript	372
Before you start	374
Configuring the communications	376
Installing DashScript	377
Installing TCP/IP	378
Assigning an IP address and installing firmware	379
Initializing the DASH for debugging	382
Initializing for 7615, 7622, 7709A, 7729R, 7729, 7727, 7750, 7750S, and 7751 ..	382
Examples for debugging a new target board	385
Initializing for 7047	387
Troubleshooting	388
Scripting commands	390
General and control	395
Basic read/write	405
Breakpoints	417
Multiple Target Support	433
Channel Support	444
 Hardware Reference	 456
 SH2 Hardware Reference	 458
ASE and Extended debug stubs (SH7615, SH7622)	460
Interrupts and Exceptions (SH7615, SH7622)	464
Debug stub and exception handling (SH7047)	468

SH2e Hardware Reference	470
Equipment covered and terminology	472
The EVB7055F evaluation board (EVB)	474
SH7055F memory map	474
Operating Modes	475
Operating mode selection	478
Reserved vectors	479
The 7055F debug stub	482
Flash memory programming	486
Anomalies	490
 SH3 Hardware Reference	 492
ASE and Extended debug stubs	494
Interrupts and Exceptions	498
 SH4 Hardware Reference	 504
ASE and Extended debug stubs	506
Interrupts and Exceptions	510
 Communications Channels	 516
About channels	518
ASE BIOS services	520
Accessing ASE BIOS services	521
ASE BIOS calls	524
Servicing buffers using CHISR	531
Interrupt handler considerations	535

CPDIAL Library Reference	536
CDial functions	538
CDial::CDialChannelEx functions	551
CDial::CTypedChannelROEx functions	558
CDial::CTypedChannelEx functions	564
CDial::Console functions	566
CDial::DA functions	586
CPDIAL error codes	596

Overview

This Release

DASH Target Interface

The DASH is an intelligent Target Interface that gives complete, scriptable, control over the target. It connects to the target via JTAG and to the development computer via Ethernet using high-speed communications channels. The target can be controlled using common script languages such as Microsoft JScript and VBScript with DashScript, a supplied COM object with commands for debugging, system test, and Flash programming. The DASH's debug features and communications channels let you develop your own tools and applications with the supplied CPDIAL library.

The DASH has 8Mbytes of RAM and an SH Series RISC processor. It uses the JTAG-based Hitachi User Debug Interface (H-UDI) to communicate with the target at up to 1.5Mbytes per second. Communication between the DASH and your computer is via Ethernet at speeds of up to 500Kbytes per second.

The DASH hardware is supplied with the DashScript COM object that you can use to configure and control the DASH. You can also configure and control the DASH with CodeScape Advanced and associated software utilities.

DashScript COM object scripting

DashScript is a COM object that supports Microsoft® JScript® and VBScript macro scripts for controlling the DASH. The DashScript commands let you communicate with a DASH and connected target via a windows scripting host. You can use the commands available in either script language to add commands of your own.

The DashScript debugging commands include reading and writing memory and registers, loading and running programs, and setting breakpoints. You communicate with the application running on the target system via communication channels. The channels are supported at one end by DashScript, and at the other by the 'ASE BIOS' calls available via the debug mechanism on the target system.

For details about the Microsoft Windows Script Host, JScript, and VBScript commands, connect to: <http://msdn.microsoft.com/scripting>

CodeScape Light and CodeScape Advanced

With the DASH, CodeScape Light and CodeScape Advanced support the complete embedded systems development cycle.

CodeScape Light includes a source-level debugger with a JTAG interface to the target, together with project management and an editor.

CodeScape Advanced adds multicore source-level debugging, advanced code profiling, fully cycle-accurate processor simulation, additional debugging regions including the ability to script your own debug regions, and operating system modules to fully support the most complex and performance-critical projects.

System requirements

Your development computer and operating system requirements depend on the software you install for use with the DASH.

Development computer

An IBM™ PC or compatible with:

- Pentium® 90MHz processor minimum, and Pentium® II 233MHz processor or above recommended.
- 32MB RAM minimum, and 128MB of RAM recommended.
- 31MB of available hard disk space minimum, 58MB of available hard disk space recommended.
- An Ethernet Network Interface Card using the TCP/IP suite, and a spare, valid IP address on your network for the DASH.

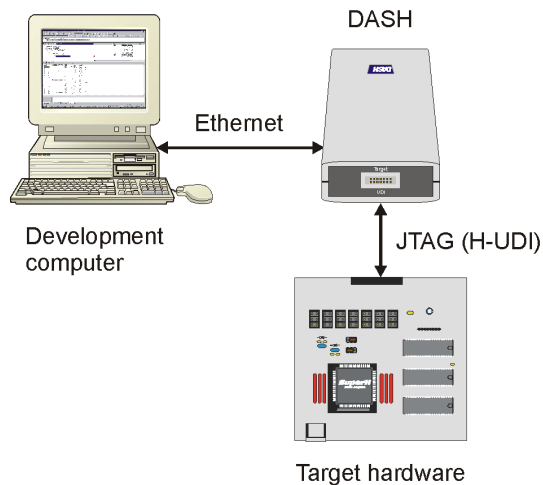
Operating system and third party software

- Microsoft Windows® 95 OSR2, 98 SE, NT 4.0 SP3 and SP6, and 2000 Professional. DashScript is tested and supported under these operating systems. There are no known issues relating to other, untested operating systems.
- A Windows scripting host. Windows Script 5.5 is required if you use VBScript. It can be downloaded from:
<http://www.microsoft.com/msdownload/vbscript/scripting.asp>

Cables and connectors

- An RJ45 8-pin shielded ethernet cable if you are connecting the DASH to a network, or a crossed RJ45 cable for connecting the DASH directly to your computer.
- A shielded ribbon cable for connecting the DASH to your target. Cables are available from Imagination Technologies for all supported Hitachi evaluation boards, *refer to “JTAG Connector Pin Details” on page 46*. Pin details are also given for each supported processor if you want make your own cable.

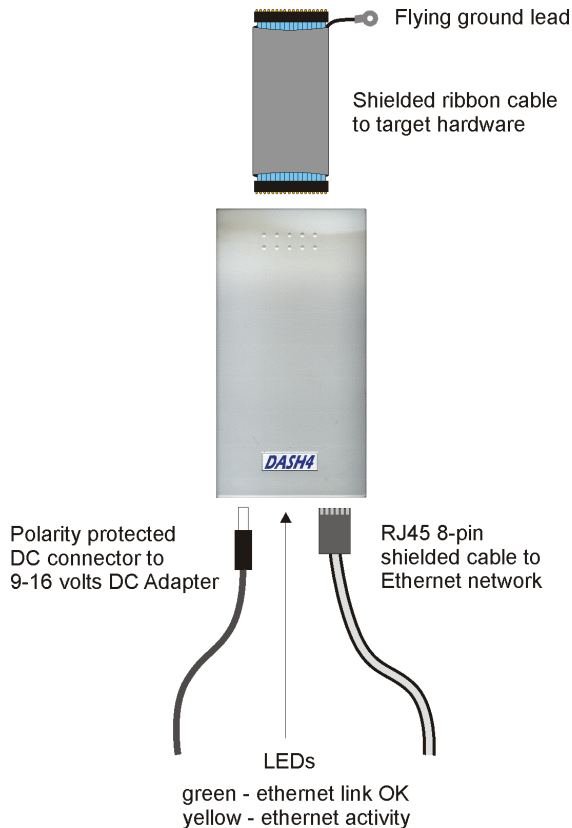
Setting up the Development System



Connecting the Hardware

Connecting an SH2, SH3, or SH4 development system

Connections to make on the DASH



Ethernet

The DASH connects to a network via an RJ45 8-pin shielded ethernet cable with a maximum length of 3m. To connect the DASH directly to your computer's network card use a crossed RJ45 cable.

Power

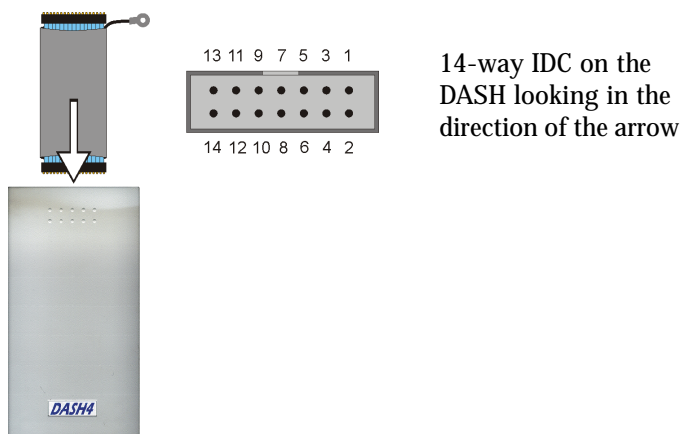
The DASH requires a 9-16 volts DC power supply of minimum 100 mA rating. The DC input to the DASH is polarity protected. A DC adapter is supplied with the DASH.

JTAG

The DASH connects to the target via a shielded 14-way IDC ribbon cable. To fully comply with EC directive 89/336/EEC concerning emissions (EN55022) and immunity (EN50082-1) you must connect the flying ground lead (if fitted) on the supplied cable to a suitable secure earthing point on your target system.

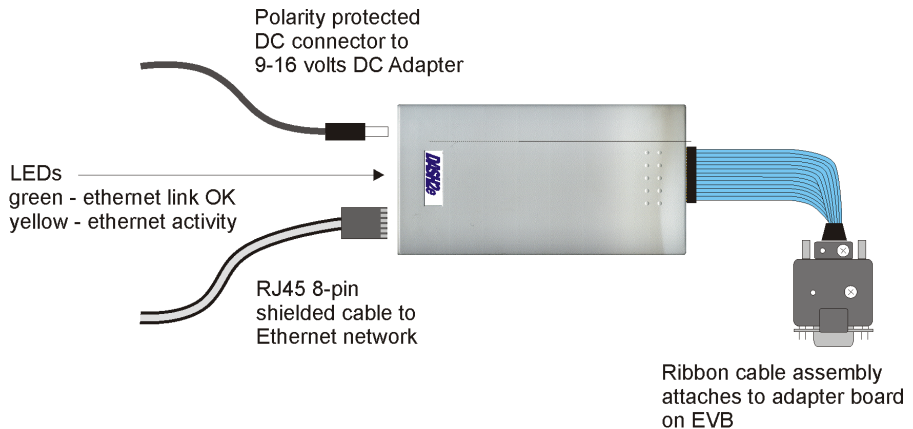
If you are making your own cable to connect to a custom installation, note that the JTAG interface has a theoretical maximum length of 300mm based on a 20MHz clock speed. As a guideline we recommend that the total cable length does not exceed 300mm (cable plus on-board track length to the processor). You may be able to achieve longer cable lengths at slower clock speeds but performance is not guaranteed. All cables must be screened and earthed as specified in this document.

Pin layout of the connector on the DASH



Connecting an SH2e development system

Connections to make on the DASH



Ethernet

The DASH connects to a network via an RJ45 8-pin shielded ethernet cable with a maximum length of 3m. To connect the DASH directly to your computer's network card use a crossed RJ45 cable.

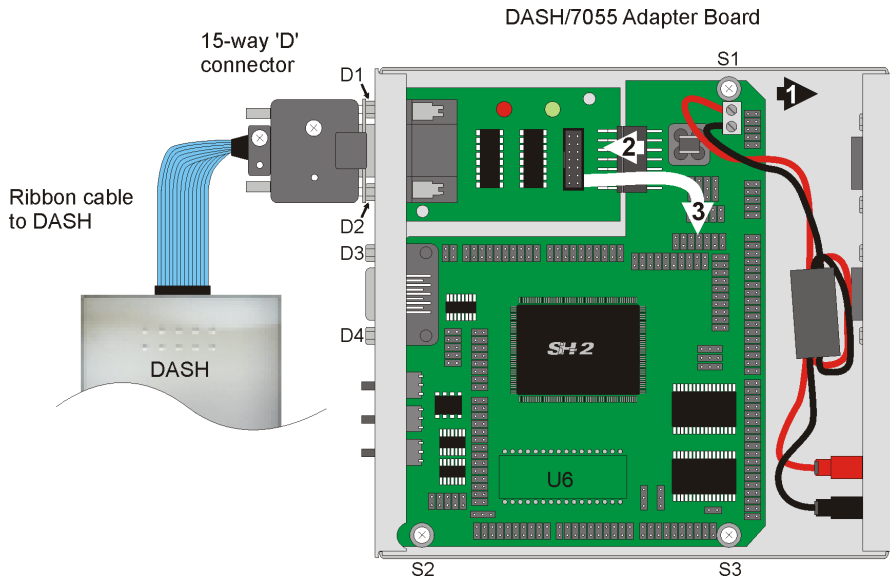
Power

The DASH requires a 9-16 volts DC power supply of minimum 100 mA rating. The DC input to the DASH is polarity protected. A DC adapter is supplied with the DASH.

Connecting the DASH to the EVB

The DASH connects to the target via a 14 way ribbon cable and the DASH/7055 Adapter Board which fits into the EVB enclosure.

Installing the DASH/7055 Adapter Board



Installing the DASH/7055 Adapter Board

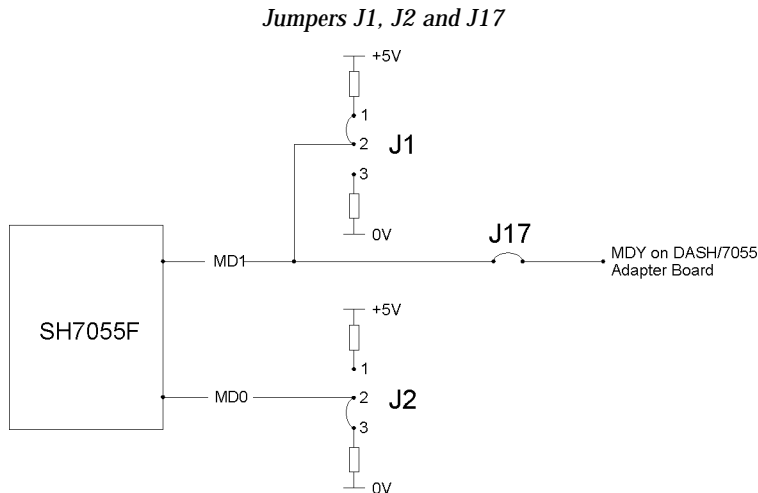
1. Before fitting the DASH/7055 Adapter Board remove the 'D' socket mounting nuts D1 and D2. *D1 and D2 are shown in position after the adapter board has been fitted.*
2. Remove the 'D' socket mounting nuts D3 and D4 and the screws S1, S2 and S3 and slide the main board in the EVB enclosure approximately 10mm in the direction of arrow 1.
3. Fit the DASH/7055 Adapter Board into the enclosure as shown in *Figure* . Locate the 15-way 'D' socket through the mounting holes in the EVB enclosure.
4. Slide the main board back into position so that it connects with the 14-way pin header on the DASH/7055 Adapter Board as shown by arrow 2.
5. Secure the DASH/7055 Adapter Board in the EVB enclosure using the mounting nuts D1 and D2 on the 15-way 'D' connector on the DASH interface board.

6. Refit the 'D' socket mounting nuts D3 and D4 and the screws S1, S2 and S3 to secure the main board in the EVB enclosure.
7. Connect the supplied 14-way IDC to IDC ribbon cable from the IDC socket on the DASH/7055 Adapter Board to the pin header on the EVB as shown by arrow 3.
8. Connect the supplied 14-way IDC to 15-way 'D' ribbon cable from the adapter board to the DASH.

Setting the jumpers on the EVB

Fit the following jumpers on the EVB before you power it up. The location of the jumpers is shown on *Figure on page 22*.

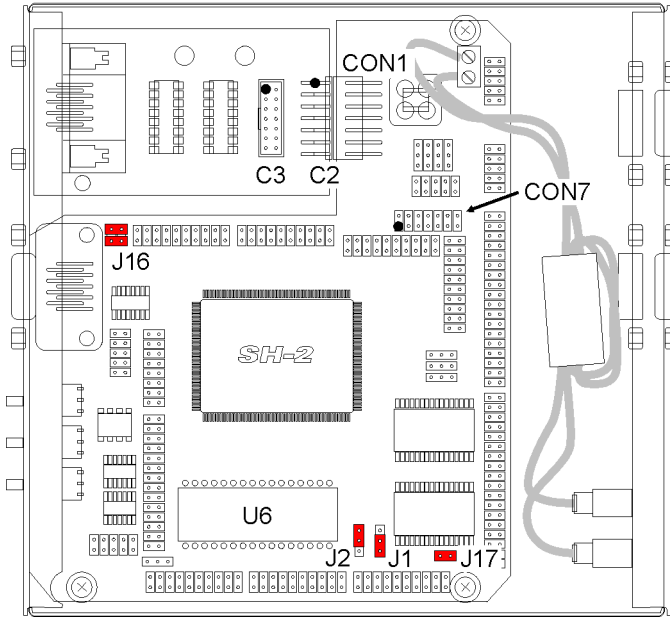
- J1 between pins 1 and 2 - Sets SH7055F mode pin MD1 to logic 1 when DASH is not connected.
- J17 - Allows MD1 to be controlled by the DASH when the DASH is connected.
- J2 between pins 2 and 3 - Sets MD0 to logic 0.



All other jumpers are application specific and do not effect the operation of the debug tools. Refer to the EVB7055F Low-cost Evaluation Board User Manual for details of all other jumper settings.

Locating the jumpers on the EVB

● Shows location of Pin 1 of C2, C3, CON1 and CON7



NOTE: *J16 changes the orientation of TxD and RxD on SCIO.
Pins 1 and 2 = 1:1 connection.
Pins 2 and 3 = crossed connection.*

Configuring the Communications

Configuring the communications

Complete the following steps to establish communications between the DASH and your computer, and the DASH and the target for first use.

1. Insert the HASP dongle (parallel or USB type) into your computer.
The dongle is unique to the product and is necessary for the software to install and run correctly. The dongle connector is covered with a label after it has been correctly programmed. If the label is missing contact technical support.
2. Install CodeScape and register for technical support. See *“Installing CodeScape” on page 27*.
3. Configure the DASH's IP address using NetFlash, and confirm that NetFlash reports the DASH is present. See *“Configuring the DASH's IP address” on page 30*.
4. Use CodeScape to initialize the target. See *“Initializing the DASH for debugging” on page 34*.

NOTE: Older dongles labelled *TimeHASP-1* will not work with versions of CodeScape after build 243. If you have one of these contact technical support for a replacement.

Installing CodeScape

1. Insert the CodeScape CD into your computer's CD drive.
2. Follow the instructions on screen.
After installation, CodeScape and its associated applications can be accessed on the Start menu in the CodeScape program group.

NOTE: *If you want to make a subsequent installation of CodeScape, we recommend that you uninstall the previous version and always do a full, new installation. If you do not do this CodeScape will not uninstall properly at a later date.*

Technical Support

Imagination Technologies offers one year free technical support with standard versions of CodeScape and two years with advanced versions.

This includes:

- Software and firmware upgrades that you can download from www.codescape.com
- Telephone: +44 113 242 9814
- Email: support@codescape.com

When contacting technical support, please provide the following:

1. Your DASH's serial number.
The serial number is shown on the base of the DASH.
2. Your Support Reference Code (SRC) and HASP dongle ID.
To view the SRC and HASP ID click Help, About CodeScape...
3. The version of CodeScape and the operating system it is running on.
4. Any error messages you see.

The Support Reference Code

A temporary SRC is generated the first time you run CodeScape using the date on your computer as the registration date. For a permanent SRC you must register the software.

To register CodeScape email support@codescape.com with:

- Your name, email address, and the company you work for.
- The product you are registering, its temporary SRC, and date it was purchased.
On receipt of this information we will send you an Access Key that you can use to generate a permanent SRC.

Generating a permanent SRC for CodeScape

- 1 Click the Start button, point to Programs, point to the CodeScape program group, and click cs-key.
- 2 Click Scan, then Register and enter the Access Key.
- 3 Click OK.
A permanent SRC is generated and you are registered for technical support.

Configuring the DASH's IP address

Installing TCP/IP

To use the DASH you must have the TCP/IP suite installed as part of your computer's network settings.

Network

If your computer is connected to a network you will probably have TCP/IP already installed and you can continue with the section *"Assigning an IP address and installing firmware"* on page 31. Ask your network administrator if you are not certain about this.

Peer-to-peer

If you are connecting peer-to-peer you do not need to know any network details but you do need a network card installed in your computer. Refer to the documentation supplied with your network card for details of how to install and configure it. Refer to Windows online help documentation for details of how to install TCP/IP on your computer and then configure the IP settings. Below are typical public values you can use for the IP address and subnet mask in a peer-to-peer connection:

- Your computer's IP address: 192.168.0.1
- Your computer's subnet mask: 255.255.255.0
- The DASH's IP address 192.168.0.2.

The IP address space 192.168.0.x is allocated for private use by the Internet Assigned Numbers Authority. In this example the last digits (1 and 2) are the device IDs assigned to your computer and the DASH respectively. You can assign any number from 1 to 254 as device IDs. Do not assign 0 or 255, these numbers are reserved.

Assigning an IP address and installing firmware

Run NetFlash to:

- Assign or change the IP address of a DASH on your network.
You can either allocate a static IP or you can use DHCP to request a dynamic address. Ask your network administrator which method you should use to assign your DASH an IP.
To use a static IP you need the subnet mask or netmask address for your network, and a spare, valid IP address on the network to assign to the DASH.
- Upgrade the firmware on a DASH target or revert to an older version.

The following versions of DASH firmware support DHCP:

- DASH4 v6.4.5 or later.
- DASH3 v5.7.5a or later.
- DASH2 v6.0.2a or later.

Notes

1. *If the DASH firmware supports DHCP, it is active by default.*
2. *If your DASH does not have firmware that supports DHCP, you can get an upgrade from www.codescape.com*
3. *When you upgrade the firmware on a DASH the state of DHCP remains as it was before you changed the firmware. To enable DHCP for a DASH after a firmware upgrade, remove the DASH from the target list then add it with DHCP enabled.*

Searching for a DASH and assigning it an IP address

1. In NetFlash click *Search/Add*.
The Search For DASH IP dialog appears.
2. In the *Hardware No* box, enter the last five digits of the DASH's serial number.
The serial number is shown on the base of the DASH.
3. Click *Search*.
If the DASH is on your network its current network settings appear in the Add Selected Target dialog.
4. Do one of the following:
 - Click OK if you want to accept the current settings.
 - Select *DHCP Enabled* then click OK.

The *IP No.* field becomes inactive and the DASH uses DHCP to request a dynamic address.

- In the *IP No.* field enter a valid spare IP address on your network then click OK.

The new target appears in the *DA ID* box and DHCP use, model identification, product serial number, firmware version and type appear in the fields below.

Changing a DASH's IP address

1. In NetFlash select a DASH from the *DA ID* box.

The current state of DHCP usage, model identification, product serial number, current firmware version, and firmware type appear in the fields below.

2. Click *Change IP*.

The Change IP Number dialog appears.

3. Do one of the following:

- Select *DHCP Enabled*.

The *IP No.* field becomes inactive and the DASH uses DHCP to request a dynamic address.

- In the *IP No.* field enter a valid spare IP address on your network.

4. Click OK.

The target's new IP address appears in the *DA ID* box next to the DASH hardware number.

Upgrading the DASH firmware

1. In NetFlash select a DASH from the DA ID box.
The current state of DHCP usage, model identification, product serial number, current firmware version, and firmware type appear in the fields below.
2. Click Reflash.
The Open dialog appears.
3. Browse for the version of firmware you want to upgrade to, select the *.fsh file and click Open.
4. Click OK.
A progress bar appears and indicates the time remaining for the firmware change. When the firmware change is complete, power cycle the DASH.

CAUTION: *Do not power cycle the DASH while the firmware is being upgraded, this may permanently damage the target.*

CAUTION: *Make sure that you use the correct version of the firmware for your target processor. Using the wrong firmware for your target may permanently damage it.*

Removing a DASH from the target list

1. In NetFlash select a DASH from the DA ID box.
The model identification, product serial number, current firmware version, and firmware type appear in the fields below.
2. Click Remove.
3. You are asked to confirm that you want to remove the selected target. Do one of the following:
 - Click Yes to remove the target and continue.
 - Click No to leave the target as it is and continue.

Initializing the DASH for debugging

Initializing for 7615, 7622, 7709A, 7729R, 7729, 7727, 7750, 7750S, and 7751

When the DASH and target are first powered up, the DASH loads a minimal ASE stub into ASE RAM onto the target to allow for basic debugging and to enable the larger Extended debug stub to be loaded. To get full debugging information the Extended debug stub must be loaded into external RAM on the target.

CodeScape lets you:

- Specify how the target's RAM initializes and load the Extended debug stub.
 - You can initialize the target's RAM with a boot ROM or by writing a script to program the memory mapped registers directly.
 - You can load the Extended debug stub as a result of a BIOS call request from the boot ROM to an address specified in the boot ROM, or you can force it to load a to specified address.
- Set the JTAG clock frequency.
- Specify the reset method. (SH4 processors only.)

For information about the ASE and Extended debug stubs to the “Hardware Reference” section in the online documentation.

For more information on writing boot ROMs and initialization scripts see “Preparing boot scripts and boot ROMs” on page 68.

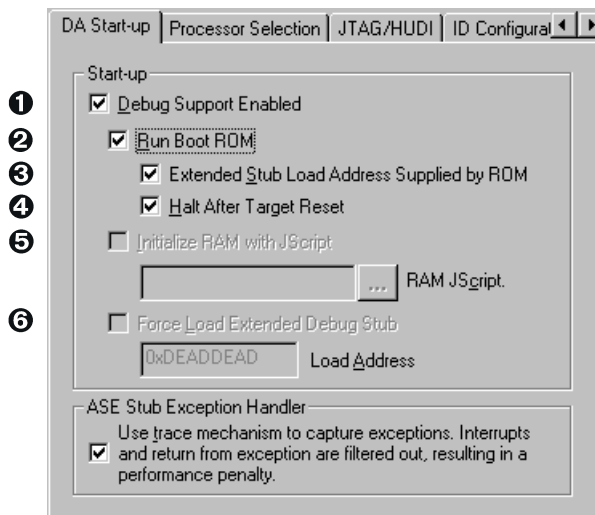
Configuring the target and loading the Extended debug stub

Set up the initialization options on the Target Set-up dialog.

1. Click Tools, select Target/Communications then click Configure...
2. Select the DA Start-up tab.

The flowchart shows how the DA Start-up options affect how the DASH initializes the target and loads the Extended debug stub. See “*Start-up options flowchart*” on page 36.

DA Start-up tab, numbers show corresponding options on the flowchart



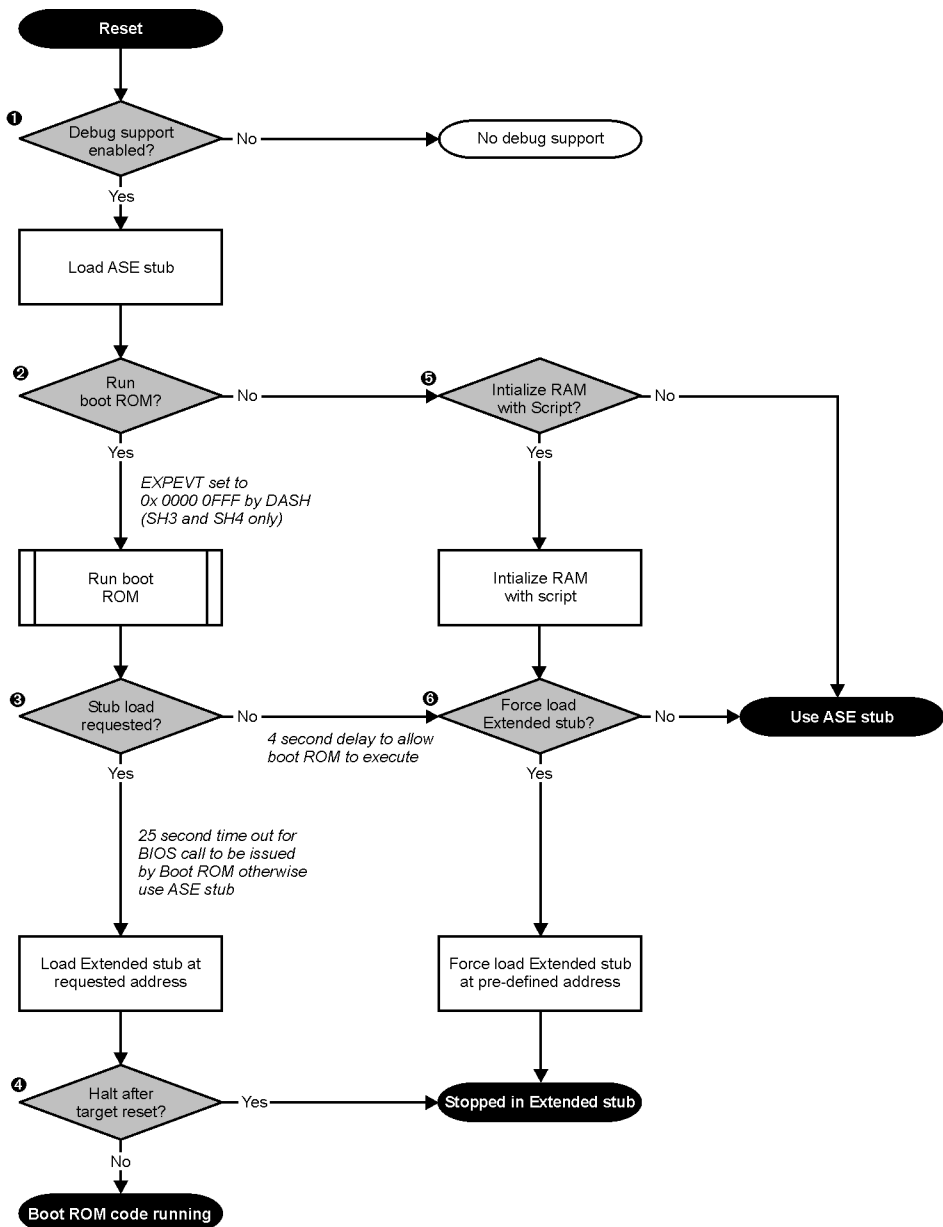
3. Check *Debug Support Enabled*.

If it is not checked the target will run without intervention from the debug tools and there will be no debug information available in CodeScape.

For more information on testing your application without debug support refer to “Debugging” in the online documentation.

Debug support always defaults to *Enabled* when the DASH is powered up so that the DASH can see the target.

Start-up options flowchart



Initializing RAM with a boot ROM

For more information see “Preparing boot scripts and boot ROMs” on page 68.

For the SH3 and SH4 the boot ROM is run from address 0xA0000000, the default reset address, after the ASE debug stub has loaded. The SH2 boots from a vector at 0xA0000000.

1. On the DA Start-up tab, select *Run Boot ROM*.
2. Do one of the following:
 - Select *Stub Load Address Supplied by ROM* if your boot ROM includes a BIOS call to load the Extended debug stub. The DASH pauses for 25 seconds to allow the BIOS call to be issued by the boot ROM, if the BIOS call is not received only the ASE debug stub is available.
 - OR--OR-
 - Select *Force Load Extended Stub* and specify where in the target’s RAM to load the Extended debug stub.
The address must be on a long word boundary. If you do not set a load address only the ASE debug stub is available.
3. Select *Halt After Target Reset* to stop your application code after start up.
Alternatively, leave this unchecked to return program control to the boot ROM after start up.

Initializing RAM with a script

Example scripts which you can use as a basis to write a script for your specific hardware are provided with the CodeScape installation.

1. On the DA Start-up tab, select *Initialize RAM with JScript*.
Browse for the script you want to run.
2. Select *Force Load Extended Stub* and specify where in the target’s RAM to load the Extended debug stub.
The address must be on a long word boundary. If you do not set a load address only the ASE debug stub is available.

CAUTION: Do not use the *WriteRegister* command in a RAM initialization script or unexpected results may occur.

Setting the JTAG clock frequency

The JTAG clock frequency (TCK) is preset to 20MHz which is suitable for most SH2, SH3, and SH4 applications.

Set the JTAG clock frequency on the JTAG/HUDI tab in the Target Set-up dialog so that it matches the requirements of the target hardware you are debugging.

The following general rules apply:

- SH4: the JTAG clock frequency must be less than the target hardware's peripheral clock frequency.
- SH2 and SH3: the JTAG clock frequency must be less than half the target hardware's CPU clock frequency.

NOTE: Refer to Hitachi documentation to determine the correct JTAG clock frequency to use for your target hardware.

NOTE: The debug tools requires that the JTAG clock remains stable during use, do not attempt to single step over an instruction that changes the contents of the Frequency Control Register, FRQCR.

Specifying the reset method (SH4 only)

If you are configuring an SH4 you can specify how the DASH generates a target reset on the JTAG/HUDI tab. *Pin Reset* is preferable to *UDI Reset* because it enables the entire target hardware to be reset.

- Pin Reset uses the DBG_RST# line.
For this to operate the target system must have a functional DBG_RST# and RESETIN# line available to the DASH. The DASH uses the RESETIN# line as confirmation that the DBG_RST# line assertion occurred correctly.
- UDI Reset can be used if there is no hard-wired reset mechanism on the target hardware.
With this selected, on a hard reset, the DASH sends a serial command on the UDI TDO line to reset the target.

ASE Stub Exception Handler (SH4 only)

The ASE debug stub uses the SH4's Branch Trace mechanism to catch exceptions and interrupts. If the ASE debug stub is being used to debug an application that uses interrupt VBR + 0x600 and its associated RTE, there will be a time penalty due to the filtering effect of processing this unwanted exception. If you do not want the ASE stub to catch any exceptions you can turn off the Branch Trace mechanism using the check box on the DA Start-up tab.

For full details refer to “Debugging exception handlers using CodeScape” on page 513.

Examples for debugging a new target board

These examples demonstrate how to set up the DA Start-up options depending on your debugging environment. All examples assume the target is fitted with RAM.

Example 1 - Test hardware with no boot ROM fitted.

This configuration allows basic debugging using only the ASE debug stub. In this mode you can verify that the target processor is running. External hardware can be read from, and written to, so that addressing of peripherals can be checked.

Debug Support Enabled	Checked
Run Boot ROM	Unchecked
Stub Load Address Supplied by ROM	Unchecked
Halt After Target Reset	Not applicable
Initialize RAM with JScript	Unchecked
Force Load Extended Stub	Unchecked

Example 2 - Test hardware with no boot ROM fitted; RAM initialized with a script.

This configuration allows basic debugging using only the ASE debug stub. With the addition of a script file the RAM and any peripherals can be initialized. At this point test programs can be loaded into RAM and executed.

Debug Support Enabled	Checked
Run Boot ROM	Unchecked
Stub Load Address Supplied by ROM	Unchecked
Halt After Target Reset	Not applicable
Initialize RAM with JScript	Checked
Force Load Extended Stub	Unchecked

Example 3 - Test hardware with a boot ROM fitted; RAM initialized using a boot ROM containing a BIOS call.

This configuration provides full debug support using the Extended debug stub. The Extended debug stub is loaded as a result of a BIOS call request from the boot ROM.

Debug Support Enabled	Checked
Run Boot ROM	Checked
Stub Load Address Supplied by ROM	Checked
Halt After Target Reset	Checked
Initialize RAM with JScript	Unchecked
Force Load Extended Stub	Unchecked

Example 4 - Test hardware with a boot ROM fitted; RAM initialized using a boot ROM that does not contain a BIOS call.

This configuration provides full debug support using the Extended debug stub. However, the Extended debug stub is forced to load at a specified address.

Debug Support Enabled	Checked
Run Boot ROM	Checked
Stub Load Address Supplied by ROM	Unchecked
Halt After Target Reset	Checked
Initialize RAM with JScript	Unchecked
Force Load Extended Stub	Checked

Initializing for 7047

The 7047 has a single debug stub that resides in ASE RAM on the target processor. This is loaded each time the DASH resets the target.

CodeScape lets you:

- Initialize the vector table.
- Initialize the target's RAM (optional).

Initializing the 7047 vector table

On reset the DASH reads the first 1kbyte of the 7047's application flash from 0x00000000, if its contents are FFFF (new unused target board) then CodeScape offers to write a valid vector table to flash from 0x00000000 to be used by the debug stub.

If the first 1kbytes of flash contain anything other than FFFF, the DASH assumes the contents are valid and does not write the vector table. The vector table can be reprogrammed at any time using the Reflash tab on the Target Setup dialog.

Initializing RAM with a script

Example scripts which you can use as a basis to write a script for your specific hardware are provided with the CodeScape installation.

On the DA Start-up tab:

1. Select *Initialize RAM with JScript*.
2. Browse for the script you want to run.

CAUTION: *Do not use the WriteRegister command in a RAM initialization script or unexpected results may occur.*

Initializing for 7055

The 7055 has a single debug stub that resides in application flash on the target processor. When the CodeScape starts up with a new 7055 target, it detects that there is no debug stub present and requests confirmation to load the debug stub onto the target.

Troubleshooting

DASH or target not found

If CodeScape cannot find the DASH or target check that the:

- DASH serial number in the ID Configure dialog is correct.
- Network cabling between your computer, the DASH, and the target is OK.
- Green *LNK* LED (ethernet link OK) is illuminated on the DASH.
 - *LNK* flashes twice while the DASH is searching for an IP address, it will flash indefinitely until it finds an IP address.
 - *LNK* flashes three times if the DASH finds its IP address duplicated elsewhere on the network. If this happens run NetFlash and reassign the DASH's IP address or enable DHCP.
- Yellow *T/R* LED (ethernet data) flashes occasionally.
- DASH is not in use by CodeScape on another computer on the network.

CodeScape should always start whether you have a DASH connected or not. You can get detailed communications diagnostics information on the Diagnostics tab in the Target Setup dialog.

Extended debug stub does not load

When *Stub Load Address Supplied by ROM* is selected on the DA Start-up tab the DASH expects the boot ROM to issue a BIOS call to load the Extended debug stub. If the BIOS call is not received within 25 seconds the reset times out and only the ASE stub is available.

Cannot access target's memory

If CodeScape cannot access any target memory you see the message “*No memory range for processor in config files*” when CodeScape starts up. If CodeScape cannot access areas of target memory, you see asterisks where there should be data in the Memory window in CodeScape.

Memory ranges and processor timing information are specified in the configuration file *dali.cfg*. If you cannot see or access the specific areas of memory you can edit *dali.cfg* to configure CodeScape with the areas of memory you want to access.

For more information refer to “Configuration file: dali.cfg” in the online documentation.

Before you edit dali.cfg, always make a copy of the original file.

JTAG Connector Pin Details

Generic SH JTAG details

The following pages provide generic pin details for supported SH processors so you can connect the DASH to custom target hardware.

Generic connection details for the SH2 processor (SH7615)

IDC socket pin number on DASH	Signal name at DASH	Direction relative to DASH	Signal description	Connection to make on SH2
1	TCK	output	UDI serial clock. Synchronises TDI and TDO.	TCK
2	GND	n/a	Signal ground.	GND
3	TRST#	output	UDI reset.	TRST
4	GND	n/a	Signal ground.	GND
5	TDI	input	UDI data in.	TDO
6	GND	n/a	Signal ground.	GND
7			no connection	
8	RESETIN#	input	Target reset monitor.	RESETP# (or set to logic 1, see note 1)
9	TMS	output	UDI mode select. Changes the meaning of TDI data.	TMS
10	GND	n/a	Signal ground.	GND
11	TDO	output	UDI data out.	TDI
12	GND	n/a	Signal ground.	GND
13	DBG_RST#	output	Power on reset.	RESETP# (see note 2)
14	GND	n/a	Signal ground.	GND

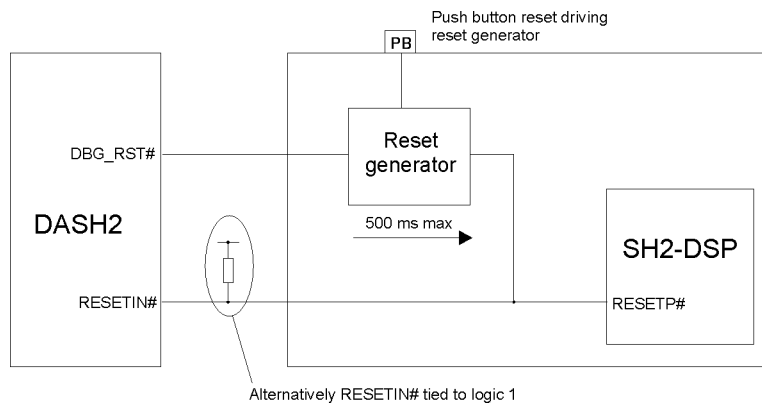
Application notes for SH2 SH7615

- 1. RESETIN# monitors the target system's on-board reset line. This allows the DASH to detect that a reset has occurred (initiated either from the DASH or elsewhere via a reset generator) so it can intervene and perform the appropriate debug reset sequence. If the line is low, the DASH assumes the target is in reset mode and waits indefinitely*

until the line goes high before taking any further action. Alternatively *RESETIN#* can be connected to *Vcc* on the target board to detect power cycles or tied to logic 1, see Figure .

2. *DBG_RST#* can be connected to *RESETP* on the SH2 directly or used to trigger a reset generator. If using a reset generator make sure that the *RESETP* signal to the SH2 is not extended by more than 500ms of the *DBG_RST#* signal from the DASH. A typical extended time is 200ms.

Monitoring the SH2 reset line



3. Make sure the SH2's *ASEMD0#* pin is logic 0. This is required to put the SH2 into ASE Mode. A good way to do this is to connect it to a Signal Ground on the on-board IDC socket which receives the DASH so that when the DASH is connected this pin is grounded.

Generic connection details for the SH2 processor (SH7047F)

IDC socket pin number on DASH	Signal name at DASH	Direction relative to DASH	Signal description	Connection to make on SH2
1	TCK	output	UDI serial clock. Synchronises TDI and TDO.	TCK
2	GND	n/a	Signal ground.	GND
3	TRST#	output	UDI reset.	TRST
4	GND	n/a	Signal ground.	GND
5	TDI	input	UDI data in.	TDO
6	GND	n/a	Signal ground.	GND
7	ASEBRK#	input	Indicates target is in ASE mode.	ASEBRKAK
8	RESETIN#	input	Target reset monitor.	RESETP# (or set to logic 1, see note 1)
9	TMS	output	UDI mode select. Changes the meaning of TDI data.	TMS
10	GND	n/a	Signal ground.	GND
11	TDO	output	UDI data out.	TDI
12	GND	n/a	Signal ground.	GND
13	DBG_RST#	output	Power on reset. Buffer to +5V (see note 6).	RESETP# (see note 2)
14	GND	n/a	Signal ground.	GND

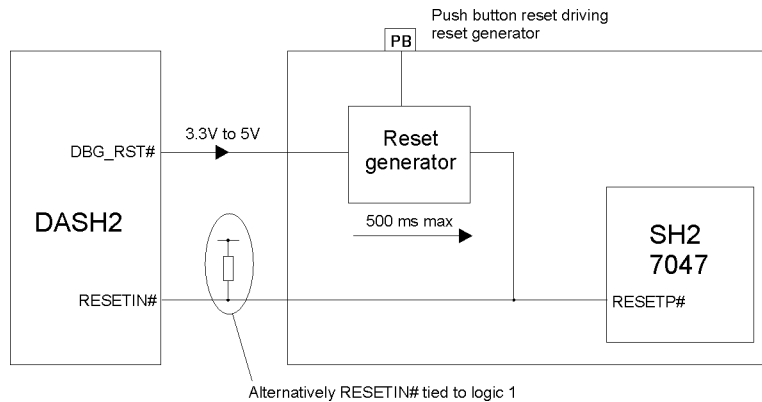
Application notes for SH2 SH7407F

1. *RESETIN# monitors the target system's on-board reset line. This allows the DASH to detect that a reset has occurred (initiated either from the DASH or elsewhere via a reset generator) so it can intervene and perform the appropriate debug reset sequence. If the line is low, the DASH assumes the target is in reset mode and waits indefinitely*

until the line goes high before taking any further action. Alternatively *RESETIN#* can be connected to *Vcc* on the target board to detect power cycles or tied to logic 1, see Figure .

2. *DBG_RST#* can be connected to *RESETP* on the SH2 directly or used to trigger a reset generator. If using a reset generator make sure that the *RESETP* signal to the SH2 is not extended by more than 500ms of the *DBG_RST#* signal from the DASH. A typical extended time is 200ms.

Monitoring the SH2 reset line



3. Make sure the SH7047F's *DBGMD#* pin is logic 0. This is required to put the SH2 into ASE Mode. A good way to do this is to connect it to a Signal Ground on the on-board IDC socket which receives the DASH so that when the DASH is connected this pin is grounded.
4. *FWP* must be pulled low on the SH7047F evaluation board.
5. *MD1* must be pulled to +5V on the SH7047F evaluation board.
6. The DASH outputs drive to 3.3V levels and can be configured as open drain. On the SH7047F the *EXT_RESET* signal requires a 5V input.

Generic connection details for the SH3 (SH7709a/7729/7729R/7727) and SH2 (7622)

IDC socket pin number on DASH	Signal name at DASH	Direction relative to DASH	Signal description	Connection to make on SH3
1	TCK	output	UDI serial clock. Synchronises TDI and TDO.	TCK
2	GND	n/a	Signal ground.	GND
3	TRST#	output	UDI reset.	TRST
4	GND	n/a	Signal ground.	GND
5	TDI	input	UDI data in.	TDO
6	GND	n/a	Signal ground.	GND
7	ASEBRK#	input	Indicates target is in ASE mode.	ASEBRKAK
8	RESETIN#	input	Target reset monitor.	RESETM (or set to logic 1, see note 1)
9	TMS	output	UDI mode select. Changes the meaning of TDI data.	TMS
10	GND	n/a	Signal ground.	GND
11	TDO	output	UDI data out.	TDI
12	GND	n/a	Signal ground.	GND
13	DBG_RST#	output	Power on reset.	RESETP# (see Note 2)
14	GND	n/a	Signal ground.	GND

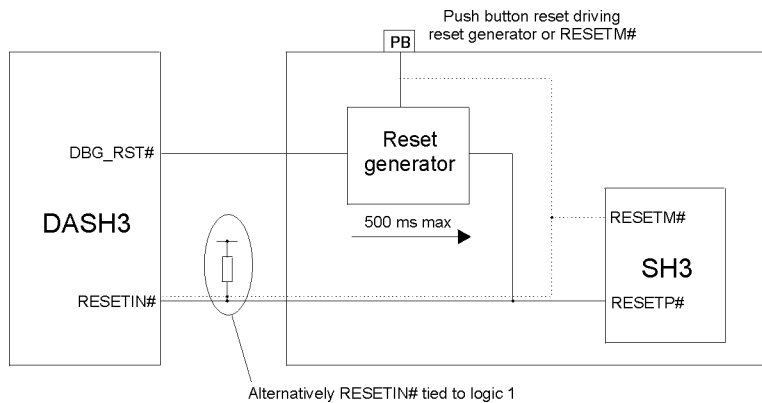
Notes for SH7709a/7729/7729R/7727

1. *RESETIN# monitors the target system's on-board reset line. This allows the DASH to detect that a reset has occurred (initiated either from the DASH or elsewhere via a reset generator) so it can intervene and perform the appropriate debug reset sequence. If the line is low, the DASH assumes the target is in reset mode and waits indefinitely*

until the line goes high before taking any further action. Alternatively *RESETIN#* can be connected to *Vcc* on the target board to detect power cycles or tied to logic 1, see Figure on page 54.

2. *DBG_RST#* can be connected to *RESETP* on the SH3 directly or used to trigger a reset generator. If using a reset generator make sure that the *RESETP* signal to the SH3 is not extended by more than 500ms of the *DBG_RST#* signal from the DASH. A typical extended time is 200ms.

Monitoring the SH3 reset line



3. Make sure the SH3's *ASEMD0#* pin is logic 0. This is required to put the SH3 into ASE Mode. A good way to do this is to connect it to a spare Signal Ground on the on-board IDC socket which receives the DASH so that when the DASH is connected this pin is grounded.
4. The SH3's *BREQ* signal (pin 122) must have a 10K pull up to 3.3 volts if it is not being used. It is not possible to perform external memory accesses on the SH3 if this pin is unconnected.

Notes for SH7622

1. The SH7622 is an SH2 processor but uses the same JTAG configuration as other SH3 processors. All the above notes also apply to the SH7622.

Generic connection details for the SH4 processor (SH7750/7751)

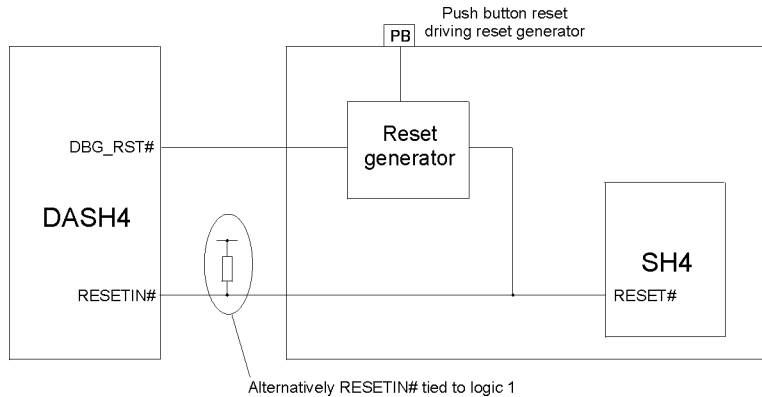
IDC socket pin number on DASH	Signal name at DASH	Direction relative to DASH	Signal description	Connection to make on SH4
1	TCK	output	UDI serial clock. Synchronises TDI and TDO.	TCK
2	GND	n/a	Signal ground.	GND
3	TRST#	output	UDI reset.	TRST#
4	GND	n/a	Signal ground.	GND
5	TDI	input	UDI data in.	TDO
6	GND	n/a	Signal ground.	GND
7	ASEBRK#	output	SH4 ASEBRK#. Stops the target and puts it into ASE mode.	ASEBRK#
8	RESETIN#	input	Target reset monitor.	RESET# (see Note 1)
9	TMS	output	UDI mode select. Changes the meaning of TDI data.	TMS
10	GND	n/a	Signal ground.	GND
11	TDO	output	UDI data out.	TDI
12	GND	n/a	Signal ground.	GND
13	DBG_RST#	output	Target reset.	Reset generator on evaluation board (see Note 2)
14	GND	n/a	Signal ground.	GND

Notes for SH4 SH7750/7751

1. *RESETIN# monitors the target system's on-board reset line. This allows the DASH to detect a reset has occurred (initiated either from the DASH or elsewhere via a reset generator) so it can intervene and perform the appropriate debug reset sequence. If the line is low, the DASH assumes the target is in reset mode and waits indefinitely*

until the line goes high before taking any further action. RESETIN# can be directly connected to the SH4 RESET# pin. Alternatively, it can be connected to Vcc on the target board to detect power cycles or tied to logic 1, see Figure on page 56.

Monitoring the SH4 reset line



2. *DBG_RST# allows the DASH to generate a reset to the target system. It should be connected to an active-low input to the target's reset generator. If no suitable way can be found to connect this then the DASH can be configured to generate a reset via the JTAG interface, refer to the Getting Started guide for details. This is less desirable because peripheral devices are not reset using this method.*

Supported development boards

The table below shows which evaluation boards are supported and which cable to use for each. Part numbers are given in the table and cables can be ordered from sales@codescape.com

	SH4 SH7750 SH7751	SH3 SH7709a SH7729, SH7729R SH7727	SH2e SH7055F	SH2 SH7615 SH7622
Solution Engine	CA09939-4SE page 61	CA09939-3 page 59		CA09939-4SE (SH7615) page 61 CA09939-3 (SH7622) page 59
EBX		CA09939-EBX page 62		
Micro Engine	CA09939-EBX page 62	CA09939-EBXs page 62		
EVB			AK09939-2e page 63	

WARNING: *Always use the correct version of the DASH to connect to your target. For example, never connect a DASH3 to an SH4 target. This can cause permanent damage to the DASH, the processor, or both.*

Connection details for the SH3 (and SH2-7622) Solution Engine

Cable part number: CA09939-3 rev D, 14-way IDC to 36 way bellows (male).

IDC socket pin number on DASH	Signal name at DASH	Signal name at SH3	Pin number on 36 way socket on Solution Engine
1	TCK	TCK	9
2	GND	GND	27
3	TRST#	TRST	11
4	GND	GND	29
5	TDI	TDO	13
6	GND	GND	31
7	ASEBRK#	ASEBRKAK	14
8	RESETIN#	RESETM (or set to logic 1)	16
9	TMS	TMS	10
10	GND	GND	28
11	TDO	TDI	12
12	GND	GND	30
13	DBG_RST#	Reset generator on evaluation board	Flying lead to test point on Solution Engine. See notes 1 and 2.
14	GND	GND	34

Notes

1. For SH7709a and SH7729 connect the flying lead to TP9 on Solution Engine.
2. For SH7727 connect the flying lead to TP5 on the Solution Engine.
3. On SH7277 Solution Engine SW1/8 (ASEMD) must be set to on.

4. *On SH7729R Solution Engine J7 must be fitted.*
5. *For SH7622 connect the flying lead to TP7 on the Solution Engine.*
6. *On SH7622 Solution Engine SW3/6 (ASEMD) must be set to on.*

Connection details for the SH4 and SH2 Solution Engine

A cable part number: CA09939-4SE rev A, 14-way IDC to 14-way IDC.

IDC socket pin number on DASH	Signal name at DASH	Connection to make on SH4	IDC pin number on Solution Engine
1	TCK	TCK	1
2	GND	GND	2
3	TRST#	TRST#	3
4	GND	GND	4
5	TDI	TDO	5
6	GND	GND	6
7	ASEBRK#	ASEBRK#	7
8	RESETIN#	RESET#	13
9	TMS	TMS	9
10	GND	GND	10
11	TDO	TDI	11
12	GND	GND	12
13	DBG_RST#	Reset generator on evaluation board	Flying lead with test clip to TP9 on Solution Engine board
14	GND	GND	14

Notes

Connect the flying ground lead to a suitable secure earthing point on the Solution Engine.

Connection details for the SH3 EBX, SH3 MicroEngine and SH4 MicroEngine

Cable part number: CA09939-EBX rev A and CA09939-EBXS, 14-way IDC to 26-way IDC.

IDC socket pin number on DASH	Signal name at DASH	Signal name at evaluation board	Evaluation board connector pin number
1	TCK	TCK	16
2	GND	GND	25/26
3	TRST#	TRST	20
4	GND	GND	25/26
5	TDI	TDO	13
6	GND	GND	25/26
7	ASEBRK#	ASEBRKAK	23
8	RESETIN#	+5v	10
9	TMS	TMS	17
10	GND	GND	1
11	TDO	TDI	15
12	GND	GND	1
13	DBG_RST#	Reset generator on target	12
14	GND	GND	1

Notes:

1. For the SH3 MicroEngine use the short version of the cable, part number CA09939-EBXS. Alternatively, you can use the standard length EBX cable but run the JTAG at 10MHz or less. Refer to *Getting Started for DASH3 and DASH4* for details of how to configure the JTAG frequency.
2. An example start-up script for the SH3 MicroEngine, *bootsh3ue.js*, is provided on the CodeScape CD.

Connection details for the SH2e EVB (SH7055)

All communications between the DASH and the SH7055F go via the DASH/7055 Adapter Board. This board routes serial and HUDI data.

The adapter board provides a switch to change the function of RESET_IN on the DASH depending on the mode of the EVB: boot mode or normal CodeScape debug mode.

See the “DASH/7055 Adapter Board schematic” on page 64.

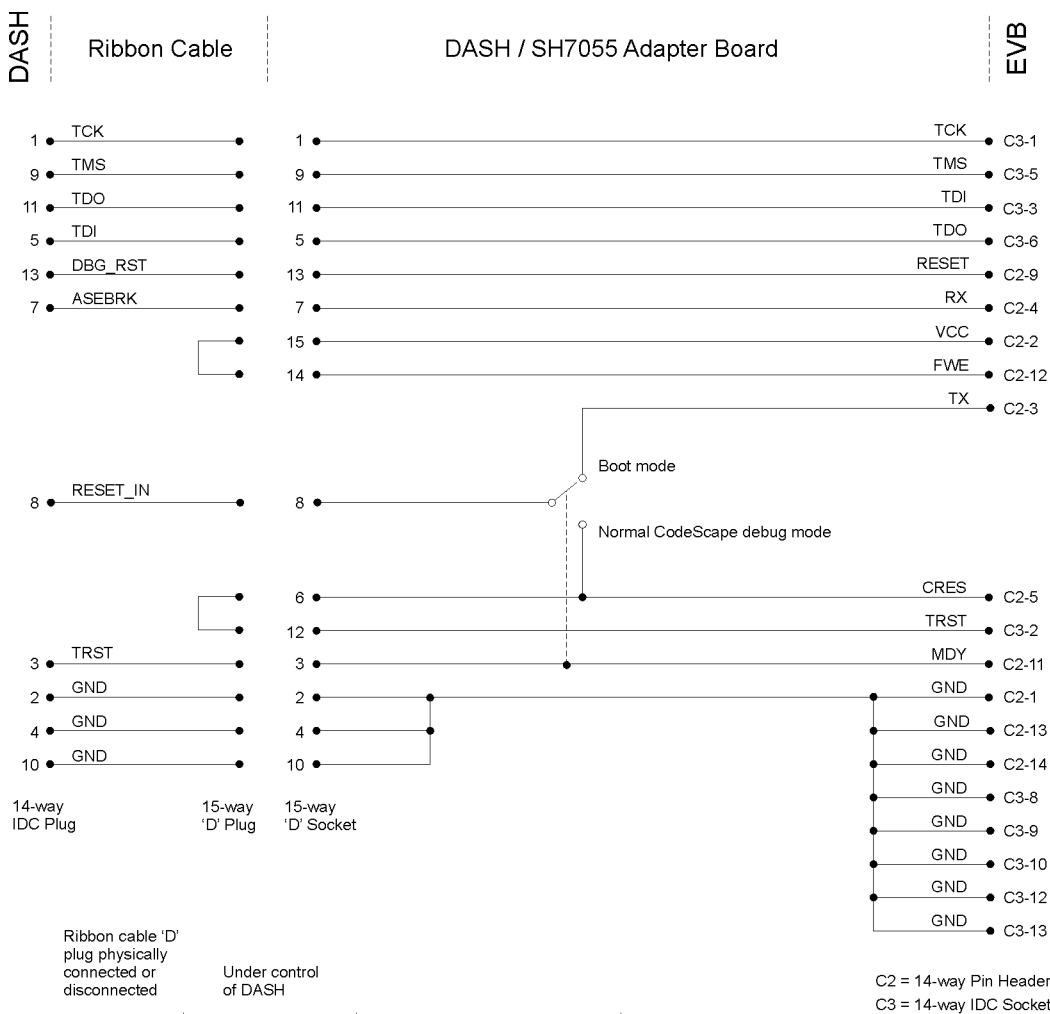
Boot mode

During boot mode, the SH7055F's mode 1 pin (MD1) is set to logic 0 under control of the MDY line from the DASH. MDY also controls the switch on the DASH/7055 Adapter Board and connects TX on the SH7055F to RESET_IN on the DASH during boot mode. The DASH then configures its RESET_IN to become an RX input ready to receive serial data. This allows application code or the debug stub to be programmed into SH7055F flash via serial lines TX and RX.

CodeScape debug mode

During normal debug mode MD1 is set to logic 1 and RESET_IN connects to CRES on the SH7055F. This gives confirmation of a push button reset on the EVB.

DASH/7055 Adapter Board schematic



DASH/7055 Adapter Board pin details

CONN1

Pin number on CON1 on EVB and C2 on adapter board	Signal name	Direction relative to EVB	Signal description
1	not used		
2	VCC	n/a	+5V supply for DASH interface board
3	TX	output	Output data from evaluation board. Only used during boot mode for FLASH reprogramming (mode 1 pin set to logic 1).
4	RX	input	Input data to evaluation board. Only used during boot mode for FLASH reprogramming (mode 1 pin set to logic 1).
5	CRES	output	Cold reset.
6	not used		
7	not used		
8	not used		
9	RESET	input	Reset (active low) to SH7055F.
10	not used		
11	MDY	input	MD1 pin of SH7055F.
12	FWE	input	FLASH write enable.
13	GND	n/a	Signal ground.
14	GND	n/a	Signal ground.

CON7)

Pin number on CON7 on EVB and C3 on adapter board	Signal name	Direction relative to EVB	Signal description
1	TCK	input	UDI serial clock. Synchronises TDI and TDO.
2	TRST#	input	UDI reset.
3	TDI	input	UDI data in.
4	not used		
5	TMS	input	UDI mode select. Changes the meaning of TDI data.
6	TDO	output	UDI data out.
7	not used		
8	GND	n/a	Signal ground.
9	GND	n/a	Signal ground.
10	GND	n/a	Signal ground.
11	not used		
12	GND	n/a	Signal ground.
13	GND	n/a	Signal ground.
14	not used		

Preparing boot scripts and boot ROMs

Writing a boot ROM for SH2, SH3 or SH4

Basic requirements of the boot ROM

Optionally the debugging system may be run with a boot ROM on the target, allowing a faster reset than can be achieved using scripts to set up the RAM and peripherals. Also, the start-up code may be a program under development intended to run when the DASH is not connected. As far as the debugging system is concerned, such a boot ROM has the basic requirements described in the following sections.

There several example boot ROMs provided in the CodeScape installation in the *Example Boot ROMs* directory.

Full details of how to implement the options for loading the debug stub are given in “*Initializing the DASH for debugging*” on page 34. Options for loading the debug stub and running a Boot ROM are set up in CodeScape and stored in EEPROM. If you are using the DASH stand-alone you must set these options using CodeScape before resetting the DASH and the target.

To recognise the special reset exception code (SH3 and SH4 only)

When the target is reset and the DASH performs an ASE debug stub load, the exception event register EXPEVT is loaded with a value so that the boot ROM can recognise the source of the reset. The values are shown in the following table:

Type of reset	Execution address	contents of EXPEVT
Target power-on reset	0xA000 0000	0x0000 0000
Target push-button reset (manual reset)	0xA000 0000	0x0000 0020

Type of reset	Execution address	contents of EXPEVT
DASH debug reset	0xA000 0000	0x0000 0FFF

When the DASH causes the reset, everything apart from EXPEVT is reset as for a power-on reset.

The boot ROM should examine the contents of EXPEVT to determine whether the DASH is connected and active. If the contents of EXPEVT are equal to 0x0000 0FFF at reset, the boot ROM should issue a BIOS call to load the Extended debug stub. If the contents of EXPEVT are not equal to 0x0000 0FFF at reset, then no further debugging operations should be done, in particular, the BRK instruction should not be used.

To instruct the DASH where to load the Extended debug stub

NOTE: *The examples given here are for SH2 and SH4 processors. For SH3 processors substitute registers R4, R5, R6, R7, R0 with R10, R11, R12, R13, R8 respectively.*

After a reset the boot ROM should configure RAM and then request the DASH to load the Extended debug stub.

The Extended debug stub requires 16K of RAM and must be loaded using an address in the P2 address space on SH3 and SH4. This address is important as it will be used later for ASE BIOS calls. We will call this address ASE_BIOS.

For SH3 and SH4 the address ASE_BIOS must be in the range 0xA000 0000 to 0xBFFF C000, i.e. the start of a 16K area entirely in the P2 address space. For SH2 the address ASE_BIOS must be in the range 0x0000 0000 to 0xBFFF C000.

The address must also be on a long word boundary (the two least significant bits must be zero).

The steps are as follows:

1. Load the R4 register with zero.
2. Load the register R5 with the address of a six-byte location CHECK_LOC to hold version information.
3. Load the register R6 with the address to load the debug stub (ASE_BIOS).
4. Execute the BRK instruction.
5. Examine the R0 register and check that it contains 2429814 (decimal).
6. Save the data returned at CHECK_LOC for later use if required.

If the version information is not required, then 5R5 should be zero before the BRK instruction is executed. Otherwise, data is returned as follows:

CHECK_LOC	Debug Stub ASCII version number.
CHECK_LOC+1	Debug Stub ASCII revision number (2 bytes).
CHECK_LOC+3	Debug Stub ASCII revision letter.
CHECK_LOC+4	Checksum (binary addition of debug stub) (word), should be 0x0000 for valid stub.

When the six steps are complete, the Extended debug stub is in memory and ASE BIOS services are ready to use. See “ASE BIOS services” on page 520.

Writing boot scripts

You can use scripts to set up RAM and peripherals in the target system. Several example boot scripts are provided in the CodeScape installation in the *Example Initialization Scripts* directory.

Full details of how to load a boot script on start up are given in “*Initializing the DASH for debugging*” on page 34.

The basic read/write CodeScape scripting commands are available for writing boot scripts. These are described in the following sections.

ReadByte

Syntax

```
ReadByte(Numeric address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a byte (8 bits) from a specified target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

ReadWord

Syntax

```
ReadWord(address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a word (16 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

ReadLong

Syntax

```
ReadLong ( address )
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a long (32 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

WriteByte

Syntax

```
WriteByte(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a byte (8 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

WriteWord

Syntax

```
WriteWord(address,  
           value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a word (16 bits) to a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

value is a number or string expression that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

WriteLong

Syntax

```
WriteLong(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a long (32 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

ReadRegister

Syntax

```
ReadRegister(RegisterName)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads from a target processor register.

Parameters and Remarks

Register Name specifies the register.

Examples

VBScript

```
Sub ReadSomeRegisters()  
    Write("Value of pc = " & ReadRegister("pc"))  
    Write("Value of r0 = " & ReadRegister("r0"))  
End Sub
```

JScript

```
function ReadSomeRegisters()  
{  
    Write("Value of pc = " + ReadRegister("pc"))  
    Write("Value of r0 = " + ReadRegister("r0"))  
}
```

ReadString

Syntax

```
ReadString(address,  
           length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated ASCII string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
call ReadString(address, 100)
```

JScript

```
if(ReadString(address, 100) != "Hello John")  
{  
    throw "Read/Write String failed"  
}
```

ReadWString

Syntax

```
ReadWString(address,  
            length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated UNICODE string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
ReadWString(address, 100)
```

JScript

```
if(ReadWString(address, 100) != "GoodBye")  
{  
    throw "Read/Write String failed"  
}
```

WriteString

Syntax

```
WriteString(address,  
            string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call WriteString(address, "Hello John")
```

JScript

```
WriteString(address, "Hello John");
```

WriteWString

Syntax

```
WriteWString(address,  
             string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) as unicode to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call WriteWString(address, "GoodBye")
```

JScript

```
WriteWString(address, "GoodBye");
```

Developing Your Application

Creating Projects and Configuring the Interface

Running CodeScape

To run CodeScape:

Click Start, point to Programs, point to CodeScape, and then click CodeScape.

Using the command-line

The command-line commands let you specify how CodeScape will run. For example, CodeScape can run from another application such as the Codewright editor or from a batch file.

To run CodeScape from the command-line type CodeScape then one or more optional switches. Always separate switches a space but do not use spaces within the argument of a switch.

The syntax is:

```
codescape[Switch]...
```

Files used by CodeScape

The *session* file contains the information needed to restore a previous debugging session.

The *program* file is an object file. It contains binary and optionally, source level debug and symbol table information produced by the assembler or compiler.

To change file or folder properties:

1. Click the file or folder whose properties you want to change.
2. On the File menu, click Properties.

NOTE: *You can drag a file's icon (or shortcut icon) into a document.*

Command-line switches

Use this switch:	To:
<code>[-]/?</code>	Run CodeScape and view Help on using the command-line.
<code>[-]/nologo</code>	Run CodeScape without the splash screen appearing.
<code>[-]/c</code>	View information on the Log tab as each command of the Imagination Technologies Fileserver library (libcross) executes. (Crosslib Verbose mode.)
<code>[-]/i=Session</code>	Run CodeScape and open the session file "Session". The session file specifies: target connections, object files used by each target, processor update rates, breakpoints, watch expressions, log expressions, and window positions and displays. If memory ranges are not set, CodeScape looks for them in DEFAULT.SSN. (Use Project Info.)
<code>[-]/autoload</code>	Run CodeScape using the last loaded session file.
<code>[-]/noautoload</code>	Run CodeScape without opening the last loaded session file. Use this option to override autoload specified in a batch file.
<code>-nomake</code>	Disable the project make facility. If CodeScape is running, it ignores this option.
<code>-s=script[,param-list]</code>	Run a script with the given (optional) parameter list
<code>-nogui</code>	If none of the other options specified require the gui to be present, exit CodeScape after loading and starting the program(s). If CodeScape is running, it ignores this option.

Loading a program file from the command-line

When you load a program file from the command-line, use the switch format:

```
-t#p# [b] [n] [r+|r-(expression)] [h|s] [c+|c-] [l+|l-]:Program
```

You must specify the target (t#) and processor (p#) to use, and the program file to load. The processor is identified by its processor ID # (0-7). The program file is specified by "program".

Unloading a program file from the command-line

When you unload a program file from the command-line, use the switch format:

```
-t#p#u
```

For example, CodeScape `-t1p1u` unloads the program file loaded on target 1, processor 1.

Optional switches for loading a program file

Use this switch:	To:
b	Download the binary from the object file.
n	Suppress debug information. The n option does not require symbols. If you use the n option without the b option it has no effect.
r+	Load and run the program file.
r-	Load, but don't run the program file.
r(expression)	Load and run the program file, then break at the address specified. "expression" is usually a symbol such as "main".
h	Perform a hard reset of the target, then load the program file.
s	Perform a soft reset of the target, then load the program file.
c+	Concatenate the sections in the program file.
c-	Don't concatenate the sections in the program file.
l+	Lock the program file.
l-	Don't lock the program file.

The following options are mutually exclusive:

- The run options: r+, r-, and r(expression).
- The reset options, h and s.
- The load options, c+ and c-.
- The lock options, l+ and l-.

CodeScape Project Manager

CodeScape's project configuration options simplify managing source and library files, and building projects. They help you to setup a project environment that includes the tools and procedures you want to use when writing, compiling, building, and debugging your application.

A project defines your development environment. It describes the file mappings, command-lines, and software tools you require to produce a program or final binary file(s).

Projects are organised in a session. A session can contain multiple projects. You can add a new project, or a new project configuration to a session at any time. You can also save a project as a template that will be available when you create a new session.

Saved with the session file are your project configuration settings, debug information, and interface settings.

This section is in three parts:

- *Overview* tells you about working with projects in CodeScape and the configurable options that are available.
- *Setting-up a new project* tells you how to setup a project for the first time.
- *Working with an existing project* tells you how to change any settings you require.

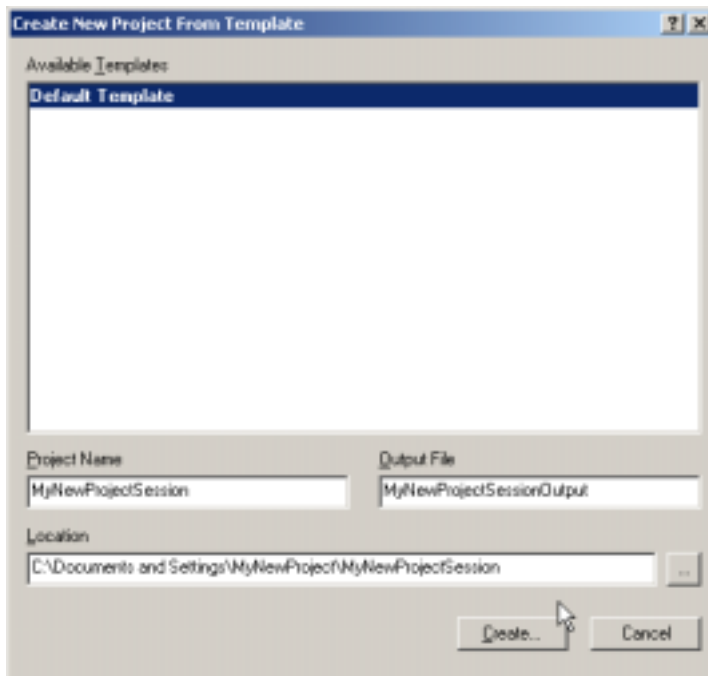
Overview

In CodeScape, a session is a container for your project(s). When you create a new session the Project Settings Wizard guides you through the steps you need to complete in order to set up your project. The process includes specifying the:

- Project name and output file.
- Configurations available to the project.
- File mappings for the linker and tools you will use.
- Files in your project.

Project name and output file

When you create a new session you must first specify a name, output file, and location for your project.



Project configurations

A project configuration consists of settings that determine the characteristics of the final output file. When you create a new project configuration, you copy its settings from an existing configuration and assign it a pre-defined underlying mode of debug or release. Debug has standard debug information, and release has standard optimization. Additional configurations that you create may have maximum optimization or contain maximum debug information.

Each project configuration has a hierarchical structure of settings for the files and folders they contain and specifies the directories in which the intermediate and final files are created. The settings at the project configuration level apply to all files within the configuration. However, you can also specify settings for individual files or a folder of files if you need to compile files with settings different from the general configuration settings. You can specify some types of settings, such as linking, only at the project configuration level.

Configuring the output file settings

You can configure the settings that determine the project output file by editing the settings for the build configuration options of the current project.

For example, you can specify a:

- Debug configuration that has full symbolic debugging information that can be used by the debugger and turns off optimizations.
-OR-
- Release configuration that does not contain any symbolic debugging information.
A Release configuration uses any optimizations set after creating the projects.

Each project Configuration also specifies the directories in which the intermediate and final files are created. The configuration settings supplied are Debug and Release. Debug has standard debug information, and Release has standard optimization.

You may want to create additional Configurations that have maximum optimization or contain maximum debug information.

Levels of optimization for your compiler are set in the tool's command line. For information on the command-line refer to the documentation supplied with your compiler.

Linker and tool file mappings

You must select a linker before specifying the tool file mappings.

Once you have selected a linker you can specify the file types to associate with the tools you want to use when working on your project. You can enter a new file type for a tool to support, or modify, or delete an existing configuration.

Setting up a version control system

With CodeScape you can access the commonly supported version control system features. These integrated operations do not include all possible capabilities of version control systems. Certain operations including setting-up a project and working directory must be done from within your version control system software.

You can:

- Specify the version control system you want to use. See *“On the Tools tab select a tool to configure, then enter a command line.” on page 104.*
- Open a session from your version control system.
- Add files to a project in your version control system.
- Get the latest version of one or more files from your version control system.
- Check In one or more files to your version control system.
- Check Out one or more files from your version control system.

To use the source control commands within CodeScape you must:

1. Install a version control system.
2. Run your version control system and setup your project on your database.
You must setup your project in your version control system before you can use any of the version control commands in CodeScape. You can setup a project in your version control system any time after you have created it.

Project files

When you have specified the tool file mappings, you add your project files. When you add your application's program files to your project they appear automatically on the Project Manager toolbar and are arranged in four folders: Source Files, Dependencies, External Dependencies, and all Other Files. File dependencies are automatically generated and maintained for your project.

- *Source Files* are the files to be compiled.
- *Dependencies* are the source file dependencies that are part of the project.
- *External Dependencies* are the source file dependencies that are not part of the project.
- *Unknown Dependencies* are the source file dependencies that are not recognized.
- *Other Files* are all other files that are part of the project, for example, binary files.

Setting up a new project: Procedures

When you setup a project for the first time, you:

1. Create a session and select a project template.
2. Specify the project configurations.
3. Specify the linker and tool file mappings.
4. Add your project files.
5. Complete the setup.

Creating a session and selecting a project template

Create a new session and select a project template, then specify a name, output file, and location for your project.

1. Click File, Session New...
The Create New Project from Template dialog appears.
2. Select a template in the Available Templates field.
If you have not previously created a template the only one available is the default.
3. Enter the Project Name.
4. Enter the Output File.
5. Specify the location to save the project to.
6. Click Create.
The Project Settings Wizard appears showing the configurations currently available to the project.

Specify the project configurations

The configurations currently available to your project appear on the Project Settings Wizard.

To open the Configurations dialog and add or remove a configuration, or to alter a configuration's mode click **Change**. The configurations associated with the CodeScape release templates are named **Debug** and **Release**.

Adding a new configuration

1. In the *New Configuration Name* box enter the name of the configuration you want to create.
2. In the *Copy Settings From* box select the configuration you want to base your new configuration on.
3. Click **Add**.
The new configuration appears in the Configuration Details tree.
4. Specify the Underlying Mode.
5. Click **OK**.

Editing a configuration's mode

1. In the Configuration Details tree select the configuration you want to edit.
2. Specify the Underlying Mode as **Debug** or **Release**.
3. Click **OK**.

Removing a configuration

1. In the Configuration Details tree select the configuration you want to remove.
2. Click **Remove**.
3. Click **OK**.

Specify the linker and tool file mappings

Specify the linker

- Select a Linker from the Current Linker list.
If the linker you require is not in the list you can specify it on the Tool Setup dialog. For more information see

Specify the tool file mappings

1. Click Add Mapping to specify the File Mappings for each tool you want to use.
 - a. Enter the Input Extension.
 - b. Select a Tool.
 - c. Select the Output Extension.
If only one Output Extension is possible the list appears in grey, if there is a choice the list is active.
4. Specify the Priority:
 - Select Extension First to sort the list by extension.
 - Select Interface First to sort the list by interface type.
5. Click OK.

Specify the project files

1. Click Add Files.
The Add Files to Project dialog appears.
2. Browse to and select the files to include in your project.
3. Click Add.
If you have not specified mappings for all of the file selected you are prompted to do so.
If you do not specify mappings for files in your project they will appear under Other Files.

Complete the setup

Once you have specified your project click Finish and you are ready to start work.

Working with an existing project: Procedures

The commands in the Project menu let you configure and build projects.

Project menu commands

Click:	To:
Make	Make the selected configuration's target file.
Rebuild All	Rebuild all project files.
Batch Build...	Select and build multiple project configurations on the Batch Build dialog.
Clean	Delete all intermediate files created during the build process, for example .obj, as well as output files such as the .exe or .dll file.
Compile	Invoke the compiler specified on the Project Setup dialog and create the object file required by the linker.
Update All Dependencies	Update the source file dependency list to reflect any changes made either since they were added to the project, or since the last build. View the Dependencies list on the Project Manager toolbar.
Add File(s)...	Select and add one or more files to the project.
Insert Project into Session...	Add a project to the current session.
Settings...	Configure the settings that determine the final project output file on the Settings dialog.
Tool Setup...	Set the environment variables for the compiler, assembler, linker, and version control system you will use for your project.
Configurations...	Add or remove a project configuration. For example, create a configuration called Maxdebug and base it on the existing configuration called debug, then specify the options you require Settings dialog.
Settings Wizard...	Open the Project Settings Wizard that guides you through the steps you need to complete in order to set up your project.
Source Control	Access the version control system options and view the current settings. See <i>"Setting up a version control system" on page 97.</i>
Edit Source Path...	Set, change, or remove the search path for your source files. For more information <i>refer to Debugging.</i>
Set FileServer Root Directory...	Set a directory path for your project to access any resource files it requires on the Set Fileserver Path dialog. The directory can be the same as the directory you set in Source File Search Paths. For more information <i>refer to LibCross Fileserver.</i>

Saving a project as a template

To create a template from your project settings, make sure that the options on the Settings dialog are as you want them then:

1. Click File, Save Project as Template.
The Save as Project Template dialog appears.
2. Enter a template name.
3. Click OK.
The template is saved in the same directory as CodeScape with the extension .tpl and appears in the list presented when you create a new session and project.

Specifying project configuration settings

The Configurations dialog lets you edit, add, and remove project configurations. The configurations associated with the CodeScape release templates are named Debug and Release.

Creating a new configuration

1. Click Project, Configurations...
The Configurations dialog appears.
2. In the *New Configuration Name* box enter the name of the configuration you want to create.
3. In the *Copy Settings From* box select the configuration you want to base your new configuration on.
4. Click Add.
The new configuration appears in the Configuration details tree.
5. Click OK.

Changing a configuration's settings

1. In the Settings dialog, select the Configuration you want to change the settings for.
2. On the project tree select the level you want to configure. Select:
 - The project folder to set project-wide options.
 - A sub-folder to set options only for all files in that folder.
 - A file to set options only for that file.
3. On the *General* tab set the Intermediate files directory path name for all levels, the Output files directory, and set the Output file for the project level only.
4. On the *Tools* tab select a tool to configure, then enter a command line.

Specifying a version control system

You can view the version control systems integrated with CodeScape from the Source Control tab in the Options dialog or in the Version Control Settings dialog.

- The Options dialog Source Control tab lets you integrate additional version control systems to use and support. You can also modify or delete existing ones.
- The Version Control Settings dialog lets you select the version control system and database you want to use for the current project.

To use the source control commands within CodeScape you must:

1. Install a version control system.
2. Run your version control system and setup your project on your database.
You must setup your project in your version control system before you can use any of the version control commands in CodeScape. You can setup a project in your version control system any time after you have created it.

Setting up a new version control system

1. Click Project, select Source Control then click Version Control Settings...
The Version Control Settings dialog appears.
2. In the *Version Control System* box select the version control system you require.
 - If your version control system is Microsoft Visual Source Safe V5.0 or above, select Source Safe Automation for optimum compatibility.
 - If the version control system you require is not in the list, close the dialog. On the Options dialog Source Control tab add the version control system.
3. In the *Database* box enter the location of your project's central database.
4. In the *Username* box enter the login username assigned to you by your version control system administrator.
If your version control login is automatic leave this box blank.
5. In the *Password* box enter the login password assigned to you by your version control system administrator.
If your version control login is automatic leave this box blank.
6. In the *Options* box select:
 - *Get file(s) when opening the session* if you want to retrieve the latest version of your project files from your version control system database and copy it to your computer when you open a session.

- *Prompt to add file(s) when inserted* if you want to add files to your version control system database when you add them to your project.
- *Check-out file(s) when edited* if you want to receive prompt when you check out a file for editing.
- *Check-in file(s) when closing session* if you want to copy the latest version of your project files from your computer to your version control system database when you close a session.

7. Click OK.

Adding, opening, compiling, and updating project files

Use the Project Manager toolbar to view and access the various elements of your project. Options include: opening, compiling, and updating selected files; adding and removing files; and creating a dependency list for a selected file.

To view the Project Manager toolbar:

1. Click View, Toolbar...
The Toolbar Configuration dialog appears.
2. Select the Project check-box.
3. Click OK.
The Project Manager appears.

For more information see *“Project Manager toolbar” on page 139*.

Making and building a project

Make the project current and build it in one of the following ways:

- Click Project, then click Make.
-OR-
- On the Project Build window, right-click, click Make.

The Input/Output window Build tab appears and automatically displays the specified build utility’s output about the build. Any standard format errors and warnings are shown in the Input/Output window Build tab.

If a build error occurs, double-click an entry to invoke the editor and open the source file at the line containing the error or warning. To advance to the next error or warning press F4.

For more information on CodeScape’s Editor *refer to Debugging*.

Use and configure the interface

You can control CodeScape using a mouse or keyboard. CodeScape has many useful toolbars that can be docked, floating, or hidden. Region-specific shortcut menus are available for the most used functions.

The user interface consists of: the menu bar, toolbars, windows, and regions.

Commands on the menu bar

File menu ALT+F

The File menu commands are for working with sessions, resetting the target, loading program files, restarting program file execution, and saving and loading binary information. Use Save binary and Load binary to move large blocks of data in and out of memory. In some cases you can also configure a serial port for serial debugging.

The File menu also displays a list of recently used session files. You cannot hide this list or change the number of files displayed.

Notes

1. *If you load a session file on a target that is different to the type it was created on, it loads without the program file.*
2. *If you add or remove a target during a session, you are prompted to restart CodeScape and, if necessary, to save unsaved session information.*

File menu commands

Click:	To:
New...	Open a new file in the source editor.
Open...	Open an existing file in the source editor.
Session New...	Open a new session file.
Session Open...	Open an existing session file.
Session Close	Close the currently open session file.
Session Save...	Save the current session file. If your file does not have a name, the File Save As dialog box appears.
Session Save As...	Save the current session file with the name you enter in the Save As dialog.
Save Project as Template	Create a template from your current project setup for use when defining future projects.
Save	Save the file in the active source editor.
Save All	Save all files open in editable regions.
Save As...	Save the file in the active source editor with a new name.

Click:	To:
Reset Target	Reset the target with either a soft reset or a hard reset. A soft reset restores the state of the target and re-initializes the monitors. If a soft reset fails you are prompted to do a hard reset. A hard reset resets the target and reloads the monitors.
Load Program File...	Load a program file to the target. If you have a project open you cannot change the name of the program file.
Unload Program File Info	Clear all debug information and close the program file. This does not remove the program file from target.
Restart	Restart the currently active program file. Restart loads the binary part of the current program file and resets the PC to the entry point (if known). If the program file's last modification time has changed, symbolic information is also loaded.
Save Binary	Specify a large block of data to save from memory. It is useful for saving bitmaps or processor specific code to a selected area of memory.
Load Binary	Specify a large block of data to move into memory. It is useful for loading bitmaps or processor specific code from a selected area of memory.
Print	Print the contents of the active window.
Print Setup	Change the printer and the printing options.
Recent Files	Loads the selected recent file in the source editor. When you open a project file it is shown first in the list and the last file is removed from the list, if the file is already in the list it is moved to the top.
Recent Sessions	Loads the selected recent session file. When you open a session it is shown first in the list and the last file is removed from the list, if the file is already in the list it is moved to the top.
Exit	Close CodeScape. When you Exit, CodeScape prompts you to save any un-saved session information. When you next open the session CodeScape loads this information for you.

Edit menu ALT+E

The Edit menu commands let you undo and redo actions; cut, copy, and paste; and perform searches in the Editor. The Edit menu appears when you open a window, create a region, or load a session.

Edit menu commands

Click:	To:
Undo	Undo the last action.
Redo	Redo the last action.
Cut	Cut the selection and paste it to the clipboard.
Copy	Copy the selection and paste it to the clipboard.
Paste	Paste the contents of the clipboard at the insertion point.
Find...	Search for a string from the insertion point.
Find Next	Search for the next instance of the string.
Find In Files...	Search for a specified string in more than one file.
Replace...	Find and replace a string. Move the insertion point to where you want to start replacing from, click Replace.
Go To Line...	Enter a line number for the address to go to.

View menu ALT+V

The View menu commands let you show and hide toolbars and configure regions.

View menu commands

Click:	To:
Toolbar...	The toolbars provide access to the main debugging functions. Show or hide toolbars using the Toolbar Configuration dialog.
Status Bar	Show or hide the status bar.
Properties...	Configure the display and update rate options for a region. This helps you to differentiate between processors, show associated regions, and represent changes in memory.

Project menu ALT+P

The commands in the Project menu let you configure and build projects.

Project menu commands

Click:	To:
Make	Make the selected configuration's target file.
Rebuild All	Rebuild all project files.
Batch Build...	Select and build multiple project configurations on the Batch Build dialog.
Clean	Delete all intermediate files created during the build process, for example .obj, as well as output files such as the .exe or .dll file.
Compile	Invoke the compiler specified on the Project Setup dialog and create the object file required by the linker.
Update All Dependencies	Update the source file dependency list to reflect any changes made either since they were added to the project, or since the last build. View the Dependencies list on the Project Manager toolbar.
Add File(s)...	Select and add one or more files to the project.
Insert Project into Session...	Add a project to the current session.
Settings...	Configure the settings that determine the final project output file on the Settings dialog.
Tool Setup...	Set the environment variables for the compiler, assembler, linker, and version control system you will use for your project.
Configurations...	Add or remove a project configuration. For example, create a configuration called Maxdebug and base it on the existing configuration called debug, then specify the options you require Settings dialog.
Settings Wizard...	Open the Project Settings Wizard that guides you through the steps you need to complete in order to set up your project.
Source Control	Access the version control system options and view the current settings.
Edit Source Path...	Set, change, or remove the search path for your source files. For more information <i>refer to the Debugging guide</i> .
Set FileServer Root Directory...	Set a directory path for your project to access any resource files it requires on the Set Fileserver Path dialog. The directory can be the same as the directory you set in Source File Search Paths. For more information <i>see LibCross Fileserver guide</i> .

Debug menu ALT+D

The Debug menu commands let you control program execution, step code, and use breakpoints. You can set a base address range to map one or more associated address ranges from.

Other commands include locking the view to an Expression that contains the PC, a register, or memory, and setting the cursor to the PC and vice-versa. The default origin is set to the value of the PC.

Debug menu commands

Click:	To:
Start Debugging	Toggle Start Debugging on and enable CodeScape's debugging features. The menu option changes to Stop Debugging. Stop Debugging disables CodeScape's debugging features. Until you start debugging the Editor is the only region you can create. This option only appears when you have created or opened a project. For more information on the debugging options see <i>Debugging</i> .
Execution	Run, stop, and restart a program. Run a program until it executes a specific address. Run all program files simultaneously. Stop all programs running simultaneously.
Breakpoints	Add, enable, disable, configure, reset, or remove data breakpoints.
Set Cursor to PC	Set Cursor to PC. Make a Source or Disassembly region active, then select this option to show the source code from the value of the PC.
Set PC to Cursor	Set PC to Cursor. Make a Source or Disassembly region active, then select this option to change the PC at the current cursor position.
Goto Address...	Enter an expression for the address to go to.

Region menu ALT+R

The Region menu commands let you split (create new) regions, delete regions, change a region's type, update all regions on demand, or stop updates to all regions. The Region menu is available when you open or create a window, region, or session.

Region menu commands

Click:	To:
Split	Split the active region left, right, up, or down.
Delete	Delete the active region.
Type	Specify a region type for the active window.
Update All Regions Now	Update the display in all open regions.
Stop All Region Updates	Stop all regions from being updated. Select this option if the region update rates interrupt the target causing jitter in your program's display. Disabling the update rate stops all region displays from updating on the current processor.

Tools menu ALT+T

The Tools menu commands include running script files and adding script files to run from the Tools menu. Any scripts you add to run from the Tools menu can also be run and the shortcut menu on the Scripts tab on the Input/Output window. You can also set-up target debug support, specify custom shortcut keys, and add programs to run from the Tools menu.

Tools menu commands

Click:	To:
Simulate Processor	Run the Simulator.
Profiler	Start the Profiler and open the Profiler Setup dialog where you can configure how your profile information is generated.
Configure Target Communication	Configure options for any or all of the targets that you are connected to.
Customize	Customize the shortcut keys, tools menu. <i>See “Customizing shortcut keys” on page 117.</i>
Options	Specify system wide configuration options on the Options dialog. <i>See “Options dialog” on page 119.</i>
User Tools	Run the associated program. You can associate a command with up to ten programs.

Customizing shortcut keys

You can specify shortcut keys for any of the commands on the menu bar or on the shortcut menus.

When you assign a shortcut key to a command CodeScape saves the setting in Keyboard File, CODESCAPE.MAC. Each time you change a keyboard shortcut CodeScape updates the information in the Keyboard File. When you run CodeScape it automatically detects and loads the Keyboard File, CODESCAPE.MAC.

You can save your settings to a Keyboard File with a specific name. This is useful if you use CodeScape on more than one computer, or if you share a computer with another person.

To save your settings to a Keyboard File with a specific name:

1. In the Shortcut Keys dialog box create and remove any shortcut keys you require, then click Save...
2. Enter a name for the Keyboard File using the extension .MAC, then click OK.

NOTE: *To load a Keyboard File, click Load... and enter the filename and location of the Keyboard File that you want to use. When you load a Keyboard File the settings it specifies are automatically copied to CODESCAPE.MAC.*

You can:

- Assign shortcut keys to a command.
- Remove shortcut keys from a command.
- Restore the shortcut keys default settings.
- Use the Microsoft Developer Studio shortcut key commands.

Assigning shortcut keys to commands

1. Click Tools, select Customize then click Keyboard...
The Shortcut Keys dialog box appears.
2. In the *Select a command:* tree, highlight a menu command to assign a shortcut.
A description of the command appears under Descriptions, and any shortcut keys appear in the *Assigned shortcuts* list.
3. Click Create Shortcut...
The Select shortcut dialog box appears.
4. Press the key combination you require.
If you press a key or key combination that is currently assigned to another command, that command appears under Replaces.
5. Click OK.
The new shortcut appears in the *Assigned shortcuts* list.
6. Click OK.

Removing shortcut keys from commands

1. Click Tools, select Customize then click Keyboard...
The Shortcut Keys dialog box appears.
2. In the Select a command tree highlight a menu command.
A description of the command appears in the Descriptions box, and any shortcut keys appear in the *Assigned shortcuts* list.
3. Click Remove.

Restoring shortcut key default settings

1. Click Tools, select Customize then click Keyboard...
2. Click CodeScape Defaults.

Using Microsoft Developer Studio shortcut key commands

1. Click Tools, select Customize then click Keyboard...
2. Click DevStudio compatible.

For menu items that do not have a Microsoft Developer Studio default value, for example Simulate Processor, the CodeScape default shortcut key is used.

Options dialog

The Options dialog box lets you specify system wide configuration options.

Interface tab

1. Under *Expression/Symbols*, select any of the following:
 - *Qualify Labels and Names* to qualify the context of labels and names by adding scope information to the front of symbols in Call Stack and Disassembly regions.
 - *Use Abbreviated File Names* to use the shortest unique form of each file name.
 - Global Expression History to share the expression history in all windows and all regions.
 - *Hide Optimized-Out Local Symbols* to prevent variables that are optimized out (and therefore have no address information) from appearing in the Local Watch region. This is the default setting, if this check-box is cleared then optimized out variables appear with the message "variable optimized out".
2. Under User Interface, select any of the following:
 - *Proportional Resizing* to proportionally resize a window and its regions.
 - *Syntax Highlighting* in Source Region to turn syntax coloring on or off in Source regions. Specify a color for any or all of the following items in your code: keywords, quotes, comments, default text, and the background.
 - *Automatic Source/Disassembly switching* to automatically show disassembly if source is not available.

Action tab

1. Under *On Fileserver Request*, select any of the following:
 - *Optimize Performance* to disable updates to the display while data is being transferred from the target to your computer.
 - *Process ASSERTs* to display a message describing when and where Fileserver_ASSERTs occur.
2. Under *On Exception*, select *Display Message* to show a message when an exception occurs.
3. In the *On Startup* field, select any of the following:
 - *Load Last Session On Startup* to load the last open session when CodeScape restarts.
 - *Open a Source Region on Startup* to open a Source region if a session is not loaded when CodeScape starts.
4. Under *On Reset*, select *Reload Program File*. CodeScape reloads your program's binary information. It also reloads the debug information if it has changed.
5. Under *On Idle*, select *Attempt to keep debug data swapped in* (may affect performance of other applications) to prevent debug information from being swapped to disk when possible. This can cause all open applications to run slowly.
6. Under *Build*, select *Unload Existing Program File Information Before Building* to tell CodeScape to unload the open program file before building.
This reduces the memory footprint during the build. It also stops the program file information from being put in the swap file (getting code back from the swap file is very slow).

Debug tab

1. Under *General*, select any of the following:
 - *Reuse Source Regions* to reuse a single region (Source, Disassembly, Edit) when debugging would otherwise create multiple regions.
Select *Reuse Source Regions* and *Start Debugging*. When the state of the PC changes, for example if you trace into a new source file, CodeScape opens the file in an active Source/Disassembly region and gives it focus.
Clear *Reuse Source Regions* and *Start Debugging*. When the state of the PC changes, for example if you trace into a new source file, CodeScape opens the file in a new Source/Disassembly region and gives it focus.
 - *Default stepping method is to step source* to trace disassembly in Disassembly and Mixed regions, and if a Source region is shown for the active target processor, trace source in all other regions otherwise trace disassembly.
 - *Save breakpoints as "label + offset"*.
 - *Allow memory access while target is running* to enable memory reads and writes without stopping the target. On by default.
2. Under *Target Specific*, select a target then select any of the following:
 - *Allow automatic software breakpoints*.
 - *Allow software breakpoints for multiple instances of inline code*.

Editor tab

1. Under *Appearance*, set any of the following to display them on the Editor region:
 - *Vertical Scroll Bar*.
 - *Horizontal Scroll Bar*.
 - *Selection Margin*.
 - *Show White Space*.
 - *Tab Size other than the default of 4*.
 - *Use Syntax Coloring*.
2. Under *Others*, select any of the following:
 - *Word Completion*.
 - *Auto Indent*.
 - *Bracket Matching*.
3. Click *Configure Languages...* The Language dialog appears. See "*Language dialog*" on page 123.

Format tab

Set color and font attributes to differentiate between processors, show associated regions, and represent changes in memory.

1. In the Targets field, select either All Targets, or a specific target to configure option for from the list.
2. In the Category field, select either All regions, or a specific region to configure option for from the list.
3. In the Color field, specify the item you want to configure the display color for.
4. Set the formatting options you require.
5. Click OK.

Tools tab

The Tools tab shows the tools used, such as compiler and assembler, and their command lines. You can add a new tool to use and support, or modify, or delete an existing configuration.

Linker tab

The Linker tab shows the linker defined for the project, the linkers that are available, and their command lines. You can add a new project linker to use and support, or modify, or you can delete an existing configuration.

Regular Expressions tab

The Regular Expressions tab shows the regular expressions used to specify and match strings to file extensions.

Source Control tab

The Source Control tab shows the revision control system used, it's command-line, and commands.

Language dialog

On the Language dialog you can:

- *Edit* the properties of a language in the list.
- *Insert* a new language to the list and specify its properties.
- *Delete* a language from the list.
- *Copy* the settings from a language in the list and create a new language definition based on those settings. For example, select C in the list and copy it to a new definition C++ then just add the additional keywords in the Language Setup dialog.

When you choose to Edit, Insert, or Copy the Language Setup dialog appears.

Language Setup dialog

1. Add or Delete any *Key Words*.
2. Specify the *Start*, *Middle*, and *End* characters for *Key Word Helpers*.
3. Enter *Symbols* and *Delimiters*.
4. Specify the *Tab Size*.
5. Select or clear *Case Sensitive* as applicable to the language.
6. Specify the tag settings:
 - In *Begin* enter the tag start characters.
For example, a comment that spans multiple lines in C/C++ starts with `/*`.
 - In *End* enter the tag close characters.
For example, a comment that spans multiple lines in C/C++ ends with `*/`.
 - In *Escape* enter the line continuation character. For most languages this is `\`.
 - Select *Multiple Line* if the Tags can span more than one line through carriage returns.
7. Enter the supported *File Extension(s)*.
8. Click Configure Syntax.
9. The Font and Color Settings dialog appears. Configure the font and color syntax settings for your Edit regions. See “*Font and Color Settings dialog*” on page 124.

Font and Color Settings dialog

On the Font and Color Settings dialog configure the font and color syntax settings for your Edit regions.

1. Under Color select the item you want to configure.
2. Specify the Foreground and Background colors.
3. Specify the Font settings.
4. Under Sample view your changes.
5. Click OK.

Window menu ALT+W

Use the Window menu to open a new window. When you open a new window, the Edit and Region menus appear on the menu bar, and additional commands appear on the Window menu which are for arranging multiple windows in CodeScape. You can select and clear commands for proportionally resizing a window and its regions, and loading the current session when you next run CodeScape.

The Window menu also displays a list of region types and highlights the currently active region. When you have more than one region in a window frame, the active region within that frame appears in the list. You cannot hide this list or change the number of regions displayed.

NOTE: *If you use a Windows® 95 or a Windows® 98 machine, creating too many new windows regions causes CodeScape to run out of system resources.*

Window menu commands

Click:	To:
New Window	Open a new window.
Cascade	Cascade all region windows inside the main window.
Tile	Tile all region windows inside the main window.
Arrange Icons	Arrange all minimized region windows at the bottom of the main window.
Zoom Region	Zoom to the region in a split window that has focus.
Close All Windows	Close all windows.

Help menu F1

Use the Help menu to get online Help and view CodeScape version information.

Help menu commands

Click:	To:
Help Topics...	View the main Help file.
Keyboard	View a list of the currently assigned shortcut keys. Keyboard shortcuts are available as an alternative to using a mouse.
About CodeScape...	View version and copyright information.

Region specific shortcut menus

Each region has two shortcut menus. The Region Type menu commands are for changing a region's type. To see the menu:

- Press CTRL+SHIFT+F10.
- OR-
- CTRL+Right-click anywhere in the region.

The Region Actions menu has region specific commands, and global commands for controlling program execution or manipulating breakpoints. To see the menu:

- Press SHIFT+F10.
- OR-
- Right-click anywhere in the region.

The commands on the toolbars

The toolbars provide access to the main debugging functions. To use the *Toolbar Configuration* check box to show or hide toolbars click View, Toolbar, then select or deselect toolbars from the list.

Toolbars and their uses

Use the:	To:
Breakpoint toolbar	Access the most common breakpoint actions.
Debug toolbar	Access the debugging actions.
Processor Combo toolbar	View the active processor for the selected target. The toolbar also provides point and click access for loading program files and configuring the current processor.
Input/Output window	View: the build utility's output when you build a project; all messages generated by the target; and all messages generated by an executing script.
Region toolbar	Set and change a region's type.
Region Combo toolbar	Set the rate at which a region's display is updated, change a region's type.
Splitter toolbar	Split regions using the mouse.
Standard toolbar	Open new windows, and create, open, and save sessions.
Target window	View the active processor for the selected target, load program files and configure the target.
Target Combo toolbar	View and configure the active target.
Editor toolbar	Use the editing actions.
Project toolbar	View and access the various elements of your project.
Build toolbar	Change your project's build configuration, compile individual files, and make your project current by building it.
Status Bar	View contextual information about commands on the interface and any other information about the current state of the interface.

View, hide, dock, and move toolbars


View toolbars

- Right-click the status bar.
-OR-
- Right-click on a blank area of any toolbar select the required toolbar.
-OR-
- Click View, Toolbar... select the required toolbar. Click OK.

Hide toolbars

1. Right-click on a blank area of any toolbar.
2. Clear the toolbar from the list.

If the toolbar is undocked:

- Right-click on the toolbar title bar and click Hide.
-OR-
- On the toolbar title bar, click .

If the toolbar is docked:

1. Click View, then point to Toolbar...
2. Clear the *toolbar* check box. Click OK.

Dock toolbars

- Drag the toolbar to an edge of the main window.
-OR-
- Double-click the title bar. The toolbar will be docked at its last docked position.

Move toolbars

1. Do one of the following:
 - On the toolbar title bar, right-click and click Move.
 - OR-
 - Click the toolbar title bar.
2. Drag the toolbar to the required position.

NOTE: You cannot dock the Target window or the Input/Output window if you are still pressing CTRL.









Commands on each toolbar

The commands on the toolbars provide access to the main debugging functions. You can also use the Keyboard shortcuts for most debugging operations, and Access keys support all operations.

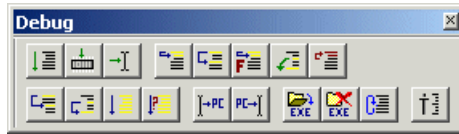
Breakpoint toolbar



Commands on the Breakpoint toolbar





To issue this command:	Click:	Press:
Toggle Breakpoint		F5
Enable Breakpoint		none available
Disable Breakpoint		none available
Configure Breakpoint(s)		CTRL+F5
Reset All Breakpoints		ALT+F5
Enable All Breakpoints		CTRL+SHIFT+F5
Disable All Breakpoints		CTRL+ALT+F5
Remove All Breakpoints		SHIFT+F5

Debug toolbar

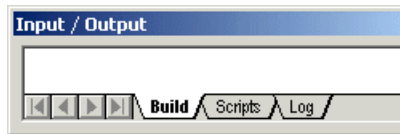


Commands on the Debug toolbar

To issue this command:	Click:	Press:
Run		F9
Run to Address		SHIFT+F9
Run to Cursor		ALT+F9
Stop		F9
Single Step (into)		F7
Forced Step (into)		none available
Step Over		F8
Step Out		CTRL+F8
Unstep		CTRL+F7
Step Run In		SHIFT+F7
Step Run Out		SHIFT+F8
Step Run		ALT+F7
Step Run Until		ALT+F8
Set Cursor to PC		CTRL+SHIFT+P

To issue this command:	Click:	Press:
Load Program File		CTRL+SHIFT+C
Unload Program File Info		CTRL+ALT+U
Set PC to Cursor		CTRL+ALT+P
Restart		CTRL+SHIFT+R

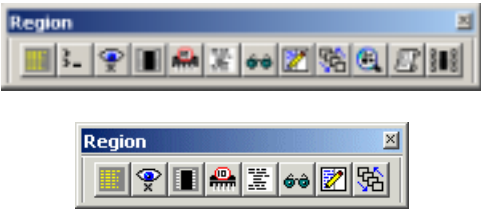
Input/Output window



The Input / Output window appears automatically and displays the:

- **Build tab** with the specified build utility's output when you build your project.
- **Scripts tab** with all messages generated by the current script.
- **Log tab** with all messages generated by the current target.

Region toolbar

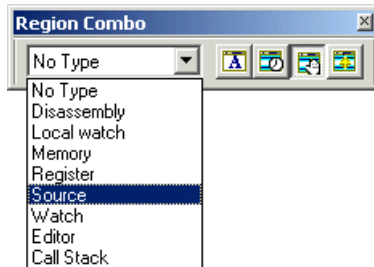


The Region toolbar lets you set or change a region’s type. To stop the display from updating in all regions press CTRL+SHIFT+U.

Commands on the Region toolbar





In the active window, create a:	Click:	To create the region in a new window press:
Disassembly region		ALT+1
Local Watch region		ALT+3
Memory region		ALT+4
Register region		ALT+5
Source region		ALT+6
Watch region		ALT+7
Edit region		ALT+8
Call Stack region		ALT+9
Peripheral region		

Region Combo toolbar



The Region combo toolbar lets you set the rate at which a region's display updates, and change a region's type.

Commands on the Region Combo toolbar

To set this command:	Click:
Region configuration	
Window update rate	
Stop all window updates	
Update all regions	

Splitter toolbar



The Splitter toolbar lets you split existing regions to create new regions.

Commands on the Splitter toolbar












To issue this command:	Click:	Press:
Split Left		CTRL+SHIFT+LEFT ARROW
Split Right		CTRL+SHIFT+RIGHT ARROW
Split Up		CTRL+SHIFT+UP ARROW
Split Down		CTRL+SHIFT+DOWN ARROW
Delete Region		CTRL+D

Standard toolbar



The Standard toolbar provides point and click access for opening a new window, and creating, opening, and saving sessions.

Commands on the Standard toolbar

To issue this command:	Click:	Press:
New window		CTRL+N
New Session		CTRL+SHIFT+N
Open Session		CTRL+O
Save Session		CTRL+S
Cut		CTRL+X
Copy		CTRL+C
Paste		CTRL+V
Print		CTRL+P
About Box		none available
Help		F1
Find in Files		none available

Target window

The Target window shows all the targets that CodeScape is connected to and the processors available in each target. The processor status for each target is shown in the Target processor display.

When you create a new window, CodeScape uses the target information from the selected processor on the target. The Target window provides point and click access for selecting a target processor.












NOTE: *The Target window can be docked at the top and bottom of the main window, or left free floating.*

Editor toolbar



The Editor toolbar provides point and click access to the editing actions.

Commands on the Editor toolbar

To issue this command:	Click:	Press:
Create a new Editor file.		none available
Open an existing Editor file.		none available
Save the current Editor file.		none available
Undo the last action.		CTRL+Z
Redo the last action.		none available
Search for a string.		CTRL+F
Replace the current selection.		none available
Toggle a Bookmark on or off.		none available
Move to the next Bookmark in the file.		none available
Move to the previous Bookmark in the file.		none available
Delete all Bookmarks.		none available

Project Manager toolbar

The Project Manager toolbar is for viewing and accessing the various elements of your project. The toolbar has two tabs, Project Manager and Class Info.

Project Manager tab

The Project Manager tab shows information about each project file in a session. The:

- *Session* folder shows the session file that the project belongs to.
- *Project Files* folder shows the active named project containing all files including source and dependencies.
- *Source Files* folder shows the files to be compiled.
- *Dependencies* folder shows source file dependencies such as header files. These files are included in the project.
- *External Dependencies* folder shows source file dependencies. Source file dependencies are not necessarily part of the project such as system headers.
- *Other Files* folder shows all other files that are part of the project, for example, binary files.

Commands on the Project Manager tab

Click:	To:
Open	Open the selected file(s) for editing.
Compile	Compile the selected file(s).
Touch file(s)	Update the time stamp for the selected file(s).
Open in Windows Explorer	Launch Windows Explorer and view the location of the selected file.
Add to Project...	Add one or more files and any dependencies to the project.
Remove From Project	Remove the selected file(s) from the project.
Make Dependencies File	Create a dependency list for the selected source file and save it to a file.
Properties	Configure fonts and colors. Set the update rate for a single region, a region type, and each processor. Change the tab settings.
Allow Docking	Toggle docking for the window on or off.

Click:	To:
Hide	Hide the window.

Class Info tab

The Class Info tab only appears after it has been activated. To do this, double-click on `classbrowser.reg` located in the same directory as `CodeScape.exe`.

This inserts the following entry into your registry:

```
[HKEY_CURRENT_USER\Software\Imagination
Technologies\CodeScape\300\CodeScape\Global]
```

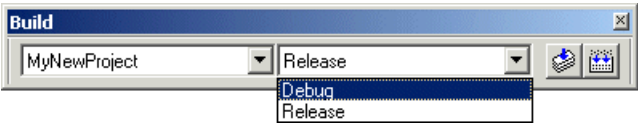
```
"EnableClassBrowser"=dword:00000001
```

The Class Info tab is available when you open a session with one or more files that contain C/C++ class definitions. The Class Info tab displays icons that represent classes and structures including functions, variables, enumerators or typedefs, and globals. Double-click an icon in the Class Info tab list to invoke the Editor and go to the corresponding line of code.

Icons on the Class Info tab



The icon:	Indicates:
	Protected function.
	Private function.
	Public function.
	Protected variable.
	Private variable.
	Public variable.
	Protected enumerator or typedef.
	Private enumerator or typedef.
	Public enumerator or typedef.

Build toolbar



The Build toolbar shows the current project and configuration. The Build toolbar lets you select the project and the project configuration you want to work on and compile individual files or make and build the whole project configuration.

Commands on the Build toolbar



To issue this command:	Click:	Press:
Compile file		none available
Make (The Make button is replaced with Stop Make when CodeScape is building.)		CTRL+M

Configuration toolbar



The configuration toolbar provides point and click access for configuring and refreshing the target.

Commands on the Configuration toolbar

To issue this command:	Click:
Configure target	
Refresh target communications	

How windows and regions work

A **Window** is a frame that you can configure as a **Region**. You can create more than one window to display different types of information such as memory contents and source code. You can also split any number of windows to create multiple **Regions**.

A **Region** lets you view information about your project.

To view a project's regions simultaneously you can:


- Open and close, minimize and maximize, cascade, and tile multiple windows.
- Split windows into multiple regions to display different types of information.
- Resize windows.
- Resize regions by moving the split bars.
- Proportionally resize a window's regions.

For information on each region type *refer to the CodeScape Debugging guide*.


NOTE: *If you use a Windows® 95 or a Windows® 98 machine, creating too many new windows or too many new regions causes CodeScape to run out of system resources.*

Using windows


Open a new window

- Click Window, then click New window.
- -OR-
- Click  on the Standard toolbar.


Minimize a window

- On the window title bar click .
- -OR-
- On the System menu, click Minimize.

Maximize a window

- On the window title bar click .
- -OR-
- On the System menu, click Maximize.

Close a window

- On the window title bar click .
- -OR-
- On the System menu, click Close.

NOTE: *If you delete the only region in a window, the window is also deleted.*

Close all windows

- Click Window, then click Close all Windows.

Move a window

1. Click the window's title bar.
2. Drag the window to the required position.

Move between windows

- Press CTRL+TAB.

Resize a window

1. Point to the window border.
2. Click and drag the window outline to the required size.

To proportionally resize a region in a window:

1. Click Window, then select Proportional resizing.
2. Point to the window's border, then click and drag the window to the required size.

Cascade all windows

- Click Window, then click Cascade.

Tile all windows

- Click Window, then click Tile.

Arrange Icons

To arrange all minimized region windows at the bottom of the session window:

- Click Window, then click Arrange Icons.

Using regions

Change a region's type

- Click Region, then point to Type, then click a region type.
-OR-
- On the Region Combo box, select a region type from the list.
-OR-
- Click a Region Type icon on the Region toolbar.
-OR-
- CTRL+Right-click, then click a region type.

Scroll through a region

- Use the LEFT ARROW and RIGHT ARROW keys to move the cursor a single character at a time.
- Use the UP ARROW and DOWN ARROW keys to move the cursor up and down a line at a time.
- Use PAGE UP and PAGE DOWN to move up and down a page at a time.
- Use the HOME and END keys to move to the first visible line and the last visible line of a file.

Move through a region's fields

- To move the cursor to the next field, press TAB.
- To move the cursor to the previous field, press SHIFT+TAB.

NOTE: *At the end of a field the cursor moves to the next field; at the end of the last field the cursor moves to the next line.*

Move between regions

To move to the region to the left:

- Click anywhere in the region to the left.
- OR-
- Press CTRL+ALT+LEFT ARROW.

To move to the region to the right:

- Click anywhere in the region to the right.
- OR-
- Press CTRL+ALT+RIGHT ARROW.

To move to the region above:

- Click anywhere in the region above.
- OR-
- Press CTRL+ALT+UP ARROW.

To move to the region below:


- Click anywhere in the region below.
- OR-
- Press CTRL+ALT+DOWN ARROW.

Create new regions


- Click Window, New Window to open a new window. On the Region toolbar click the region type you require.
- OR-
- Split an existing region.

Split regions


To split a region to the left:

- Click Region, point to Split, then click Left.
-OR-
- Click  on the Splitter toolbar.
-OR-
- Press CTRL+SHIFT+LEFT ARROW.


To split a region to the right:

- Click Region, point to Split, then click Right.
-OR-
- Click  on the Splitter toolbar.
-OR-
- Press CTRL+SHIFT+RIGHT ARROW.

To split a region above:


- Click Region, point to Split, then click Up.
-OR-
- Click  on the Splitter toolbar.
-OR-
- Press CTRL+SHIFT+UP ARROW.

To split a region below:

- Click Region, point to Split, then click Down.
-OR-
- Click  on the Splitter toolbar.
-OR-
- Press CTRL+SHIFT+DOWN ARROW.

Delete a region

Make the region active, then do one of the following:

- Click Region, click Delete.
- OR-
- Click  on the Splitter toolbar.
- OR-
- In the region, press CTRL+Right-click and click Delete Region.

NOTE: If there is only one region in a window, deleting it also deletes the window.


Update the display in all open regions

- Click Region, then click Update all regions now.

Configuring regions

The Region Configuration dialog commands let you set the update rate for a single region, a region type, and each processor.

To configure an active region:

- Right-click, click Properties...
- OR-
- On an active region's title bar, double-click .

The Region Configuration dialog appears.

Set the mode, target, processor, and region type

Do the following:

1. Set the Mode. Select:
 - *Apply to active region only* to configure the active region.
 - *Apply to all regions of selected type* then select a Region type to configure all regions of a selected type on all processors.
2. Then do one of the following:
 - Click Apply to view your configuration changes without leaving the dialog.
 - OR-
 - Click OK to set the configuration changes for your project.

Set the region update rates

Use the Update Rate tab to specify when CodeScape will update information in each region.

If the update rate for a region's display interrupts the target causing jitter in your program, set the Foreground and Background sliders to Min.

To specify when CodeScape will update information in a region:

1. Select the Update tab.
2. Drag the Foreground slider to set the update rate for when the region has focus. Set the slider to Max to continually update the display (approximately 14Hz). Set the slider to Min to update the display at approximately 1/10th of the Max setting.
3. Drag the Background slider to set the update rate for when the region does not have focus. Set the slider to Max to continually update the display. Set the slider to Min to prevent updates to the display.

Configuration file: dali.cfg

The memory configuration file `dali.cfg` provides CodeScape with target specific memory access information. It also lets you specify the default breakpoint type (hardware or software).

Edit `dali.cfg` to make sure that it specifies the memory areas and processor specific timing information you require. Before you edit `dali.cfg`, first make a copy of the original file and save it with a new name so that you can re-create the original at any time.

File format

Dali.cfg has four sections that describe how target memory is used for each processor. You only need to specify information for the processors that you use. Each section has a heading with the format:

```
[SECTION LABEL_SectionName]
```

Where `SECTION LABEL` is the processor type and `SectionName` is the section heading.

The section headings are:

- `[SECTION LABEL_ValidMemory]` specifies valid memory areas, access methods, and timing information.
- `[SECTION LABEL_SharedMemory]` specifies shared memory blocks, and cached memory areas.
- `[SECTION LABEL_CachedMemoryForSpeed]` specifies memory areas to be cached by CodeScape when the target halts.
- `[SECTION LABEL_Settings]` specifies the default breakpoint type (hardware or software).

Each processor type is defined with a `SECTION LABEL`:

A section labelled:	Has information for this processor:
DASH MasterSH4EVA	7750 & 7751
DASH MasterSH3	7709a
DASH MasterSH3DSP	7729
DASH MasterSH2E	7055 & 7055MCM

Example

`[DASH MASTERSH3DSP_ValidMemory]` is the section heading that specifies for a 7729 processor the valid memory areas, access methods, and timing information.

Valid memory

Description

Specifies:

- Valid areas of memory for your code.
- Valid areas of memory to initialize RAM with a script. This is useful if you do not have a boot ROM and want to initialize the target's RAM with a script. An example script called examboot.js is provided on the CodeScape installation CD.

The available fields are:

- Access Method: Read, Write, ReadWrite, on-chip Flash.
- Start and End Address of the memory section being defined.
- Access Size: BYTE, WORD, LONG.
- Expression. Memory is valid when the expression evaluates to true.

Syntax

```
Access Method = Start Address, End Address, Size, [Expression],  
[Restrictions], [Byte, Word, Long, Quad, Cache, Byte, Word, Long,  
Quad, Cache]
```

Remarks

The Access Method, Start Address, End Address, and Size fields must be completed and contain valid values. The Expression, Restrictions, and simulator Write/Read fields are optional.

- Access Method can be: Read, Write, ReadWrite, or on-chip Flash.
On-chip Flash is only valid for 7055 and 7047.
- Size can be: BYTE, WORD, or LONG.
Size controls the size of commands sent to the target. So, if you display a byte where the memory access size is LONG, 4 bytes will be read by the debug stub, but only the byte required will be displayed.
- The expression “Expr” column is disabled for SH4.

Shared memory

Description

Marks specific mirrored memory images as shared memory.

Syntax

```
SharedMemory = Start Address, End Address, Tag:Label
```

Remarks

Software breakpoints cause exceptions during program execution if encountered in a memory area other than the one they were defined in. For example, if 0x0C000000 to 0x0D000000 is an image of 0x8C000000 to 0x8D000000 and a software breakpoint is inserted at address 0x0C00B000 then encountered at 0x8C00B000, CodeScape reports an unknown exception.

To avoid this, mark mirrored memory images as shared memory in the configuration file dali.cfg. The example file defines three areas of shared memory as the same (Tag:B). When the exception at 0x8C00B000 is executed it is then treated as a valid breakpoint.

This also affects labels marked as shared memory. Note that if two instances of shared memory also share the same label, for example, Start then the second instance also appears as Start.

Cached memory

Description

Marks specific memory areas to be cached by CodeScape when the target halts.

Syntax

```
CacheMemory = Start Address, End Address
```

Remarks

When the target is halted CodeScape can cache read from the specified memory area. This increases CodeScape's speed of operation and can make significant time savings when you are debugging.

For example, when the first variable in a Watch region is on the stack, CodeScape retrieves a memory block around this point. If the second variable in the Watch region is also on the stack, CodeScape can access the variable from its cache saving time by not having to communicate with the target again.

When you restart the target CodeScape discards the cached memory.

Software breakpoint settings

Description

Specifies either to use, or not use software breakpoints during tracing.

Syntax

```
AutoSoftBreakEnabled = 0
```

Remarks

AutoSoftBreakEnabled controls the use of soft breakpoints (insertion of ASE-Break op-code) during tracing. Set the value to 0 (default) to disable software breakpoints and set the value to 1 to enable them.

Debugging

Debugging

CodeScape's debugger

CodeScape's powerful source-level debugger has extensive software and hardware debugging features that let you:

- View the application you are debugging in various ways, including structure browsing and stack analysis.
- Trace your program at source level or instruction level. All trace operations immediately stop program execution and can be interrupted by breakpoints.
- Control breakpoint conditions and actions. Options include breaking on data accesses within memory ranges and on external peripheral access.

You can use debugger in one of two ways:

- As the debugging feature of CodeScape's Embedded Development Environment (EDE) where you can also manage and build your project.
- As a stand alone debugger that is associated with an external project management and build process.

Once you have set up CodeScape to debug you can create the regions you require to view your program file and start debugging.

Starting the debugger

To start the debugger you must do one of the following:

- Open a session that contains a project, or open a session and add a project to it. This method lets you use CodeScape's EDE to debug program files, and manage and build your project.
- Load a program file, or open a session that does not contain a project. This method lets you use CodeScape as a debugger only. You can associate the CodeScape debugger with an external project management and build process.

Before you can start debugging you must load the debug stub onto your target processor. You can select the debug stub (ASE or Extended) you want to load and set how it loads onto your target processor. For information on loading the debug stubs refer to the Getting Started Guide for your DASH.

Debug support is always reset to "enabled" when the DASH is powered up so that it can see the target. For information on testing your application without debug support see *"Testing your application without debug support" on page 163*.

Debugging with CodeScape's EDE

1. Run CodeScape.
2. Do one of the following:
 - Create a session and setup your project environment.
 - OR-
 - Open a session containing project information.
3. Click Debug, Start Debugging.

Until you start debugging the Edit region is the only one you can create. CodeScape's debugging features are enabled and the menu option changes to Stop Debugging.

When you stop debugging CodeScape's debugging features are disabled. Source regions become Edit regions and all other regions disappear. When you next start debugging the regions are restored.

For more information refer to *Creating Projects and Configuring the Interface*.

Debugging without CodeScape's EDE

1. On the command line type: `codescape -nomake`
CodeScape starts with the project management and build utility disabled.
If CodeScape is running, it ignores this option.
2. Load your program file to the target.

Testing your application without debug support

You may want to test your application by running it without support from the debugging tools: the CodeScape software and the DASH hardware. To do this you run the target with no debug support, this resets the target so that it boots either from ROM or from Flash. Where the target boots from depends on the processor you are using.

Run a target with no debug support

1. Click Tools, select Target / Communications then click Configure.
2. On the DA Start-up tab clear the Debug Support Enabled check box.
3. Click OK.

CodeScape scans the network for the target and when no target is found displays "OFFLINE" in the target window. In the Source region the message "No Target Available" appears.

The application continues to run on the target without support from the debugging tools until you re-enable debug support.

Re-enable debug support

1. On the Tools menu select Configure Target / Communications...
2. On the DA Start-up tab check the Debug Support Enabled check box.
3. Click OK.

CodeScape shuts down, scans the network for the target, then restarts and reloads the debug stub onto the target. You can now use CodeScape with full debug support.

If you are connected to an SH2e target (SH7055F and SH7055MCM) the Reflash tab appears on the Target Set-up dialog. This facilitates reflashing of the debug stub in the event that the stub has been corrupted or there is a version mismatch between the debug stub and the DASH firmware. Refer to SH2e Debug Interface for more information.

Region types

You can create different windows and regions that give you various views on the application you are debugging.

The views include:

- Target window.
- Input/Output window.
- Source and Disassembly regions.
- Call Stack region.
- Watch and Local Watch regions.
- Memory region.
- Register region.
- Edit region.

Set formatting and other options for these views in the Options dialog. Options include *Reuse Source Regions* that lets you reuse a single region (Source, Disassembly, Edit) when debugging would otherwise create multiple regions. For more information *refer to the Creating Projects and Configuring the Interface guide*.

Target window



When you run CodeScape it scans for valid targets and displays them in the Target window. The Target window shows all the targets that CodeScape is connected to and the processors available in each target. The processor status for each target is shown in the Target processor display. When you create a new window, CodeScape uses the target information from the selected processor on the target.

Using the shortcut menu on the Target window

Table 1: Controlling target processor execution

Use:	To:
Configure Processor...	Set the update rate for the current processor.
Execution	Run, stop, and restart your program. Run your program until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single step commands, or run the step commands.
Breakpoints	Add, enable, disable, configure, reset, or remove data breakpoints.
Reset Target	Do a soft reset or a hard reset. If you reset the target you are prompted to reload the program file.
Load Program File	Download a program file to the selected processor on the target.
Unload Program File Info	Clear all debug information and close the program file without removing the program file from target.
Restart	Restart the currently active program file. Restart loads the binary part of the current program file and resets the PC to the entry point (if known). If the program file's last modification time has changed, symbolic information is also loaded.
Allow Docking	Toggle docking of the window on or off.
Hide	Hide the window.

Target Processor display

- *To show the processor(s)* for a target, in the Target toolbar double-click on the target or click .
- *To hide the processor(s)* for a target, in the Target toolbar double-click on the target or click .

Input/Output window

The Input/Output window appears automatically and shows any relevant output on the Build, Scripts, Log, and Find In Files tabs. You can also open the window from the Toolbar Configuration dialog. The Input/Output window displays the:

- **Build tab** with the specified build utility's output when you build your project. Any standard format errors and warnings are shown in the Build tab. You can scroll through the information as it is generated, or press F4 to move through any listed errors one at a time.
The Edit region automatically opens your project file at the line containing the first error or warning. If there is no active Edit region CodeScape creates one. You can then use the Build tab to navigate to all subsequent errors.
- **Scripts tab** with all messages generated by the current script.
- **Log tab** with all messages generated by the current target. Note that text strings longer than 132 characters are truncated.
For example, you can use `printf ()` in your code to output a message to the Log region when a breakpoint triggers.
- **Find In Files tab** with a list of all files containing a search string at your specified location. Double-click an item in the list to examine the associated file.
To search for a string in multiple files, click Edit, Find In Files then enter your search requirements in the Find In Files dialog.

NOTE: *If you enable high level optimization when you build your project the compiler output can make source-level tracing confusing.*

Shortcut menus on the Input/Output window

The shortcut menu on the Build tab

Use:	To:
Make	Make your project current by building it.
Stop Make	Stop the active project build.
Rebuild All	Rebuild all project files.
Batch Build...	Select and build multiple project configurations set on the batch build dialog. <i>Refer to the Creating Projects and Configuring the Interface guide.</i>
Clean	Delete all intermediate files created during the build process, for example .OBJ files, as well as output files such as the .EXE or a .DLL.
Next Error	Move to the next error in the list.
Previous Error	Move to the previous error in the list.
Print...	Print the contents of the build tab.
Save To File...	Save the contents of the build tab to a file.
Clear	Clear the contents of the Project Build window.
Allow Docking	Toggle docking for the window on or off.
Hide	Hide the window.

The shortcut menu on the Scripts tab

Use:	To:
Run Script	Select and run a script.
Clear	Clear the contents of the Script tab.
Print...	Print the contents of the build tab.
Save To File...	Save the contents of the build tab to a file.
Allow Docking	Toggle docking for the window on or off.
Hide	Hide the window.

The shortcut menu on the Log tab

Use:	To:
Configure Log...	Configure the Log tab.
Print...	Print the contents of the Log tab.
Save To File...	Save the contents of the Log tab to a file.
Execution	Run, stop, and restart your program. Run your program to the cursor position, or until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Reset Log	Clear the contents of the Log tab.
Allow Docking	Toggle docking for the window on or off.
Hide	Hide the window.

The shortcut menu on the Find In Files tab

Use:	To:
Clear	Clear the contents of the Find In Files tab.
Next	Move to the next file in the list.
Previous	Move to the previous file in the list.
Allow Docking	Toggle docking for the window on or off.
Hide	Hide the window.

Source and Disassembly regions

The Source and Disassembly regions let you debug your code from different views.

- Disassembly regions are for debugging your program at instruction level (assembly code).
- Source regions are for debugging and editing your original source code.

When you edit your source code the changes appear in the corresponding Source region when the display is updated. A * appears in a Source region's title bar if you edit your source code and do not re-build the program file. Always save any changes that you make to a file edited in an external editor before using *Make* to compile and build your project.


Notes

1. *Place the mouse pointer over a variable or expression to quickly view its value.*
2. *If no debug information appears in a Source region, compile all source files in your project with debugging turned on.*
3. *Shared memory address information is only available in a Disassembly region and is used when looking up file, line, and symbol information.*
4. *If no source is available CodeScape shows the disassembly instead.*



Enable Automatic Source/Disassembly Switching

1. Click Tools, Options...
The Options dialog appears.
2. On the Interface tab select the *Automatic Source/Disassembly Switching* check box.
3. Click OK.

Lock the display origin to an expression

1. On the region title bar click .
The Lock View dialog appears.
2. Enter an expression for the region origin.
3. Click OK.

Synchronize the cursors in Source and Disassembly regions

1. In the Source region:
 - Right-click and click Synchronize Cursor.
 - OR-
 - On the Source region title bar click .
2. In the Disassembly region:
 - Right-click and click Synchronize Cursor.
 - OR-
 - On the Disassembly region title bar click .

The cursors for the Disassembly and Source regions are now synchronized. When you move the cursor in the region with focus, the cursor in the synchronized region shows the corresponding line of code.

NOTE: *You can only synchronize regions that are in the same window and are connected to the same target.*

Set or change the path for locating source files

In the Source File Search Path dialog you set, change, or remove a directory path name for CodeScape to look for source files in a source region.

Set the path for locating source files

1. Click Project, then click Edit Source Path...
The Source File Search Path dialog appears.
2. Do one of the following:
 - Type in the Path of the source files.
Click Add to enter the path in the Directories list.
 - OR-
 - Click Browse and select the required directory.
The path is automatically added to the Directories list.
3. Click OK.

Remove a path from the Source File Search Path

1. Click Project, then click Edit Source Path...
The Source File Search Path dialog box appears.
2. Select the path you want to remove from the Directories list.
3. Click Remove.

Using the shortcut menu in a Source region

The commands on the shortcut menu are for debugging in the region and configuring the source view. Right-click anywhere in the region to access the shortcut menu.

Configuring the Source view

Use:	To:
Cut	Cut the current selection in the Editor file and paste it to the Windows Clipboard.
Copy	Copy the current selection in the Editor file and paste it to the Windows Clipboard.
Paste	Insert the contents of the Windows Clipboard at the current cursor position.
Synchronize Cursor	Synchronize the cursors in Source and Disassembly regions to compare source and assembly code. Move the cursor in one region to see the cursor in the synchronized regions at the corresponding line of code.
Show Headers	Toggle the region's header bar on and off.
Show Address	Corresponding address for the first line of code generated by the source code line.
Show Line Nos.	Line numbers for each line of source in the left-hand column.
Start/Stop Debugging	Enables/disables CodeScape's debugging features. When enabled you can create regions you require to view the program you are debugging. Until you start debugging the Editor is the only region you can create.
Execution	Run, stop, and restart your program. Run your program to the cursor position, or until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Set Cursor to PC	Show source code from the value of the PC.
Set PC to Cursor	Change the PC to the current cursor position.
Goto Address...	Enter an expression for the region origin to display from.
Goto Source File...	Select the required source file. See <i>"Go to a source file referenced in the program file"</i> on page 173.
Display As	Display the program file in the active region as source code, or assembler code, or both source and assembler code.
Evaluate	Evaluate the selected value, symbol, or expression.

Use:	To:
Tabs...	Enter value to set the tab size in spaces.
Undo...	Undo the last action.
Redo...	Redo the last action.
Key Strokes	Record and playback keystrokes.
Find...	Search for a string. <i>See "Search in the Source region" on page 173.</i>
Find Next...	Search for the next instance of the string.
Replace...	Replace the current selection.
Go To Line...	Enter a line number to go to.
Bookmarks	Toggle a Bookmark on or off; move to the next, or previous Bookmark; or delete all Bookmarks.
Comment	Show/hide comments in the Edit region.
Properties	Set the update rate for the region and processor.
Syntax Highlighting	Turn syntax coloring on or off. Turn case sensitivity on or off. Specify a color for any or all of the following items in your code: keywords, quotes, comments, default text, and the background.

Go to a source file referenced in the program file

1. Right-click and click Goto Source File.
The List Files in Program File dialog appears.
2. Select the required source file.
3. Click OK.

If the path is incorrect an error message appears in the Source region. Click Project, then click Edit Source Path and enter the correct path for the source files.

NOTE: *Code is not generated for data-only files, or if the GNU -g command is not set when compiling. If code is not generated an error message appears.*

Search in the Source region

1. Right-click in the Source region, click Find.
The Find dialog appears.
2. Under *Find what* type the search string.
 - To search for whole words and not parts of a larger word, select the *Match whole word only* check box.
 - If the search is case sensitive, select the *Match case* check box.
3. Select the search *Direction* as Up or Down.
4. Click Find Next.
 - To continue the search to the next item click Find Next again.
 - To end the search and exit the Find dialog, click Cancel.
 - To continue searching for the most recent search item after exiting the Find dialog, right-click select Tools then click Find Next.

Using the shortcut menu in a Disassembly region

The commands on the shortcut menu are for debugging in the region and configuring the disassembly view. Right-click anywhere in the region to access the shortcut menu.

Configuring the disassembly view

Use:	To show:
Copy	Copy the current selection in the Editor file and paste it to the Windows Clipboard.
Synchronize Cursor	Synchronize the cursors in Source and Disassembly regions to compare source and assembly code. Move the cursor in one region to see the cursor in the synchronized regions at the corresponding line of code.
Source Annotation	Disassembled code with source-code annotations and symbols.
Show Headers	The header bar in the region.
Show Address	The location address of the disassembled code.
Show Labels	Symbolic label replacement of the disassembled code.
Show Opcode Words	Op-code in words for the disassembled region.
Show Hexadecimal	Operand values in hexadecimal.
Show Uppercase	Instructions in upper case.
Show Symbols	Operand values as symbols.
Show EAs & Lits	The effective address and literals.
Highlight Slotted Instructions	The value of a variable or expression by highlighting the line it appears on.
Execution	Run, stop, and restart your program. Run your program to the cursor position, or until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Set Cursor to PC	Show the source code from the value of the PC.
Set PC to Cursor	Change the PC at the current cursor position.
Goto Address...	Enter an expression for the region origin to go to.

Use:	To show:
Goto Source File...	Goto a source file referenced in a program file. If the path is incorrect an error message appears in the Source region. Code is not generated for data-only files, or if the GNU -g command is not set when compiling.
Display As	The program file in the active region as source code, or assembler code, or both source and assembler code.
Evaluate...	Evaluate a specific value, symbol, or expression.
Tools	Search in the Disassembly region. Repeat the last search run. Specify an address to disassemble to a file. See “Search in the Disassembly region” on page 176, and “Specify an address to disassemble to a file” on page 177.
Properties	Set the update rate for the region and processor.

Search in the Disassembly region

1. Right-click in the Disassembly region, point to Tools, then click Find.
2. Under *Find type* the Search string.
3. Under *Start Address type* the Start address.
4. If the search is case sensitive, select the *Case sensitive* check box.
5. Select one of the following radio buttons: Length (the amount of data), or End Address.
6. Type the search item in the text box below.
7. Select one of the following radio buttons: All fields (default), Words, Opcode, OpSrc, OpDest, or Label (address).
8. Click OK.

NOTE: *Right-click then click Find next to continue searching for the same item.*




Specify an address to disassemble to a file

This general purpose dialog is for writing a block of memory or disassembly in hexadecimal to a file.

1. Right-click in the Source region, point to Tools, then click Disassemble to File.
2. Under *Destination Filename* enter the name of the file to write to.
3. Under *Start Address* enter the start address in hexadecimal.
4. Do one of the following:
 - Select *Length* and enter the length in hexadecimal.
 - OR-
 - Select *End Address* and enter the end address in hexadecimal.
5. Click OK.

Call Stack region

Use the Call Stack region to view a list of active function calls. Viewing the call stack can help you trace the course of function execution. When the target stops, for example at a breakpoint, CodeScape displays the name, label, or address of the current function at the top of the list in the Call Stack region. Execution trace history is shown below the current function with its start point at the bottom of the list.

The cursor  shows the location of the PC. Double-click a function call in the list to synchronize it with its corresponding line of source. CodeScape highlights the function call with  as it occurs in active Source and Disassembly regions. Watch and Local Watch regions change accordingly. The cursor  remains at the current program position.

The shortcut menu in a Call Stack region

Use:	To click commands to:
Show Parameter Names	Toggle function parameter names on or off.
Show Parameter Types	Toggle function parameter types on or off.
Show Parameter Values	Toggle function parameter values on or off.
Show Parameter Registers/Stack	Toggle function parameter registers on or off.
Show Octal	Display function values in octal.
Show Decimal	Display function values in decimal.
Show Hexadecimal	Display function values in hexadecimal.
Auto Radix	Automatically view each item with the appropriate radix.
Execution	Run, stop, and restart your program. Run a program until it executes a specified address. Run all program files simultaneously. Stop all program files simultaneously. Use the single stepping commands, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Properties	Set the update rate for the region and processor.

Watch and Local Watch regions

The Watch and Local Watch regions display variables and expressions, one per row. Each row has four columns, the *Symbol* is in the first column and its:

- *Type* is in the second column.
- *Location* expression is in the third column.
- *Value* (if applicable) is in the fourth column.

In some instances the *Symbol* is preceded by:

- meaning that you can place a watch point on the expression.
- ⊕ meaning that you can expand the expression.
- ⊖ meaning that you can collapse the expression.

In a Watch or Local Watch region, you can:

- Highlight changes in data values between execution steps.
- Edit the value of an expression in the Watch region and the Local Watch region.

NOTE: *You can only edit the actual expression in a Watch region.*

C++ name demangling in a Watch or Local Watch region

C++ name de-mangling is performed on all variable names. This means that you can enter the symbol for a name as it appears in your original source. All C types are supported including structs/classes, unions, arrays, enumeration (enum), float/double, bool, and references.

You can browse data to:

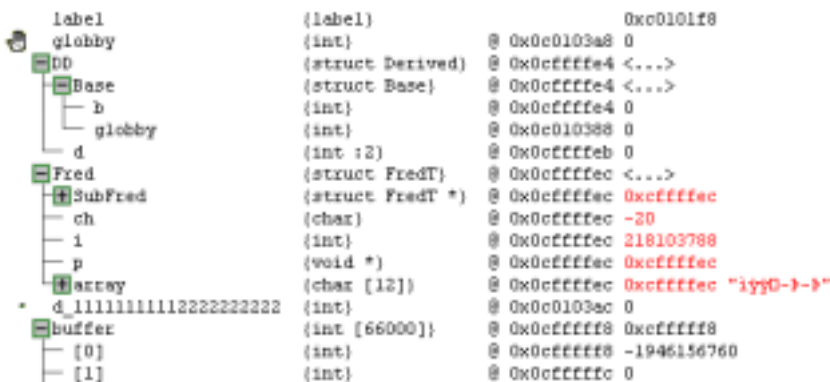
- Expand and collapse branches of the hierarchical tree view of the structure.
- See exactly where the structures are in memory.
- Edit the values of any variables.

If you place a watch (data) breakpoint on a member of a union it will trigger for all members of that size, regardless of type. This also applies to anonymous unions, except that two members of the same size appear as two variables sharing the same address in memory.

Expanding expressions

When you expand an expression, each child expression is shown in the tree directly below the parent.

For example:



Expressions are added to expanded:

- Pointers, to show the dereferenced item.
- Arrays. An expression is added for each element of the array.
- Structures. An expression is added for each member.

Watch region

In the Watch region you can enter variables and expressions. The scope of variables in a Watch region is global. If an expression goes out of scope during program execution, a message appears.

- If an expression's value can be determined, as is the case for a static variable, its value is shown in the region.
- If an expression's value cannot be determined, no value is shown.
- If an expression comes back into scope, its value is shown.

The shortcut menu in a Watch region

Use:	To:
Copy	Copy the current selection and paste it to the Windows Clipboard.
Paste	Insert the contents of the Windows Clipboard at the current cursor position.
Delete	Delete an entry.
Open	Expand a structure, pointer, or array.
Close	Collapse a structure, pointer, or array.
Insert	Insert a new watch expression.
Append	Add a variable to the end of the active list.
Promote to Root	Create an alias of the current selection. The alias is automatically appended to the root of the Watch tree and appears as the last item in the list at that point. Note that this only applies to expandable elements such as structures and arrays.
Column Header Settings	Toggle the following on or off: header bar, column separators, location column, type column, keep the cursor row expression in view.
Radix	Display watch expressions in: octal, decimal, hexadecimal, or tell CodeScape to automatically use the appropriate radix for viewing each item. Display the selected watch expression in: octal, decimal, hexadecimal, or use the region's default radix.
Edit Watch Value...	Modify the value of a variable or watch expression.
Execution	Run, stop, and restart your program. Run your program until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run the step commands.

Use:	To:
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Highlight Changes	See where an expression changed.
Cache Expanded Symbols	Save the expanded state of symbols. Each time a scope with saved expanded symbol information is entered the symbol automatically expands.
Properties	Set the update rate for the region and processor.


Browsing data in a Watch region

Add a symbol or variable

- Right-click, click Insert to add a symbol or variable at the cursor position.
-OR-
- Right-click, click Append to add symbol or a variable at the end of the list of variables.
-OR-
- Press ENTER to enter a new symbol or variable at the current cursor position.


Expand a structure or array

Select the structure or array you want to expand, then:

- Click on .
- OR-
- Press SPACEBAR.
- OR-
- Right-click and click Open/Close.

Collapse a structure or array

Select the structure or array you want to collapse, then:

- Click on .
- OR-
- Press SPACEBAR.
- OR-
- Right-click and click Open/Close.

Editing variables in a Watch region

Edit a variable's data value (structure, array, or union)

1. Double-click the value to be edited.
2. Edit the value.
3. Do one of the following:
 - Press ENTER.
 - OR-
 - Right-click then click Edit Watch Value...
The Expression Evaluator dialog appears. Enter a valid expression. Click OK.

Delete a parent expression and all child expressions

1. Expand the structure or array.
2. Select the component you want to delete, then:
 - Right click and click Delete.
 - OR-
 - Press DELETE.

Local Watch region

The Local Watch region automatically displays all local variables in view from the current position of the PC. Variables are automatically added to the display as they come into the scope of a function.

Notes

1. *If there is more than one variable of the same name in the current scope, all but the inner most variable of that name are unavailable.*
2. *If there is more than one variable of the same name in the same scope they are shown in italics.*


The shortcut menu in a Local Watch region

Use:	To:
Copy	Copy the current selection and paste it to the Windows Clipboard.
Delete	Delete the current selection.
Open	Expand a structure, pointer, or array.
Close	Collapse a structure, pointer, or array.
Column Header Settings	Toggle the following on or off: header bar, column separators, location column, types column.
Radix	Display watch expressions in: octal, decimal, hexadecimal, or tell CodeScape to automatically use the appropriate radix for viewing each item. Display the selected watch expression in: octal, decimal, hexadecimal, or use the region's default radix.
Edit Local Value	Modify the value of a variable or watch expression.
Execution	Run, stop, and restart your program. Run your program until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Highlight Changes	See where in memory an expression changed.
Cache Expanded Symbols	Save the expanded state of symbols. Each time a scope with saved expanded symbol information is entered the symbol automatically expands.
Properties	Set the update rate for the region and processor.

Browsing data in Local Watch region


Expand a structure or array

Select the structure or array you want to expand, then:

- Click on .
- -OR-
- Press SPACEBAR.
- -OR-
- Right-click and click Open/Close.

Collapse a structure or array

Select the structure or array you want to collapse, then:

- Click on .
- -OR-
- Press SPACEBAR.
- -OR-
- Right-click and click Open/Close.

Editing variables in a Local Watch region

Modify the value of a variable or watch expression

1. Do one of the following:
 - Select the value to be changed. Press ENTER.
 - OR-
 - Right-click then click Edit Local Value...
The Expression Evaluator dialog appears.
2. Enter a valid expression.
3. Click OK.

Delete a parent expression and all children

1. Expand the structure or array.
2. Select the component you want to delete, then:
 - Right click and click Delete.
 - OR-
 - Press DELETE.

NOTE: *If you delete a parent expression, any child expressions are also removed from the region.*

Memory region

Use the Memory region to view the target's memory contents from a specific address. You can view memory as ASCII characters, Bytes, Words, or Longs. Write protect an area of memory to prevent memory contents changing in the current memory region. As you scroll through a Memory window the cursor moves a line at a time and the slider speed increases. The slider automatically returns to the center position when you stop scrolling. Double-click a variable or expression to quickly view and edit its value.

The shortcut menu in a Memory region


Use:	To click commands to:
Byte	Display memory as bytes.
Bytes (Words)	Display memory as words.
Bytes (Longs)	Display memory as longs.
Bytes (Quadwords)	Display memory as quadwords.
Display Format	Show memory as: hexadecimal values, decimal numbers without indicating if they are positive or negative, decimal numbers and indicate if they are positive or negative, floats, or doubles. Only show the sign for negative numbers (and not positive numbers) when the display is set to Signed Decimal. Show leading zeros when the display is set to Unsigned Decimal or Signed Decimal. Display: the ASCII value for each byte memory, MBCS, and nudge the MBCS decoding origin.
Highlight Changes	See where the target's memory changed.
Set Bytes Per Line...	Display a specific number of bytes per line.
Edit ASCII	Change an ASCII value in the Memory region.
Edit Memory Value	Change a value in the Memory region.
Follow Pointer (multi level)	Follow a pointer in memory.
Goto Address...	Set the origin.
Write Protect	Toggle write protect.
Change Inc/Dec Value...	Change the increment value.

Use:	To click commands to:
Execution	Run, stop, and restart your program. Run your program to the cursor position, or until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single step, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Tools	Search for a pattern in memory. Repeat the last search. Fill a range of memory with data. Write a block of memory in hexadecimal to a file.
Properties	Set the update rate for the region and processor.

Viewing Memory

View Memory regions

Do one of the following:

- Click Region, point to Type and click Memory.
-OR-
- On the Region toolbar, click .
-OR-
- In any region, CTRL+Right-click and click Memory.

Set the origin

1. Right-click and click Goto Address...
The Goto Address... dialog appears.
2. Enter an address or symbol for the new origin.
3. If you enter an invalid symbol a warning appears with a command to invoke the Origin dialog.
4. Click OK.

NOTE: *The origin is initially set to the value of the PC. You can also set the origin to an expression.*

Always display memory from a specified address

1. Do one of the following:
 - Click Edit, then click Go To...
 - OR-
 - Right-click and click Goto Address...
The Goto Address... dialog appears.
2. Type or select a memory location or expression from the Expression list.
3. Select Lock, click OK.

Follow a pointer in memory

1. Select the memory location holding the value of the pointer.
2. Right-click and click Follow Pointer.
The Memory region origin changes to display memory from the location specified by the value of the pointer.

Write a block of memory in hexadecimal to file

1. In the Memory region, right click, point to Tools and then click Hex Dump to File.
The Hex Dump to File dialog appears.
2. Under *Destination Filename* enter the name of the file to write to.
3. Under *Start Address* enter the start address in hexadecimal.
4. Then select:
 - *Length* and enter the length of the memory block in hexadecimal.
 - OR-
 - *End Address* and enter the end address in hexadecimal.
5. Click OK.
The specified block of memory is written to a file.

Editing memory

Change byte, word, or long values in the Memory region

- Use the + and - keys to increment or decrement the current value.
-OR-
- Double-click or press ENTER, then type over the existing value.
-OR-
- Right-click, click Edit memory value...
The Expression Evaluator dialog appears. Enter a valid expression then click OK.

NOTE: *Make sure you enter valid values for the radix. CodeScape displays all Memory values in hexadecimal by default but you can change the display format on the region's shortcut menu.*

Filling memory with specific data

1. In a Memory region, right-click and select Display Format then click Display as Hexadecimal to view the region in hexadecimal.
2. Tools, click Fill...
3. Under *Fill Expression* enter a value.
4. Under *Start Address* enter a value.
5. Select *End address*, or *Length*.
6. Enter a value in the text box below.
7. Select the Mode as Text (ASCII), Byte, Word, Long, or Quad.
8. Do one of the following:
 - Select *Convert Native Endian*, to show the real memory value.
-OR-
 - Deselect *Convert Native Endian*, to store memory as byte sequences.
9. Click OK.

Searching memory

To define and search an area of memory for a specified pattern of data:

1. In a Memory region, right-click and select Tools...
2. Click Find.
The Find In Memory dialog appears.
3. Under *Find Pattern* enter a search string.
In Binary, Octal, Decimal, and Hex modes the search pattern is delimited by either commas or semi-colons (optional).
4. Under *Start Address* enter a value.
The default value is the start address of the region. If you have not run a search, the address at the start of the current memory block is used.
5. Select End Address, or Length.
The default value for the end address is the last displayed byte in the Memory region.
6. Enter a value in the text box below.
7. Select the Width as either Byte, Word, or Long.
8. Select Forward or Reverse to specify the direction of the search.
9. Click OK.

NOTE: *If a specified search is not valid, 'Invalid Address' appears in the field(s) that require editing.*

NOTE: *If a match is found its address appears. A search skips over any sensitive areas such as invalid memory areas, write-only memory, and memory reserved for the monitors.*

Width

Width aligns the search pattern with the data in the target's memory. This specifies how the search pattern and the memory contents are compared. The allowable width depends on the search mode selected.

Valid mode and width combinations

For this mode:	Valid widths are:
Binary	Byte, Word, Long, Quad
Decimal	Byte, Word, Long, Quad
Hex	Byte, Word, Long, Quad
Text	N/A

These patterns are equivalent with Hex and Byte widths set:

FF, FF, FF, FF, 34, DC
FF; FF; FF; FF; 34; DC
FFFFFFFF34DC

The \ specifier

Use the \ specifier to include special characters in text searches.

NOTE: *Always enclose a text search string in quotes.*

Example

The pattern "How are you\?" searches for "How are you?"

The ? wildcard

The wild card character '?' can be used in Binary and Hex modes. Use '?' to specify a nibble in Hex mode and a bit in Binary mode that always results in a successful match.

Examples

In Hex mode: FF?F matches FF0F,FF1F,FF2F,...FFFF

In Binary mode: ???1111 matches 00001111,00011111,...11111111

The @ wildcard

The wild card character, '@', can be used in Text modes. '@' specifies a double-byte character.

Automatic padding

The search pattern is automatically left-padded for the Binary, Decimal, and Hex modes. The padding type is either '0' or '?' depending on the delimiter used.

Delimit with commas

Delimit a search pattern with commas (the default) to left-pad it with zeros.

For example, in Hex mode with Byte width:

FFFFFFFF34DC and FF,FF,FF,FF,34,DC do the same search.

The comma separator in is implied in the first search pattern. More examples, in Hex mode and Word width, are:

f,87d,a automatically pads to 000F,087D, 000A

f87da automatically pads to 000F,87DA

, automatically pads to 0000

NOTE: *A single comma used on its own produces the pattern 0000. Use this feature carefully. For example: ,7 automatically pads to 0000,0007.*

Delimit with semi-colons

Delimit a search pattern with semi-colons to pad it with the '?' wild card. Examples, in Hex mode and Word width, are:

f;87d;a automatically pads to ???F,?87D,???A

f;87da automatically pads to ???F,87DA

; automatically pads to ????

Equivalent search patterns

Use the comma and semi-colon delimiters to do the same search pattern in different ways. The following patterns are the same when the Hex and Byte widths are set:

FF,FF,FF,FF,34,DC

FF;FF;FF;FF;34;DC

FFFFFFFF34DC

You can mix the comma and semi-colon delimiters to produce precise search patterns.

For example, in Hex mode and Long width:

f;f0f0f0f0,ffffff?,7; pads to: ???????F,F0F0F0F0,FFFFFFF?,???????7

Register region

The Register region shows the contents of a processor's register block and flags. It shows the value of the SEA (Source Effective Address) and DEA (Destination Effective Address). The SEA and DEA show the source effective address (read from) on the left-hand side and the contents of that address (write to) on the right-hand side.

The shortcut menu in a Register region

Use:	To click commands to:
Increment Register	Apply the current Increment Value (1 is the default) to the contents of the register.
Decrement Register	Apply the current Decrement Value (1 is the default) to the contents of the register.
Change Inc/Dec Value...	Change the Increment/Decrement Value.
Highlight Changes	See where changes occurred during the last operation.
Write Protect	To prevent data from being written to the currently active Register region.
Edit Register	Change the selected register value.
Column Format	Display registers in two, or four columns. Select Auto to tell CodeScope to choose.
Show Banked Registers	Toggle the banked register display on or off.
Show Float Registers	Toggle the floating point register display on or off.
Show Contents at SEA/DEA	Toggle the display of values of the SEA and the DEA. The SEA and DEA show the source effective address (read from) on the left-hand side and the contents of that address (write to) on the right-hand side.
Show Float Registers As Hexadecimal	Toggle the floating point register display between decimal and hexadecimal.
Execution	Run, stop, and restart your program. Run your program to the cursor position, or until it executes a specified address. Run all of your program files simultaneously. Stop all of your programs running simultaneously. Use the single stepping commands, or run the step commands.
Breakpoints	Toggle a breakpoint on or off. Enable, disable, configure, reset, and remove breakpoints.
Tools	Save the current Register Block. Restore the last saved Register Block.
Properties	Set the update rate for the region and processor.

Change the display format

The registers are displayed in the available area by default. You can set the display to two or four columns.

Display registers in two columns

- Right-click, point to Column Format then click 2 Columns.

Display registers in four columns

- Right-click, point to Column Format then click 4 Columns.

Display registers in the available area

- Right-click, point to Column Format then click Auto Format.

Change the value of a register

1. Move the cursor to the register value you want to change.
2. Do one of the following:
 - Use + or - to increment or decrement the current value.
-OR-
 - Type the new value at the cursor.
-OR-
 - Double-click a register, type a value or expression, press ENTER.
The Register Evaluator dialog appears.
-OR-
 - Right-click then click Edit Register... The Register Evaluator dialog appears.

NOTE: *Any alphanumeric characters appear in upper case.*

Change the Increment/Decrement Value

1. Right-click and click Change Inc/Dec Value...
The Register Increment/Decrement dialog appears.
2. Type a value for the amount by which to increment or decrement a register.
3. Click OK.

Write protect a register

1. Right-click in the region.
2. Then do one of the following:
 - If Write protect is not selected, click Write Protect.
 - OR-
 - If Write protect is selected, exit the shortcut menu.

NOTE: *Write protect only stops register values being changed for a specific region. If you edit another Register region, the changes appear in the write protected region.*

Enter an expression for the instruction at the PC

1. Double-click the register value, then:
 - Enter the expression. Press ENTER.
 - OR-
 - Right-click and click Edit Register...
The Register Evaluation dialog appears.
2. Enter the expression.
3. Click OK.

NOTE: *If you enter an invalid expression, the Register Evaluation dialog appears showing the Invalid Register Expression.*

Highlight recently changed attributes

Recently changed attributes are shown briefly in red (default).

To use another color to highlight a changed attribute, do the following:

1. Click Tools, Options.
2. Select the Format tab.
3. Under *Targets* select the target to apply the attribute changes to.
4. Under *Category* select the region or tool to apply the attribute changes to.
5. Under *Color* select Highlight Data Changed.
6. On the Foreground drop-down select the color you want to use.
7. Click OK.

NOTE: *You cannot highlight changed attributes by setting a different Background color.*

Save the state of the registers

Right-click, point to Tools, then click Save Register.

NOTE: *The register states for each target processor are saved one at a time and cannot be stacked. The state of the registers is stored internally to CodeScape, not in a file.*

Retrieve the state of the registers

Right-click, point to Tools then click Restore Register.

SuperH target processor register display

The Register region shows the registers for the following Hitachi SuperH processors: 7047F, 7055, 7615, 7750, 7751, 7729R, 7727, 7729, and 7709A.

The basic software-accessible registers are divided into distinct groups. The three register groups common to all supported SuperH processors are: general registers, control registers, and system registers. Apart from the 7709a each processor has a fourth register group; either DSP registers, or floating-point registers.

- DSP registers are the fourth group for: 7615, 7629R, 7727, and 7729.
- Floating-point registers are the fourth group for: 7055, 7750, and 7751.

For more detailed information about the registers for any of the supported processors refer to the relevant Hitachi programming manual.

General registers

The Register region shows the values of the SuperH 16 general registers (Rn) numbered R0-R15.

R0 acts as a fixed source or destination register in some instructions, and as an index register in:

- Indirect indexed register addressing mode.
- Indirect indexed GBR addressing mode.

R14 works as the frame pointer during debugging.

R15 works as a hardware stack pointer (SP) during exception processing.

Control registers

The Register region shows the values of the control registers. The control registers are different for each processor, but the following are common to all: SR (Status Register), GBR (Global Base Register), and VBR (Vector Base Register). Additional registers for the:

- 7615 are Repeat Start Register (RS), Repeat End Register (RE), and Modulo Register (MOD).
- 7750 are Saved Status Register (SSR), Saved Program Counter (SPC), Saved General Register (SGR), and Debug Base Register (DBR).
- 7727, 7729R, and 7729 are Saved Status Register (SSR), Saved Program Counter (SPC), Repeat Start Register (RS), Repeat End Register (RE), and Modulo Register (MOD).
- 7751, and the 7709a are Saved Status Register (SSR) and Saved Program Counter (SPC).

The status register flags indicate processing states. The list below describes the flags and their values for all of the processors *Table , page -202* shows the flags that can be set for each processor.

- MD: Processor mode, set to 0 is user mode and set to 1 is privileged mode.
- RB: General register bank specifier in privileged mode (set to 1 by a reset, exception, or interrupt).
- BL: Exception/interrupt block bit (set to 1 by a reset, exception, or interrupt).
- RC[11:0]: 12-bit Repeat Counter.
- DSP bit: DSP operation mode, set to 1 and DSP instructions are enabled, set to 0 and DSP instructions are treated as illegal instructions, only SH3 instructions are supported.
- DMYbit: Modulo addressing enable for Y side.
- DMXbit: Modulo addressing enable for X side.
- FD: FPU disable bit (cleared to 0 by a reset). Set 1 and an FPU instruction causes a general FPU disable exception, and if the FPU instruction is in a delay slot, a slot FPU disable exception is generated. (FPU instructions: H'F*** instructions, LDC(.L)/STS(.L) instructions for FPUL/FPSCR).
- M, Q: Used by the DIV0S, DIV0U, and DIV1 instructions.
- IMASK: Interrupt mask level
External interrupts of a lower level than IMASK are masked.

- RF[1:0]: Used for Repeat Control.
- S: Specifies a saturation operation for a MAC instruction.
- T: True/false condition or carry/borrow bit.
- Reserved bits: These bits always read 0, and the write value should always be 0.

Status register flags supported by each processor

Flag	Processor							
	7047F	7055F/ MCM	7615	7750	7751	7727	7729/R	7709A
MD								
RB								
BL								
RC								
DSPbit								
DMYbit								
DMXbit								
FD								
M, Q								
IMASK								
RF								
S								
T								
Reserved								

Key

	Indicates that the processor supports the flag.
	Indicates that the processor does not support the flag.

Changing the value of a status register

1. Move the cursor to the register value you want to change.
2. Do one of the following:
 - Use + or - to increment or decrement the current value.
-OR-
 - Type the new value at the cursor.
-OR-
 - Right-click then click Edit Register...
The Register Evaluator dialog appears.

NOTE: *Any alphanumeric characters appear in upper case.*

Setting the status register (SR, SSR, or FPSR) flags

1. Move the cursor to the flag you want to set.
2. Then:
 - Press 1 to set the current flag.
The bit's flag appears in upper-case.
-OR-
 - Press 0 to clear the current flag.
The bit's flag appears in lower-case.

NOTE: *Press SPACEBAR to toggle the status registers.*

System registers

The Register region displays the MACH and MACL (high and low multiply and accumulate registers), PR (Procedure Register), and PC (Program Counter). The MACH and MACL registers store the results of multiply and accumulate operations. The PR stores a return address from a subroutine procedure.

DSP registers

The processors 7622, 7729R, 7727, and 7729 all have eight DSP data registers and one DSP control register. The data registers are 32-bit width with the exception of registers A0 and A1. Registers A0 and A1 have guard bits.

Floating-point registers

The 7055 has sixteen 32-bit floating-point registers, FR0 to FR15, FR0 is the index register for the FMAC instruction. It also has two 32-bit floating-point system registers: the floating-point communication register (FPUL) and the floating-point status/control register (FPSCR).

The 7750, and the 7751 each have thirty-two floating-point registers, FR0 to FR15 and XF0 to XF15 that can be assigned to one of two banks: either FPR0_BANK0 to FPR15_BANK0, or FPR0_BANK1 to FPR15_BANK1.

7750 and 7751 floating-point exceptions

The floating point exception handler for the 7750 and the 7751 needs software assistance when the V, O, U, or I bits are enabled in the **FPSCR.ENABLE** field. An FPU exception is raised for many of the floating point op-codes including fadd, fsub, and fmul, regardless of whether an exception occurs.

When any of these bits are set, the target processor can stop with an exception on a floating point instruction, even if you set safe values. For example, fmul R4, R5 where R4=1.5, and R5=1.5.

The debug stub (v2.8.0a onwards) analyzes FPU exceptions to find out if they are valid.

- If an Invalid Floating-Point Exception (Invalid FPU) occurs, it is handled by the debug stub and a large loss of processor performance is incurred (several hundred clocks).
- If a Valid Floating-Point Exception (Valid FPU) occurs, it is handled by CodeScape and an even greater loss of processor performance is incurred.

When a Valid FPU occurs, CodeScape displays a status message describing the type of exception that caused it in the Target window. The status messages are: FPU error (E), Invalid operation (V), Divide by Zero (Z), Overflow (O), Underflow (U), and Inexact (I).

Notes

1. *The Debug Stub passes FPU instructions executed in slots to CodeScape. If an unwanted slotted exception occurs, CodeScape traces around it, then resumes running the program.*
2. *It is recommended that the floating-point enable bits are not used when program performance is needed.*
3. *For more information about SH4 floating-point exceptions, refer to the Hitachi SH7091 Programmer's Manual.*

Edit region

In an Edit region you can manage, edit, and print source files. When you edit your source code the changes are displayed in the corresponding Source region when the display is updated. A * appears in the Source region's title bar if you edit your source code and do not re-build the program file. Always save any changes that you make to a file edited in an external editor before using the Make option to compile and build your project in CodeScape.

Any standard format errors and warnings are shown in the Project Build window. You can scroll through the information as it is generated, or press F4 to move through any listed errors one at a time. CodeScape's Edit region automatically opens your project file at the line containing the first error or warning. You can then use the Project Build window to navigate to all subsequent errors. If there is no active Edit region CodeScape creates one for you.

Notes

1. *To configure the appearance, behavior, and language options in an Edit region click Tools, Options... then select the Editor tab.*
2. *For more information refer to Configuring Projects and Using the Interface.*
3. *If you build your project in CodeScape and edit your source in an external editor you can use the line error information on the Build tab to manually locate the error in your source.*

The shortcut menu in an Edit region

Use:	To click commands to:
Cut	Cut the current selection in the Editor file and paste it to the Windows Clipboard.
Copy	Copy the current selection in the Editor file and paste it to the Windows Clipboard.
Paste	Insert the contents of the Windows Clipboard at the current cursor position.
Evaluate	Evaluate a specific value, symbol, or expression.
Tabs...	Enter a new tab value.
Undo...	Undo the last action.
Redo...	Redo the last action.
Key Strokes	Record and playback keystrokes.
Find...	Search for a string.

Use:	To click commands to:
Find Next...	Search for the next instance of the string.
Replace...	Replace the current selection.
Go To Line...	Enter a line number to go to.
Bookmarks	Toggle a Bookmark on or off; move to the next, or previous Bookmark; or delete all Bookmarks.
Comment	Show/hide comments in the Edit region.
Properties	Set the update rate for the region and processor.
Syntax Highlighting	Turn syntax coloring on or off. Turn case sensitivity on or off. Specify a color for any or all of the following items in your code: keywords, quotes, comments, default text, and the background.

Opening and saving files

Open an existing file

1. Click File, Open...
The Open dialog appears.
2. Select the required file.
3. Click Open.

Open a new file

1. Click File, New...
The New dialog appears.
2. Select the required file type.
3. Specify the Filename and Location.
4. Select Add To Project to add the file to the active project.
This option is only available when a session is open in CodeScape.
5. Click OK.

Save a file

- Click File, Save

Save all files

- Click File, Save All

Save a file with a new name

1. Click File, Save As...
The Save As dialog appears.
2. Enter a new name for the file.
3. Click Save.

NOTE: *If you save an un-named file, the File Save As dialog appears.*

Search and replace

Search a file

1. Move the cursor to where you want to start searching from.
2. Do one of the following:
 - Click Edit, then click Find...
 - OR-
 - Right-click, click Find...The Find dialog appears.
3. Under *Find What* enter the search string.
4. Under *Direction* select Up or Down.
5. Select any of the following options:
 - *Match whole word only* to find only the exact search word and not words that contain your search. For example, if you enter the word “fred” CodeScape will return only instances of “fred” and not other words such as “freddie”.
 - *Match case* to find only strings that match the case of the characters in your search string exactly.
 - *Regular expression* if you entered a regular expression under *Find what*.
6. Do one of the following:
 - Click Find Next to continue searching.
 - OR-
 - Click Mark All to add a bookmark to all lines containing your search string.

Replace a string

1. Move the cursor to where you want to start replacing from.
2. Do one of the following:
 - Click Edit, then click Replace...
 - OR-
 - Right-click, click Replace...The Replace dialog appears.
3. Under *Find what* enter the search string.
4. Under *Replace with* enter the new string.
5. Select any of the following options:
 - *Match whole word only* to find only the exact search word and not words that contain your search. For example, if you enter the word “fred” CodeScape will return only instances of “fred” and not other words such as “freddie”.
 - *Match case* to find only strings that match the case of the characters in your search string exactly.
 - *Regular expression* if you entered a regular expression under *Find what*.
6. In the Replace In field, select:
 - Selection.
You must have some text selected in the Editor for this option to be available.
 - OR-
 - Whole file.
7. Do one of the following:
 - Click Find Next to continue searching without replacing a found item.
 - OR-
 - Click Replace to replace the first instance of your search string.
 - OR-
 - Click Replace All to replace all instances of your search string.

Copy, cut, and paste text

Copy text

1. Select the text you want to copy by highlighting it.
2. Click Edit, then click Copy.
3. Put cursor where you want to paste the information.
4. Click Edit, then click Paste.
The information is copied from its original location and appears in its new location.

Move text

1. Select the text that you want to move by highlighting it.
2. Click Edit, then click Cut.
3. Put cursor where you want to paste the information.
4. Click Edit, then click Paste.
The text is removed from the original location and appears in its new location.

Record and play back Key Strokes

Recording Key Strokes

You can automate repetitive keyboard tasks by recording and playing back Key Strokes. The Play Back feature is available until you record a new set of Key Strokes or until you close the active Editor region.

If you create a new Editor region while recording keystrokes, the recorder will remain in record mode and continue recording Key Strokes, but only in the Editor region it was activated in. The recorded Key Strokes will only play back into the region they were recorded in.

1. Right-click in the Edit region, select Key Strokes and click Start Recording.
2. Move the cursor to where you want to begin typing and enter the Key Strokes that you want.
3. Right-click in the Edit region, select Key Strokes and click Stop Recording when you have finished recording your keystrokes.

Playing back Key Strokes

1. Move the mouse pointer to where you want to play back the recorded Key Strokes.
2. Right-click in the Edit region, select Key Strokes and click Play Back.

The recorded keystrokes are played back into the active Editor region at the selected location.

Using bookmarks

You can set bookmarks to mark frequently accessed lines in your source file. Bookmarks are removed when the file containing them is closed or reloaded. Bookmarks store only the current line, not the column offset of the cursor. When a line containing a bookmark is deleted, the bookmark is also deleted.

Set a bookmark

1. Move the cursor to the line where you want to set a bookmark.
2. Right-click, select Bookmarks then click Toggle.
An indicator appears in the margin next to the text.

Set a bookmark at all lines that contain a specific string

1. Right-click, then click Find.
2. Under *Find what* enter the search string.
3. Click Mark All.
An indicator appears in the margin of each line that contains the specified string.

Delete a bookmark

1. Move the cursor to the line containing the bookmark you want to delete.
2. Right-click, select Bookmarks then click Toggle.
The indicator disappears from the margin next to the text.

Delete all bookmarks

- Right-click, select Bookmarks then click Delete All.

Move to the Book Mark after the cursor

- Right-click, select Bookmarks then click Next.

Move to the Book Mark before the cursor

- Right-click, select Bookmarks then click Previous.

Running and stopping programs


Running and stopping programs

The run options are:

- Run all processors simultaneously, and stop all processors simultaneously.
- Run a program, run a program until it executes a specified address, and run a program to the cursor position.
- Stop a program.

Run all processors simultaneously


Do one of the following:

- Click Debug, point to Execution then click Run All.
-OR-
- Right-click, point to Execution then click Run All.
-OR-
- On the Debug toolbar, click .

NOTE: *Program execution will stop at a breakpoint, or if an error occurs.*


Stop all processors simultaneously

To stop all programs running simultaneously, do one of the following:

- Right-click in the Target window, point to Execution then click Stop All.
-OR-
- On the Debug toolbar, click .
- OR-
- Click Debug, point to Execution then click Stop All.


NOTE: All processors will stop immediately.

Run a program

1. In the Target region, select the processor where your program is loaded.
2. Do one of the following:
 - Click Debug, then click Run.
 - OR-
 - Right-click, point to Execution then click Run.
 - OR-
 - On the Debug toolbar, click .

NOTE: *Program execution will run until you stop it, or if an error occurs.*

Run a program until it executes a specified address

1. Do one of the following:
 - Click Debug, point to Execution then click Run to Address...
 - OR-
 - Right-click, point to Execution then click Run to Address...
 - OR-
 - On the Debug toolbar, click .

The Run to Address/Instructions dialog appears.


2. Under *Expression* enter an address.
3. Click OK.

The address is the name of a function or an expression that resolves to an address.

You can add breakpoints using Run to Address at any time during program execution, program execution stops when a breakpoint is encountered. Program execution will stop at the specified address, or at a breakpoint or if an error occurs.

Run a program to the cursor position

In an active Source or Disassembly region, do one of the following:


- Click Debug, point to Execution then click Run to Cursor.
-OR-
- Right-click, point to Execution then click Run to Cursor.
-OR-
- On the Debug toolbar, click .

You can add breakpoints using Run to Cursor at any time during program execution, program execution stops when a breakpoint is encountered. Program execution will stop at the cursor position, or at a breakpoint, or if an error occurs.

NOTE: *In the Call Stack region, use Run to Cursor to return to a specific function outside of the current one.*

Stop a program

To stop a program running, in the Target region, select the processor on which your program is loaded, then do one of the following:

- Right-click in the Target window, point to Execution then click Stop.
-OR-
- On the Debug toolbar, click .
- Click Debug, point to Execution then click Stop.

NOTE: *The processor will stop immediately.*

Stepping into (tracing) code

Trace functions are available either on the debug toolbar or on shortcut keys.

You can choose either source level tracing in an active Source region, or instruction level in an active Disassembly region. If you trace without a Source or Disassembly region open, tracing behaves as if a Disassembly region is open.

You can trace a program at any time when debugging. All trace operations immediately stop program execution. All trace operations can be interrupted by breakpoints.

The trace options are:


- Single step a line of source code.
- Force step a line of source code at the disassembly level.
- Step over a line of source code.
- Step out of a line of source or disassembled code (return from function).
- Undo a step.
- Enable Animated Step Run (animate trace).
- Step run into a line of source or disassembled code.
- Step run out of a line of source or disassembled code.
- Step Run Until.
- Restart processor execution.
- Stop a program.

Single step a line of source code

Execution trace history is generated when single stepping.

- Click Debug, point to Execution then click Step.

-OR-

- On the Debug toolbar click .

In an active Disassembly/any non-source region, the target executes the instruction at the PC.

In an active Source region, the target executes the instruction at the PC. It stops when all low-level assembly instructions generated by the single source instruction have been executed.

This includes:

- All instructions for a source macro instruction.
- Any C instructions that generate several assembler instructions.


Subsequent source lines (called by the current source line) may be in a different function or file, as determined by the execution flow.

NOTE: *A trap instruction is treated as a subroutine (a BSR or JSR). Trap 32 is reserved by CodeScape and treated as a single instruction. Use Forced Step Into to step into the trap 32 routine. Stepping into trap 32 may cause the monitor to fail.*

Force step a line of source code at the disassembly level

- Click Debug, point to Execution then click Forced Step Into.

-OR-

- On the Debug toolbar click .


In a Disassembly region, Forced Step Into causes individual assembly instructions to be traced one at a time where this is normally not allowed, for example stepping into a trap 32.

In a Source region, the target executes the instruction at the PC with the current register values then stops at each individually generated assembler instruction. Each individual assembler instruction is traced using the disassembly-level Single Step instead of the source-level Single Step.

Stepping over code

Execution trace history is generated when stepping over. This is useful when you need to undo a step operation. Step Over performs a single step if Step Over is not relevant in the current context.

To step over a line of source code:

- Click Debug, point to Execution then click Step Over.
- OR-
- On the Debug toolbar click .


In disassembled code, the target executes the instruction at the PC then stops. A Trap, JSR or BSR is treated as a single instruction and program execution halted on the next instruction in memory when the routine is complete.

In source code, the target executes the instruction at the PC then stops when the source file reference has changed. When stepping over a function call, the entire function is executed. Execution is halted on the next source line.

NOTE: *You cannot step over conditional branches.*

Step out of a line of source or disassembled code


Use Step Out (return from function) to run to and stop at the end of the current function call.

- Click Debug, point to Execution then click Step Out.
- OR-
- Right-click in a region, point to Execution then click Step Out.
- OR-
- On the Debug toolbar, click .

Undo a step

- Click Debug, point to Execution then click Unstep.

-OR-

- On the Debug toolbar click, .

CodeScape keeps a history of trace actions. Trace history is built-up and discarded automatically. When you Unstep, only the current state of the processor and memory contents are untraced. You can Unstep:

- Instructions that are executed as a series of individual disassembly instructions.
- Traces in Source and Disassembly regions as long as there is a trace history left.

Notes

1. *You cannot Unstep blocks of instructions.*
2. *Where code is stepped over, all trace history to this point is lost.*

Enable Animated Step Run (animate trace)

Select Enable Animated Step Run (default) to update all regions as each instruction executes.

- Click Debug, then click Enable Animated Step Run.

CodeScape will trace instructions and display updated information in the active window until a breakpoint occurs, or until another command is issued, for example *Run* or *Stop*.

Step Run In


Use Step Run In to run to then stop at the start of each successively nested function call.

- Click Debug, point to Execution then click Step Run In.

-OR-

- Right-click in a region, point to Execution then click Step Run In.


-OR-

- On the Debug toolbar click .

CodeScape will run to the start of the next function and then stop.


Step Run Out

Use Step Run Out to run to and stop after each successively nested function call has completed.

- Click Debug, point to Execution then click Step Run Out.
-OR-
- Right-click in a region, point to Execution then click Step Run Out.
-OR-
- On the Debug toolbar click .

CodeScape will run to the end of the current function and then stop.

Step Run Until...


1. Do one of the following:
 - Click Debug, point to Execution then click Step Run Until...
-OR-
 - On the Debug toolbar click, .
2. Enter an expression to run to in the Expression Evaluator.

CodeScape will trace instructions and display updated information in the active window until a breakpoint occurs, or until another command is issued, for example start/stop.

NOTE: If the expression evaluates to a zero result, tracing continues.


Restart processor execution

Restart reloads the open program file, including the debug information, and resets the PC to the entry point (if known). If the program file's last modification time has changed, symbolic information is loaded.

- Right-click in the Target window, point to Execution then click Restart.
- -OR-
- On the Debug toolbar, click .
- -OR-
- Right-click in any region, point to Execution then click Restart.

Stop a program running

To stop a program running, in the Target region, select the processor on which your program is loaded, then do one of the following:

- Right-click in the Target window, point to Execution then click Stop.
- -OR-
- On the Debug toolbar, click .
- -OR-
- Click Debug, point to Execution, then click Stop.

Breakpoints

CodeScape has extensive software and hardware debugging features including breaking on data accesses within memory ranges and on external peripheral access. Breakpoint options include:

- Adding breakpoints, and adding a breakpoint at the current cursor position.
- Removing breakpoints, and removing all breakpoints.
- Enabling and disabling breakpoints, and resetting breakpoints.

Breakpoint options are on the Breakpoint toolbar, the Debug menu, the region specific shortcut menus, and the Configure breakpoint(s) dialog. In the Configure breakpoint(s) dialog you can also set any specific breakpoint features you require.

Important

Software breakpoints cause exceptions during program execution if encountered in a memory area other than the one they were defined in. For example, CodeScape reports an unknown exception if the range 0x0C000000, 0x0D000000 is an image of 0x8C000000, 0x8D000000 and a software breakpoint that is inserted at address 0x0C00B000 occurs at 0x8C00B000.



To avoid this, mark mirrored memory images as shared memory in the configuration file dali.cfg. The example below defines three areas of shared memory as the same (Tag:B). The mirrored ASE-breaks are hidden and a breakpoint symbol is shown at their addresses.

```
[TARGET DEVELOPMENT MACHINE MasterSH4EVA_SharedMemory]
SharedMemory = 0x0C000000, 0x0CFFFFFF, Tag:B
SharedMemory = 0x8C000000, 0x8CFFFFFF, Tag:B
SharedMemory = 0xAC000000, 0xACFFFFFF, Tag:B
```

NOTE: *If you add a breakpoint that uses a register or variable name as its address, the expression is only evaluated the first time it occurs during program execution.*

Adding breakpoints

You can add a breakpoint in Source, Disassembly, Memory, and Watch regions. You can add breakpoints at any time during program execution. Program execution stops when a breakpoint occurs.

When a breakpoint is set and enabled in a Source or Disassembly region, the breakpoint set icon, , appears in the first column. When a breakpoint is disabled, the breakpoint disabled icon, , appears in the first column. When a watched variable is visible in the Watch region, the watched variable icon appears. In a Memory region the background color of the specified address changes.

A breakpoint is set with the following default behavior:


- Code breakpoint execution is halted once it has been triggered and no other action, such as logging, is performed. Code breakpoints are implemented in hardware if a ROM address is encountered or software otherwise.
- Watch breakpoints can be triggered by any read or write data access to hardware. A message appears when the breakpoint has been triggered and all conditions have been met by default.

All breakpoint locations are tested to make sure that they are placed and configured correctly. If a problem is found a message appears prompting you to re-configure the breakpoint.

You can add a breakpoint only to a line that generates code. (Shown by a “.” in column one of a Source region or Watch region, or at any point in the Disassembly region.)


To change the default behavior of a breakpoint see *“Configuring breakpoints” on page 232*.

Add and run to a code breakpoint

1. Right-click, click Goto Address.
2. On the Breakpoint toolbar, click  to set a breakpoint.
3. Right-click, click Execution, click Run.
Your program will run until the breakpoint occurs.


Add a breakpoint at the current cursor position


In a Source or Disassembly region you can add code breakpoints. In a Watch or Memory region you can add data breakpoints. In any region, place the cursor in the required position then:

- Click Debug, point to Breakpoints then click Toggle Breakpoint.
-OR-
- Right-click, point to Breakpoints then click Toggle Breakpoint.
-OR-
- On the Breakpoint toolbar, click .

Removing breakpoints


In any region, place the cursor on the required breakpoint then:

- Click Debug, point to Breakpoints then click Toggle Breakpoint.
-OR-
- Right-click, point to Breakpoints then click Toggle Breakpoint.
-OR-
- On the Breakpoint toolbar, click .
- -OR-
- In the Configure breakpoint(s) dialog, select the breakpoint that you want to disable then click Remove.

The breakpoint set icon, , will disappear from the code window.


Remove all breakpoints



In any region, place the cursor on the required breakpoint then:

- Click Debug, point to Breakpoints then click Remove all Breakpoints.
-OR-
- On the Breakpoint toolbar, click .
- OR-
- Right-click, point to Breakpoints then click Remove all Breakpoints.
-OR-
- In the Configure breakpoint(s) dialog, click Remove All.

Enable a disabled breakpoint


In any region, place the cursor on the required breakpoint then:

- Click Debug, point to Breakpoints then click Enable Breakpoint.
-OR-
- Right-click, point to Breakpoints then click Enable Breakpoint.
-OR-
- On the Breakpoint toolbar, click .
- OR-
- In the Configure breakpoint(s) dialog, click Code Settings and select Breakpoint Enabled.

The breakpoint set icon changes from  to  to show that the breakpoint is enabled.


Disable an enabled breakpoint

In any region, place the cursor on the required breakpoint then:


- Click Debug, point to Breakpoints then click Disable Breakpoint.
-OR-
- Right-click, point to Breakpoints then click Disable Breakpoint.
-OR-
- On the Breakpoint toolbar, click .
- OR-
- In the Configure breakpoint(s) dialog, click Code Settings and clear Breakpoint is Enabled.

The breakpoint set icon changes from  to  to show that the breakpoint is disabled.


Enable all breakpoints

- Click Debug, point to Breakpoints then click Enable all Breakpoints.
-OR-
- Right-click, point to Breakpoints then click Enable all Breakpoints.
-OR-
- On the Breakpoint toolbar, click .

Disable all breakpoints

- Click Debug, point to Breakpoints then click Disable all Breakpoints.
-OR-
- Right-click, point to Breakpoints then click Disable all Breakpoints.
-OR-
- On the Breakpoint toolbar, click .

Reset all breakpoints

- Click Debug, point to Breakpoints then click Reset all Breakpoints.
-OR-
- Right-click, point to Breakpoints then click Reset all Breakpoints.
-OR-
- On the Breakpoint toolbar, click .

NOTE: *Resetting all breakpoints sets all conditional values, including the current count, to their starting conditions.*

Reset the trigger count for a breakpoint

- In the Configure breakpoint(s) dialog select the breakpoint, click Reset.


Reset only the current value of the count for a breakpoint

- In the Configure breakpoint(s) dialog select the breakpoint, click General Conditions and click Reset Current.

Configuring breakpoints

CodeScope allows breakpoint configuration including data accesses within memory ranges and breakpoints on external peripheral devices.

You can select and configure, and add and configure breakpoints manually in the Configure breakpoint(s) dialog. To configure a breakpoint:

- Click Debug, point to Breakpoints then click Configure Breakpoint(s)...
-OR-
- Right-click, point to Breakpoints then click Configure Breakpoint(s)...
-OR-
- On the Breakpoint toolbar, click .

Notes

1. *Software breakpoints cause exceptions during program execution if encountered in a memory area other than the one they were defined in.*
2. *Watch breakpoints trigger on data access, and code breakpoints trigger on the fetch-execute phase of the instruction cycle.*

The Configure breakpoint(s) dialog

In the Configure breakpoint(s) dialog you can:

- Add, remove, and configure code and watch breakpoints.
- Enable or disable a breakpoint, set its location, and the resources it will use.
- Specify when a breakpoint will occur.
- Configure a prompt for when a breakpoint occurs.

Using the Code Settings tab

Code breakpoints trigger on instruction execution. When a code breakpoint triggers the PC is at the same instruction in the pipeline. The Code Settings tab becomes available when you add or select a code breakpoint to configure.

1. Do one of the following:
 - Select a code breakpoint to configure from the list.
 - OR-
 - Click Add Code to add a code breakpoint to configure.
2. Select Breakpoint Enabled (default), to enable a breakpoint.
You may be prompted to re-configure a disabled breakpoint. This can occur during code execution, restoring sessions, or when attributes could not be validated when configuring commands within this dialog. A disabled breakpoint does not affect code execution or use any hardware resources.
3. Specify the position in memory where the code will stop on execution. Under *Location Expression*:
 - Enter the required expression.
 - OR-
 - Click Define. The Breakpoint Location Expression dialog appears. Evaluate the expression to set the location address.
4. Then do one of the following:
 - Select C/C++, to use C/C++ expression syntax.
 - OR-
 - Select Assembly, to use assembly language syntax.
5. In the Implementation Mechanism group box:
 - Select Automatic and CodeScape will manage breakpoint resources.
Breakpoints are implemented in software by default. If this is not possible then hardware resources are used.
 - OR-
 - Select Software to specify a software breakpoint.
 - OR-
 - Select Hardware to set a hardware breakpoint that is specific to your target processor.

Using the Watch Settings tab

Watch (data) breakpoints trigger on memory data access. When a Watch breakpoint triggers the PC is several instructions ahead of that breakpoint in the pipeline.

The Watch Settings tab is available when you select or add a watch breakpoint to configure.

1. Do one of the following:
 - Select a watch breakpoint to configure from the list.
 - OR-
 - Click Add Watch to add a watch breakpoint to configure.
2. Select *Breakpoint Enabled* (default), to enable a breakpoint.

You may be prompted to re-configure a disabled breakpoint. This can occur during code execution, restoring sessions, or when attributes are not validated during command configuration in this dialog. A disabled breakpoint does not affect code execution or use any hardware resources.
3. Specify the position in memory where the breakpoint is accessed. Under *Location Expression*:
 - Enter the required expression.
 - OR-
 - Click Define. The Breakpoint location expression dialog appears. Evaluate the expression to set the location address.
4. Select *Include Data Condition* to change the Watch Access breakpoint into a Watch Data breakpoint that uses the features of the UBC (User Break Controller). Enter the required Data Expression, then click Define. The Breakpoint watch data expression dialog appears. Evaluate the expression to set the location address.
5. Then do one of the following:
 - Select C/C++, to use C/C++ expression syntax.
 - OR-
 - Select Assembly, to use assembly language syntax.

6. Under *Implementation mechanism*:

- Select Automatic and CodeScape will manage breakpoint resources. Breakpoints are implemented in software by default. If this is not possible then hardware resources are used.

-OR-

- Select Software to specify a software breakpoint.

-OR-

- Select Hardware to set a hardware breakpoint that is specific to your target processor.

7. Under *Access Size* enter the Access Size required (the default is Any). When you use Toggle to add a watch breakpoint its size, if known, is used instead of Any.

8. Under *Access Type* select the Access Type required. The default is Both read and write access.

NOTE: *If you place a watch (data) breakpoint on a member of a union it will trigger for all members of that size, regardless of type. This also applies to anonymous unions, except that two members of the same size appear as two variables sharing the same address in memory.*

Using the General Conditions tab

The General Conditions tab is for defining conditions that must be valid before a breakpoint is triggered. You can condition a breakpoint by memory access type and data value, and confirm that it executed on the correct trigger count.

NOTE: *To use a conditional expression select Include Conditional Expression.*

1. Do one of the following:
 - Enter a valid expression in the Include Conditional Expression text box.
 - OR-
 - Click Define to open the Breakpoint condition expression dialog, then define the expression.
2. Do one of the following:
 - Select C/C++, to use C/C++ expression syntax.
 - OR-
 - Select Assembly, to use assembly language syntax.

The expression is evaluated for a logical result where a value of zero represents false and non-zero values represent true.
3. Select *Include Trigger Count Condition* to include the trigger condition.
4. Enter the value for the *Current Count*.

The *Trigger Count Condition* is true when the *Current Count* reaches the specified value.
5. Under Counters, check that the *Current* value matches the *Trigger* value you set. Click Reset Current to return the current count to zero.
6. Select when to increment the count.

The default is to increment the Current Count whenever the breakpoint occurs or is evaluated.
7. If both expression and count conditions are included, select when to break in the expression. The default is OR.

Using the Trigger Actions tab

Use the commands on the Trigger Actions tab to specify how CodeScape responds when a breakpoint has triggered.

Select any or all of the following radio buttons:

- *Halt execution when conditions match* stops the program executing when the breakpoint conditions have been met.
Clear this check box to continue execution after all other requested actions have been performed.
- *Single shot - breakpoint is discarded when conditions match* discards the breakpoint after it has been triggered and all conditions have been met.
- *Message box prompt when conditions match* displays a message when the breakpoint has been triggered and all conditions have been met.
- *Beep when conditions match* tells your computer to beep when the breakpoint has been triggered and all conditions have been met.
- *Log when* tells CodeScape to either to produce a log when the breakpoint has been triggered or every time. Make sure that you enter a valid Log expression.
If there is no Log region for the Target Processor, CodeScape creates one.

Using the Advanced tab

NOTE: *The Location Address text box is read-only. To set the location, click the Code Settings tab.*

NOTE: *The ASID Mask Selector field is set to its default state and cannot be configured. It will be enabled in future releases.*

On the Advanced tab are commands for using the Hardware Implementation Mechanism. These commands apply only to Watch breakpoints and Code breakpoints.

Using the Advanced tab to specify code breakpoint options

1. Select *Location Mask* to specify which bits of the Location Address to mask out. Set Location Mask bits to 1 to ignore the corresponding Location Address bit, 0 otherwise.
2. Under *Break Mode* select either:
 - Before Execution.
 - OR-
 - After Execution.

Using the Advanced tab to specify watch breakpoint options

1. Select *Location Mask* to specify which bits of the Location Address to mask out. Set Location Mask bits to 1 to ignore the corresponding Location Address bit, 0 otherwise.
2. Under *Data Mask* set Data Mask bits. Set the bits to 1 to ignore the corresponding Data Address bit, 0 otherwise.
3. Under *Bus cycle* select the bus cycles to include. Select either CPU, or Peripheral (DMA), or both.

Using the Global tab to specify the debug environment for Hitachi SH4-EVA processors

On the Global tab are commands for setting the target processor's debug environment. The Global tab appears when you connect to an SH4-EVA target processor and you can specify any of the available options.

1. In the Global ASE Break Conditions for SH4-EVA CPU field:
 - *Select Enable on-chip access detection* and CodeScape will generate an on-chip I/O exception.
The values displayed are the last on-chip address accessed, and the last on-chip data access when the exception occurred.
 - *Select Enable break after LDTLB instruction execution* and CodeScape will generate an LDTLB instruction break.
The values displayed are the last PTEH loaded, and the last PTEL loaded into the MMU.
2. Under *Global UBC Exception Handler Option* select Use DBR vector (default).
CodeScape will use the debug stub default exception handler for UBCs. This lets you define exception handling routines in your program, and to modify the VBR without affecting the behavior of UBC breakpoints.

Breakpoint expression format

CodeScape has a powerful expression formatting facility for controlling the display of expressions in the Log tab on the Input / Output window.

Control formatting with expressions that work in a similar way to the C 'printf' function. The expressions are numbered from 0 and can be any valid debugger expression referencing register names or memory locations. The syntax for a formatting expression is:

```
{"FormattingString" | FormattingString}[ ,C/C++Expression]
```

Formatting string

A formatting string is a series of alphanumeric characters and three special format specifiers.

Formatting string

Use the format:	To:
\character	Explicitly define a character. For example, \\$ displays a \$ character.
\$param_num	Change the next argument index. For example, \$0 sets the argument index to 0.
%[flags] [width] [.precision] type	Print a series of formatted characters and values to the Log tab on the Input / Output window. Type %% to print a single percent character.

In the format %[flags] [width] [.precision] type, use the fields in the following ways:

- **[flags]** is an optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.
- **[width]** is an optional number that specifies the minimum number of characters output.
- **[.precision]** is an optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values.
- **type** is a required character that determines whether the associated argument is interpreted as a character, a string, or a number.

Flags specification

A flag directive is a character that justifies output and prints signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.

Flags

Flag	Meaning
-	Left align the result within the given field width. The default is right align.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type. The sign appears only for negative signed values by default.
0	If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and - appear, the 0 is ignored. If 0 is specified with a non-integer format (e.g. f, g, e) the 0 is ignored. The default is no padding.
blank (' ')	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear. Default: No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. Default: No blank appears. When used with the e, or f format, the # flag forces the output value to contain a decimal point in all cases. Default: Decimal point appears only if digits follow it. When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Default: Decimal point appears only if digits follow it. Trailing zeros are truncated. Ignored when used with c, d, i, u, or s.

Width specification

The second optional field of the format specification is the width specification. The width argument is a non-negative decimal integer controlling the minimum number of characters printed. If the output value has fewer characters than the specified width, blanks are added to the right of the value unless the left align flag (-) is set. If width is prefixed with 0, zeros are added instead of blanks (not useful for left aligned numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if width is not given, all characters of the value are printed to the Log tab (subject to the precision specification).

If the width specification is an asterisk (*), an int argument from the argument list supplies the value. The width argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause the truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

Precision specification

The third optional field of the format specification is the precision specification. It specifies a non-negative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value. If precision is specified as 0 and the value to be converted is 0, the result is no characters output, as shown below:

```
"%.0d", 0    /* No characters output */
```

If the precision specification is an asterisk (*), an int argument from the argument list supplies the value. The precision argument must precede the value being formatted in the argument list.

Type specification

Character, type, and output format

Character	Type	Output Format
c	int	Single-byte character.
C	int	Single-byte character.
d	int	Signed decimal integer.
i	int	Signed decimal integer.
o	int	Unsigned octal integer.
u	int	Unsigned decimal integer.
x	int	Unsigned hexadecimal integer, using "abcdef."
X	int	Unsigned hexadecimal integer, using "ABCDEF."
e	double	Signed value with the form [-]d.ddd e [sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is three decimal digits, and sign is + or -.
f	double	Signed value with the form [-]dddd.ddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	double	Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is only used when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.

Character	Type	Output Format
G	double	Identical to the g format.
p	Pointer to	Prints the address pointed to by the argument in the form similar to %X (i.e. uppercase hexadecimal digits).
s	String	Specifies a single-byte-character string. Characters are printed up to the first null character or until the precision value is reached.
S	String	Specifies a single-byte-character string. Characters are printed up to the first null character or until the precision value is reached.
i	Address	Displays (by disassembling) the op-code at the specified address.
l	Address	Displays (by disassembling) the op-code at the specified address with qualified symbol names if available.
t		Insert timestamp.
T		Insert timestamp.

Examples using the types specification

Expression	Description
"%X", pc	Evaluate and display the expression "pc" (this could be a variable or register) as an uppercase hexadecimal number. Output: 3b0
"0x%08x", \$pc	Evaluate and display the expression "\$pc" (this must be a register) as a lowercase hexadecimal number prepended by "0x" and padded with zeroes to 8 characters. Output: 0x000003b0
"0x%08x -> \$0 %I ", \$pc	Evaluate and display the expression "\$pc" (this must be a register) as a lowercase hexadecimal number prepended by "0x" and padded with zeroes to 8 characters followed by the disassembly of the op-code at that address with qualified symbol names. The \$0 resets the parameter index back to zero so the expression "\$pc" is used for both formatted options. Output: 0x000003b0 -> mov.l #BaseClass::i, r3

Evaluating expressions

The expression evaluator dialog is used for several operations, including: Edit Register, and Goto Address. In the dialog you can use the C/C++ expression evaluator or Assembler format.

Expression evaluator dialog

The expression evaluator is a general purpose dialog used for several operations, including: edit register, edit memory value, edit local value, edit watch value, and goto address.

The options on the Expression Evaluator

Use the:	To:
Expression Combo box	Edit an existing expression, or select one from the history list.
Result field	View the results of an expression evaluation including any error messages.
Expression Format radio buttons	Select C/C++, or Assembly as the expression format.
Default radix radio buttons	Select binary, octal, decimal, or hex, as the radix to use for the expression, or specify another radix in the Other text box. For C expressions this permits only control of the output radix.
Evaluate button	Evaluate an expression in the Expression Combo text box.
Symbol button	Use the Symbol Completion dialog to search for a symbol from those available in the program file.
File button	View a list of all the files used to build the program file in the List Files in Program File dialog. The dialog also provides access to the address for "file:line number" information.
Lock check box	Lock the current expression to a file or symbol.

NOTE: *When you evaluate assembler expressions in a Watch region the maximum number of characters you can enter is 127.*

Symbol Completion dialog

Use the Symbol Completion dialog to search for a symbol in the program file.

Options on the Symbol Completion dialog

Use the:	To:
Find String text box	Enter the first few characters of the symbol to search for.
Only Search For Symbols Within Scope check box	Search for symbols in scope (select the check box), or to search for symbols in the whole program (clear the check box).
Include Linkage Level Symbols check box	Include low level symbols in the search (select the check box).
Possible Completions text box	View a list of all symbols that match the current Search String.
Lookup button	Click Lookup to start another search.
OK button	Accept the current search string.
Cancel button	Ignore current search string.

C/C++ expressions

The C/C++ expression evaluator accepts expressions in a C-like format.

Operator precedence

Operator	Type	Usage	Description
()	Primary	[expr]	Parenthesis Brackets
[]	Primary	pointer[expr]	Subscripting
.	Binary	object.member	Member selection
->	Binary	pointer->member	Member selection
sizeof()	Unary	sizeof(expr)	Size of object
sizeof()	Unary	sizeof(type)	Size of type
-	Unary	- expr	Unary Minus
+	Unary	+ expr	Unary Plus
~	Unary	~ expr	Bitwise NOT
!	Unary	! expr	Logical NOT
*	Unary	* expr	De-reference
&	Unary	& lvalue	Address of
*	Binary	expr *expr	Multiply
/	Binary	expr/expr	Divide
%	Binary	expr % expr	Modulo (remainder)
+	Binary	expr + expr	Add (plus)
-	Binary	expr - expr	Subtract (minus)
<<	Binary	expr << expr	Shift Left
>>	Binary	expr >> expr	Shift Right
<	Binary	expr < expr	Less than

Operator	Type	Usage	Description
<=	Binary	expr <= expr	Less than or equal
>	Binary	expr > expr	Greater than
>=	Binary	expr >= expr	Greater than or equal
==	Binary	expr == expr	Equal
!=	Binary	expr != expr	Not Equal
&	Binary	expr & expr	Bitwise AND
^	Binary	expr ^ expr	Bitwise Exclusive OR
	Binary	expr expr	Bitwise Inclusive OR
&&	Binary	expr && expr	Logical AND
	Binary	expr expr	Logical Inclusive OR

Operands that the C/C++ operators act on

Operand	Definition
Constants (Floating or Integer)	Constants can be: hexadecimal numbers prefixed with '0x', octal numbers prefixed with '0', or unsigned numbers postfixed with a 'U'. Characters, for example 'A', are not accepted.
Registers	The name of a valid register.
Symbols	Symbol names take into account their type. For example a variable defined as (char chr = 'A') would return 'A' when evaluated. To get the address of the object '&chr' is required.

Notes

1. *Scope operator, '::' The scope operator is implemented as the standard for class name operations, for example `c_basic::print` and for global operations, for example `::print`. The scope operator also has a custom implementation for file operations, for example `"bob.c"::print`.*
2. *Assignment operators, like `=`, `+=`, `*=`, `++`, `--`, are not implemented in this release.*

Assembler expressions

Operator precedence:

Operator	Type	Usage	Description
()	Primary	(expr)	Parenthesis Brackets
[]	Primary	[expr]	Address of
-	Unary	- expr	Negative expr
+	Unary	+ expr	Positive expr
~	Unary	~ expr	Bitwise NOT
<<	Binary	expr << expr	Shift left
>>	Binary	expr >> expr	Shift right
&	Binary	expr & expr	Logical AND
!	Unary	! expr	Logical NOT
	Binary	expr	Logical Inclusive OR
^	Binary	^ expr	Logical Exclusive OR
*	Binary	expr * expr	Multiply
/	Binary	expr / expr	Divide
%	Binary	expr % expr	Modulo (remainder)
+	Binary	expr + expr	Add (plus)
-	Binary	expr - expr	Subtract (minus)
=	Binary	expr = expr	Equals
<>	Binary	expr <> expr	Not Equals
<	Binary	expr < expr	Less Than
<=	Binary	expr <= expr	Less Than or Equals
>	Binary	expr > expr	Greater Than

Operator	Type	Usage	Description
>=	Binary	expr >= expr	Greater Than or Equals

Operands that the assembly operators act on

Operand	Definition		
Constants (Integer)	Constants can be defined in several operators to denote different radix:		
	Variable Hex Decimal Binary	X_<number>	where X is a single digit base prefix '\$' or '0x' postfix 'h' prefix '#' postfix 'd' prefix '%' postfix 'b'
Registers	The name of a valid register.		
Symbols	Symbols are evaluated to labels, so a variable of type (char chr = 'A'), would return the address of (label to) the variable A when evaluated. Labels can be qualified by: 'b', 'w', 'l' for byte, word or long respectively [symbol]@b, [symbol]@w, [symbol]@l :<number> for the filename line number		

Writing Scripts

Writing scripts

With CodeScape's script commands you can write:

- Boot scripts that initialize the DASH.
See "Preparing boot scripts and boot ROMs" on page 68.
- Scripts that automate routine tasks when run.
For example, write a script that will load three different binary files from specified locations into target memory, then add it to CodeScape's menu and assign it a shortcut key.
- Scripts that setup the Script region to complete an operation of your choice.
For example, write a script that sets the Script region display to permanently examine a specific area of memory.

CodeScape's script commands are demonstrated in example Microsoft® JScript® and VBScript files provided on the release CD. You can use the functions available in either script language to add commands of your own.

For details about using JScript and VBScript connect to the scripting area on the Microsoft Developer Network at: <http://msdn.microsoft.com/scripting>

Using scripts

When you run a script the Input/Output window appears automatically and displays the Script tab with all messages generated by the current script.

To open the Input/Output window without running a script:

- Click View, Toolbar then select the Input/Output check-box and click OK.

The shortcut menu on the Scripts tab

Use:	To:
Run Script	Select and run a script.
Clear	Clear the contents of the Script tab.
Print	Print the contents of the script tab.
Save To File...	Save the contents of the script tab to a file.
Allow Docking	Toggle docking for the window on or off.
Hide	Hide the window.

Assign a shortcut key to the script

After adding a script to the menu you can assign it a shortcut key. To do this:

1. Click Tools, select Customize then click Keyboard...
The Shortcut Keys dialog box appears.
2. In the Select a command tree highlight the script's menu command.
3. Click Create Shortcut...
4. The Select shortcut dialog box appears.
5. Press the key combination you require.
If you press a key or key combination that is currently assigned to another command, that command appears under Replaces.
6. Click OK.
The new shortcut appears in the Assigned shortcuts text box.
7. Click OK.

Scripting commands

CodeScape scripting commands are described in the following format:

Syntax

`Command(parameters)`

Supported by

Indicates which of the following support the command: DASH boot scripts, Running scripts, Script region.

Description

Describes what the command does.

Parameters and Remarks

parameters are the parameters the command takes.

Examples

VBScript

`Command(parameters)`

JScript

`Command(parameters);`

GetTargetInfo

Syntax

```
GetTargetInfo(target)
```

Supported by

Running scripts, Script region.

Description

Returns an identifier with information about the specified target.

Parameters and Remarks

target identifies the target to find out about.

Examples

VBScript

```
GetTargetInfo(target)
```

JScript

```
GetTargetInfo(target);
```

General and control

LoadProgramFile

Syntax

```
LoadProgramFile(pathname)
```

Supported by

Running scripts, Script region.

Description

Loads the specified program file and debug information. Returns 1 if a file is loaded, otherwise it returns 0.

Parameters and Remarks

pathname specifies the directory path and file to load. For example, C:\dashscript\load.j.

Examples

VBScript

```
LoadProgramFile("e:\projects\maketest\hello.elf")
```

JScript

```
LoadProgramFile("e:\\projects\\maketest\\hello.elf")
```

LoadProgramFileEx

Syntax

```
LoadProgramFileEx(filename, hardReset, binaryOnly)
```

Supported by

Running scripts, Script region.

Description

Loads the specified program file and debug information. Returns 1 if a file is loaded, otherwise it returns 0.

Parameters and Remarks

filename is the name of the file to load.

hardReset is a boolean which will perform a hard reset when set else none.

binaryOnly. Set to: 0 to load binary and symbols, 1 to load only binary, and 2 to load only symbols.

Examples

VBScript

```
LoadProgramFileEx("e:\projects\maketest\hello.elf", false, 1)
```

JScript

```
LoadProgramFileEx("e:\\projects\\maketest\\hello.elf", false, 1)
```

LoadBinaryFile

Syntax

```
LoadBinaryFile(pathname,  
               Numeric binary location)
```

Supported by

Running scripts, Script region.

Description

Loads a binary file from the specified location into target memory. Returns 1 on success, otherwise it returns 0.

Parameters and Remarks

pathname identifies the file.

Numeric binary location specifies where to load the file from.

Examples

VBScript

```
Sub LoadSomeBinary()  
    call LoadBinaryFile "d:\projects\codescape\example.bin", "201392128"  
    call LoadBinaryFile "d:\projects\codescape\example.bin", 201392128  
    call LoadBinaryFile "d:\projects\codescape\example.bin", "0xc010000"  
    call LoadBinaryFile "d:\projects\codescape\example.bin", "main"  
End Sub
```

JScript

```
function LoadSomeBinary()  
{  
    LoadBinaryFile("d:\\projects\\codescape\\example.bin", "201392128");  
    LoadBinaryFile("d:\\projects\\codescape\\example.bin", 201392128);  
    LoadBinaryFile("d:\\projects\\codescape\\example.bin", "0xc010000");  
    LoadBinaryFile("d:\\projects\\codescape\\example.bin", "main");  
}
```

HardReset

Syntax

```
HardReset ( )
```

Supported by

Running scripts, Script region.

Description

Resets the target board.

Parameters and Remarks

None.

Examples

VBScript

```
HardReset ( )
```

JScript

```
HardReset ( ) ;
```

SoftReset

Syntax

```
SoftReset ( )
```

Supported by

Running scripts, Script region.

Description

Resets the debug system.

Parameters and Remarks

None.

Examples

VBScript

```
SoftReset ( )
```

JScript

```
SoftReset ( ) ;
```

Run

Syntax

Run ()

Supported by

Running scripts, Script region.

Description

Runs the target processor.

Parameters and Remarks

None.

Examples

VBScript

Run ()

JScript

Run () ;

IsRunning

Syntax

```
IsRunning()
```

Supported by

Running scripts, Script region.

Description

Queries the run state of the target processor. Returns 1 if running, 0 if not running.

Parameters and Remarks

None.

Examples

VBScript

```
Dim Running
Running = 1
Do
    Running = IsRunning
Loop Until Running = 0
```

JScript

```
Run();
while(IsRunning() != 0);
```

Stop

Syntax

```
StopTarget( )
```

Supported by

Running scripts, Script region.

Description

Stops the current target. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

None

Examples

VBScript

```
if Stop() <> 0 then
    Write("Target has stopped")
else
    Write("Target has not stopped")
end if
```

JScript

```
if(Stop())
{
    Write("Target has stopped");
}
else
{
    Write("Target has not stopped");
}
```

SymbolExists

Syntax

```
SymbolExists(string Symbol)
```

Supported by

Running scripts, Script region.

Description

Looks up the specified symbol in the symbol table. Returns 0 if the parameter specified is not a string with an alphabetic first character, or if the symbol could not be found.

Parameters and Remarks

string Symbol is the symbol to search for in the specified string.

Examples

VBScript

```
if SymbolExists("main") <> 0 then
    Write("Symbol exists")
else
    Write("Symbol does not exist")
end if
```

JScript

```
if(SymbolExists("main"))
{
    Write("Symbol exists");
}
else
{
    Write("Symbol does not exist");
}
```

EvaluateSymbol

Syntax

```
EvaluateSymbol(string Symbol)
```

Supported by

Running scripts, Script region.

Description

Searches for the symbol specified in the string. Returns the value of the symbol from the symbol table on success.

Parameters and Remarks

string symbol is a string that includes a symbol to search for.

The string must have an alphabetic first character. The string can be written as a whole expression.

Examples

VBScript

```
var main = EvaluateSymbol("main")
```

JScript

```
var main = EvaluateSymbol("main");
```

Basic read/write

ReadByte

Syntax

```
ReadByte(Numeric address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a byte (8 bits) from a specified target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

ReadWord

Syntax

```
ReadWord(address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a word (16 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

ReadLong

Syntax

```
ReadLong (address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a long (32 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    Write("Byte at main = " & ReadByte("main"))  
    Write("Word at main + 4 = " & ReadWord("main + 4"))  
    Write("Long at main + 8 = " & ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    Write("Byte at main = " + ReadByte("main"));  
    Write("Word at main + 4 = " + ReadWord("main + 4"));  
    Write("Long at main + 8 = " + ReadLong("main + 8"));  
}
```

WriteByte

Syntax

```
WriteByte(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a byte (8 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

WriteWord

Syntax

```
WriteWord(address,  
           value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a word (16 bits) to a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

value is a number or string expression that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

WriteLong

Syntax

```
WriteLong(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a long (32 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call WriteByte ("main", 255)  
    call WriteWord ("main + 4", "0xabcd")  
    call WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    WriteByte("main", 255);  
    WriteWord("main + 4", "0xabcd");  
    WriteLong("main + 8", "0xfedcba");  
}
```

WriteRegister

Syntax

```
WriteRegister(Register name,  
              Numeric value)
```

Supported by

Running scripts, Script region.

Description

Writes to a target processor register.

Parameters and Remarks

Register name identifies the register.

Numeric value is a number to assign to that register.

Examples

VBScript

```
WriteRegister ("fr0", 3.14159)
```

JScript

```
WriteRegister("fr0", 3.14159);
```

ReadRegister

Syntax

```
ReadRegister(RegisterName)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads from a target processor register.

Parameters and Remarks

Register Name specifies the register.

Examples

VBScript

```
Sub ReadSomeRegisters()  
    Write("Value of pc = " & ReadRegister("pc"))  
    Write("Value of r0 = " & ReadRegister("r0"))  
End Sub
```

JScript

```
function ReadSomeRegisters()  
{  
    Write("Value of pc = " + ReadRegister("pc"))  
    Write("Value of r0 = " + ReadRegister("r0"))  
}
```

ReadString

Syntax

```
ReadString(address,  
            length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated ASCII string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
call ReadString(address, 100)
```

JScript

```
if(ReadString(address, 100) != "Hello John")  
{  
    throw "Read/Write String failed"  
}
```

ReadWString

Syntax

```
ReadWString(address,  
            length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated UNICODE string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
ReadWString(address, 100)
```

JScript

```
if(ReadWString(address, 100) != "GoodBye")  
{  
    throw "Read/Write String failed"  
}
```

WriteString

Syntax

```
WriteString(address,  
            string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call WriteString(address, "Hello John")
```

JScript

```
WriteString(address, "Hello John");
```

WriteWString

Syntax

```
WriteWString(address,  
             string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) as unicode to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call WriteWString(address, "GoodBye")
```

JScript

```
WriteWString(address, "GoodBye");
```

Breakpoints

CreateBreakpoint

Syntax

```
CreateBreakpoint (type,  
                  location)
```

Supported by

Running scripts, Script region.

Description

Creates a breakpoint of the given type at the given address. Returns a breakpoint identifier on success, otherwise 0.

Parameters and Remarks

To create a Code Breakpoint set *Type* to 0.

To create a Watch Breakpoint set *Type* to 1.

location specifies where to set the breakpoint.

Examples

In the examples, the breakpoint types are represented by the following:

```
BPTYPE_CODE=0;  
BPTYPE_WATCH=1;
```

VBScript

```
Sub CreateWatchBP()  
    breakID= CreateBreakpoint(BPTYPE_WATCH, "main")  
    SetWatchBreakpointParameters(breakID,  
                                true,  
                                "14",  
                                BPEXPR_C,  
                                BPACCESSSIZE_BYTE,  
                                BPACCESSTYPE_WRITE)  
End Sub
```

JScript

```
function CreateWatchBP()  
{  
    breakID= CreateBreakpoint(BPTYPE_WATCH, "main");  
    SetWatchBreakpointParameters(breakID,  
                                true,  
                                "14",  
                                BPEXPR_C,  
                                BPACCESSSIZE_BYTE,  
                                BPACCESSTYPE_WRITE);  
}
```

ClearBreakpoint

Syntax

```
ClearBreakpoint ( ID )
```

Supported by

Running scripts, Script region.

Description

Clears the specified breakpoint created by CreateBreakpoint on success.

Parameters and Remarks

ID is the identifier returned by CreateBreakpoint.

Examples

VBScript

```
ClearBreakpoint ( "ID" )
```

JScript

```
ClearBreakpoint ( "ID" ) ;
```

ClearAllBreakpoints

Syntax

```
ClearAllBreakpoints()
```

Supported by

Running scripts, Script region.

Description

Removes all breakpoints.

Parameters and Remarks

None.

Examples

VBScript

```
ClearAllBreakpoints()
```

JScript

```
ClearAllBreakpoints();
```

EnableBreakpoint

Syntax

```
EnableBreakpoint (identifier,  
                 boolean enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables a breakpoint. Returns 1 on successfully enabling or disabling the breakpoint, otherwise it returns 0.

Parameters and Remarks

identifier specifies the breakpoint.

Set the *boolean enable* value to:

- 1 to enable the specified breakpoint.
- 0 to disable the specified breakpoint.

Examples

VBScript

```
call EnableBreakpoint (breakID, true)
```

JScript

```
EnableBreakpoint(breakID, true);
```

EnableAllBreakpoints

Syntax

```
EnableAllBreakpoints(enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables all breakpoints. Returns 1 on successfully enabling or disabling the breakpoint, otherwise it returns 0.

Parameters and Remarks

Set *enable* to:

- 1 to enable all breakpoints.
- 0 to disable all breakpoints.

Examples

VBScript

```
call EnableBreakpoints (1)
```

JScript

```
EnableBreakpoints(1);
```

SetBreakpointAction

Syntax

```
SetBreakpointAction(identifier,  
                    numeric action,  
                    boolean enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables specific actions when a breakpoint has been hit. The allowed actions are:

- Stop the target.
- Remove the breakpoint.
- Display a message box.
- Beep.

Returns 1 on success, otherwise it returns 0.

NOTE: *You must call `SetBreakpointAction` separately for each action you require. The examples demonstrate how to do this.*

Parameters and Remarks

identifier specifies the breakpoint.

Set the *boolean enable* value to:

- 1 to enable the specified breakpoint action.
- 0 to disable the specified breakpoint action.

Set the *numeric action* value to:

- 0 to stop the target when the breakpoint when it is hit.
- 1 to remove the breakpoint after it has been hit.
- 2 to display a message box prompt when the breakpoint has been hit.
- 3 to beep when the breakpoint has been hit.

Examples

VBScript

```
call dash.SetBreakpointAction (breakID, BPACTION_HALT, true)
call SetBreakpointAction (breakID, BPACTION_ONESHOT, false)
call SetBreakpointAction (breakID, BPACTION_PROMPT, false)
call SetBreakpointAction (breakID, BPACTION_BEEP, true)
```

JScript

```
dash.SetBreakpointAction(breakID, BPACTION_HALT, true);
SetBreakpointAction(breakID, BPACTION_ONESHOT, false);
SetBreakpointAction(breakID, BPACTION_PROMPT, false);
SetBreakpointAction(breakID, BPACTION_BEEP, true);
```

SetBreakpointCondition

Syntax

```
SetBreakpointCondition(identifier,  
                        string expression,  
                        numeric expression type,  
                        numeric trigger count,  
                        boolean incOn,  
                        boolean breakWhen)
```

Supported by

Running scripts, Script region.

Description

Sets a conditional expression for a breakpoint. Returns 1 on success, otherwise it returns 0.

Parameters and Remarks

identifier specifies the breakpoint.

string expression is a string representing the condition.

Set *numeric expression type* to 0 for C/C++, or to non-zero for assembly.

numeric trigger count is the number of hits before breakpoint actions are performed.

incOnTrue derements the trigger count only when conditions are true.

incOnFalse always decrements the trigger count.

Set *boolean breakWhen* as false to break either when the trigger ccount reaches 0 or when condition is true. Set *breakWhen* as true to break when trigger count reaches zero and the condition is true.

Examples

VBScript

```
call setBreakpointCondition (breakID,  
                             "index == 375",  
                             BPEXPR_C,  
                             37,  
                             true,  
                             true)
```

JScript

```
setBreakpointCondition(breakID,  
                       "index == 375",  
                       BPEXPR_C,  
                       37,  
                       true,  
                       true);
```

SetWatchBreakpointParameters

Syntax

```
BOOL SetWatchBreakpointParameters(Identifier,  
                                   boolean incDataCondition,  
                                   string dataCondition,  
                                   numeric expressionType,  
                                   numeric accessSize,  
                                   numeric accessType)
```

Supported by

Running scripts, Script region.

Description

Sets the parameters for a watch breakpoint.

Parameters and Remarks

identifier specifies the breakpoint.

boolean incDataCondition is set to true to include a data condition otherwise false.

string dataCondition is a numeric expression that sets the data condition.

numeric expressionType is a numeric expression that sets the expression type. To set the expression syntax to:

- C/C++, enter the value 0.
- Assembly, enter the value 1.

numeric accessSize is a numeric expression that sets the size. To set the size to:

- Any, enter the value 0.
- Byte, enter the value 1.
- Word, enter the value 2.
- Long, enter the value 4.
- Quad, enter the value 8.

numeric accessType is a numeric expression that sets the type. To set the type to:

- Read, enter the value 1.
- Write, enter the value 2.

-
- Read, or Write enter the value 3.

Examples

VBScript

```
call SetWatchBreakpointParameters (breakID,  
                                     true,  
                                     "14",  
                                     BPEXPR_C,  
                                     BPACCESSSIZE_BYTE,  
                                     BPACCESSTYPE_WRITE)
```

JScript

```
SetWatchBreakpointParameters(breakID,  
                              true,  
                              "14",  
                              BPEXPR_C,  
                              BPACCESSSIZE_BYTE,  
                              BPACCESSTYPE_WRITE);
```

SetBreakpointDataMask

Syntax

```
SetBreakpointDataMask(identifier,  
                        data mask)
```

Supported by

Running scripts, Script region.

Description

Sets the data mask for a watch breakpoint.

Parameters and Remarks

identifier specifies the breakpoint.

data mask must be a 32-bit number.

Examples

Each example is to mask 10 bits.

VBScript

```
call SetBreakpointDataMask (breakID,  
                             0x000003FF)
```

JScript

```
SetBreakpointDataMask(breakID,  
                       0x000003FF);
```

SetBreakpointLocationMask

Syntax

```
SetBreakpointLocationMask(breakID,  
                           maskSelect)
```

Supported by

Running scripts, Script region.

Description

Masks a location for the breakpoint. The identifier specifies the breakpoint, and mask. Select is a value that specifies the bits to mask.

To mask:

- No bits, enter the value 1.
- The lower 10 bits, enter the value 2.
- The lower 12 bits, enter the value 3.
- The lower 16 bits, enter the value 4.
- The lower 20 bits, enter the value 5.
- All bits, enter the value 6.

Returns true 1 on success, otherwise it returns 0.

NOTE: *This command is for SH4 targets.*

Parameters and Remarks

breakID is the identifier that specifies the breakpoint.

maskSelect specifies the bits to mask.

Examples

VBScript

```
call SetBreakpointLocationMask (breakID,  
                                BPLOCMASK_LOW10)
```

JScript

```
SetBreakpointLocationMask(breakID,  
                           BPLOCMASK_LOW10);
```

SetBreakpointLocationMaskEx

Syntax

```
SetExtendedBreakpointLocationMask(breakID,  
                                   mask)
```

Supported by

Running scripts, Script region.

Description

Masks a location for the breakpoint.

Returns true 1 on success, otherwise it returns 0.

NOTE: *This command is for SH2 targets allowing full 32-bit masking.*

Parameters and Remarks

breakID is the identifier that specifies the breakpoint.

maskSelect specifies the bits to mask.

Examples

VBScript

```
call SetExtendedBreakpointLocationMask(breakID,  
                                         BPLOCMASK_LOW10)
```

JScript

```
SetExtendedBreakpointLocationMask(breakID,  
                                   BPLOCMASK_LOW10);
```

Multiple Target Support

GetCurrentTarget

Syntax

```
GetCurrentTarget()
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the current target.

Parameters and Remarks

The strings returned from `GetCurrentTarget` should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = GetCurrentTarget()
```

JScript

```
target = GetCurrentTarget();
```

GetFirstTarget

Syntax

```
GetFirstTarget()
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the first target.

Parameters and Remarks

The strings returned from `GetFirstTarget` should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = GetFirstTarget()
```

JScript

```
target = GetFirstTarget();
```

GetNextTarget

Syntax

```
GetNextTarget ( current )
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the next target. The return string is empty if all targets have been identified.

Parameters and Remarks

current identifies the current target.

The strings returned from GetNextTarget should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = GetNextTarget ( current )
```

JScript

```
target = GetNextTarget ( current ) ;
```

SelectTarget

Syntax

```
SelectTarget(target)
```

Supported by

Running scripts, Script region.

Description

Selects the chosen target as the current target. Returns true if successfully selected, otherwise false.

Parameters and Remarks

target identifies the target to use.

Examples

VBScript

```
SelectTarget(target)
```

JScript

```
SelectTarget(target);
```

CopyData

Syntax

```
CopyData(target,  
         address,  
         size,  
         numElems,  
         destTarget,  
         destAddress)
```

Supported by

Running scripts, Script region.

Description

Copies data from one target to another.

Parameters and Remarks

target specifies the target to copy from.

address specifies the address to copy from.

size specifies the size of data (1, 2, 4, and 8 are valid sizes).

numElems specifies the number of elements of the given size to copy.

destTarget specifies the target to copy to.

destAddress specifies the address to copy to.

The maximum size allowed in one transfer is 65536, so *size* * *numElems* must be less than 65536.

The command lets you transfer data from a big endian target to a little endian target (and vice versa) correctly. This command should not be used to transfer large amounts of data.

Examples

VBScript

```
call CopyData(target, address, size, numElems, destTarget, destAddress)
```

JScript

```
CopyData(target, address, size, numElems, destTarget, destAddress);
```

RunTarget

Syntax

```
RunTarget(target)
```

Supported by

Running scripts, Script region.

Description

Runs the target specified. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

target identifies the target to run.

Examples

VBScript

```
if RunTarget(target) <> 0 then
    Write("Target is running")
else
    WScript.echo("Target is not running")
end if
```

JScript

```
if(RunTarget(target))
{
    Write("Target is running");
}
else
{
    Write("Target is not running");
}
```

StopTarget

Syntax

```
StopTarget(target)
```

Supported by

Running scripts, Script region.

Description

Stops the target specified. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

target identifies the target to stop.

Examples

VBScript

```
if StopTarget(target) <> 0 then
    Write("Target has stopped")
else
    Write("Target has not stopped")
end if
```

JScript

```
if(StopTarget(target))
{
    Write("Target has stopped");
}
else
{
    Write("Target has not stopped");
}
```

RunAllTargets

Syntax

```
RunAllTargets( )
```

Supported by

Running scripts, Script region.

Description

Runs all targets that are connected. Returns the number of targets successfully started.

Parameters and Remarks

None.

Examples

VBScript

```
RunAllTargets( )
```

JScript

```
RunAllTargets( ) ;
```

StopAllTargets

Syntax

```
StopAllTargets()
```

Supported by

Running scripts, Script region.

Description

Stops all the connected targets. Returns the number of targets successfully stopped.

Parameters and Remarks

None.

Examples

VBScript

```
StopAllTargets()
```

JScript

```
StopAllTargets();
```

IsTargetRunning

Syntax

```
IsTargetRunning(target)
```

Supported by

Running scripts, Script region.

Description

Queries the run status of a specific target. Returns true if the target is running, otherwise false.

Parameters and Remarks

target identifies the target to test.

Examples

VBScript

```
IsTargetRunning(target)
```

JScript

```
Write("Target is " + (IsTargetRunning(target) ? " " : "not ") + "running");
```

For more information refer to “Communications Channels” on page 516.

LibCross Fileserver

LibCross Fileserver Library

The LibCross fileserver provides low level routines that interface CodeScape with the standard C run-time library (libc.a). The fileserver supports these functions:

```
int debug_open (const char *filename, int flags, ...);

int debug_close (int file);
int debug_read(int file, char *ptr, int len);
int debug_write (int file, char *ptr, int len);
int debug_lseek(int file, int offset, int origin);

char * debug_getcwd(char *buffer, int maxlen);
int debug_chdir(const char *dirname);
int debug_mkdir(const char *dirname);
int debug_rmdir(const char *dirname);

int debug_findfirst(const char *filespec, struct SNASM_finddata_t
*fileinfo);
int debug_findnext(int handle, struct SNASM_finddata_t *fileinfo);
int debug_findclose(int handle);
int _ASSERT(int nFlag);
int debug_printf(char *format, ...);
int debug_runscript(const char *filename, SCRIPTTYPE eScriptType);
```

The header file `usrnsasm.h` has information on using the fileserver functions. It defines all functions and custom data types such as `struct SNASM_finddata_t`.

If the fileserver returns an error refer to the `errno` in your C run-time library documentation for a description of the problem.

Hitachi version of the library (libcrs.lib)

The files server version to use with the Hitachi SHC compiler contains a precoded wrapper functions for the system calls `debug_open()`, `debug_close()`, `debug_read()`, `debug_write()`, `debug_lseek()`.

NOTE: Do not transfer more than 32K in any SINGLE read or write command as not all communications are buffered by the files server transport functions.

This release includes:

- `.\libcrs` - contains source and object files for the transport functions.
- `.\sample` - contains a demonstration program 'sample.elf'.

Set the relative path for fileserver operations

Set or change relative path name of the default directory for your fileserver based operations in the Set fileserver directory dialog. (This can be the same as the directory you set in Source File Search Paths.)

Do the following:

1. Click Project, then click Set FileServer Root Directory.
2. Enter the Path of the default directory that you wish to use for fileserver operations.
3. Click OK.

NOTE: *If the display update rate interrupts the target when it is loading information to the fileserver directory on your computer, click Tools, Options...and select Fileserver Optimization.*

NOTE: *Any configuration commands you set are saved in a session file when you exit including software breakpoints and watches, and program update rates.*

Fileserver functions

debug_open

Syntax

header required

```
int debug_open (const char *filename, int flags [,int pmode]); include <usrsnasm.h>
```

Description

Opens a file.

Parameters

- *filename* is the name of the file to open.
- *flags**Open* are the flags for the type of operations desired.
- *pmode* is the permission mode.

Return value

Returns a file handle for an open file. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The setting:	Means that the file cannot be opened as:
SNASM_EACCES	It is read-only; or it is not a shared resource; or the path or filename are incorrect.
SNASM_EEXIST	The filename already exists.
SNASM_EINVAL	An invalid flags argument is defined.
SNASM_EMFILE	No file handles are available, close one or more files and try again.
SNASM_ENOENT	The file or path not found.

Remarks

The flags parameter can be a combination of the following definitions defined in `<sn_fcntl.h>`.

<code>SNASM_O_RDONLY</code>	Open for read only.
<code>SNASM_O_WRONLY</code>	Open for write only.
<code>SNASM_O_RDWR</code>	Open for read and write.
<code>SNASM_O_APPEND</code>	Writes done at end of file.
<code>SNASM_O_CREAT</code>	Create new file.
<code>SNASM_O_TRUNC</code>	Truncate existing file.
<code>SNASM_O_NOINHERIT</code>	File is not inherited by child process.
<code>SNASM_O_TEXT</code>	Text file.
<code>SNASM_O_BINARY</code>	Binary file.
<code>SNASM_O_EXCL</code>	Exclusive open.

`SNASM_O_BINARY` and `SNASM_O_TEXT` are required parameters when opening the file.

Within the open wrapper command for Hitachi, the flags parameter is translated from machine specific to a compiler independent format for translation transfer to the host. For example, `O_BINARY` will be converted to `SNASM_O_BINARY`.

The `pmode` argument is required only when `SNASM_O_CREAT` is specified. If the file already exists, `pmode` is ignored. Otherwise, `pmode` specifies the file permission settings, which are set when the new file is closed the first time.

`debug_open` applies the current file-permission mask to `pmode` before setting the permissions.

`pmode` is an integer expression containing one or both of the following manifest constants:

<code>SNASM_S_IREAD</code>	Reading only permitted.
<code>SNASM_S_IWRITE</code>	Writing permitted (permits reading and writing).
<code>SNASM_S_IREAD SNASM_S_IWRITE</code>	Reading and writing permitted.

debug_close

Syntaxheader required

```
int debug_close ( int file);#include <usrsnasm.h>
```

Description

Closes a file.

Parameters

int file is the handle returned by debug_open to the file.

Return value

debug_close returns 0 if the file closed successfully. If the return value is -1 an error occurred, refer to errno for the following setting.

The errno setting:	Means that the file cannot be closed because:
SNASM_EBADF	The file handle is invalid.

Remarks

CodeScape will close all open file handles when either the target is reset or when the CodeScape application is closed.

debug_read

Syntaxheader required

```
int debug_read( int file, char *ptr, int len);#include <usrnsasm.h>
```

Description

Reads data from a file.

Parameters

- *file* is the handle to the file.
- *ptr* is the pointer to buffer where read data is to be stored.
- *len* is the maximum number of bytes.

Return value

debug_read returns the number of bytes read. If the function tries to read at end of file, it returns 0. If the return value is -1 an error occurred, refer to errno for the following setting.

The errno setting:	Means that the data cannot be read because:
SNASM_EBADF	The file handle is invalid; or the file is not open for reading; or the file is locked.

Remarks

The debug_read operation occurs from the position of the file pointer. After a successful debug_read, the file position is at the return value number of bytes along the file.

Use debug_lseek to move the file position around.

The fileserver can also be told to read from standard in (STDIN). To do this issue the command debug_read(STDIN, buffersize, buffer_ptr). The Standard Input dialog box appears in CodeScope. Enter the text you require.

NOTE: *The number of characters is limited to the buffer size displayed.*

debug_write

Syntax header required

```
int debug_write ( int file, char *ptr, int len);#include <usrsnasm.h>
```

Description

Writes data to a file.

Parameters

- *file* is the handle to the file.
- *ptr* is the pointer to a buffer where read data is to be stored.
- *len* is the number of bytes.

Return value

debug_write returns the number of bytes written. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The errno setting:	Means that:
SNASM_EBADF	The file handle is invalid; or the file is not open for writing.
SNASM_ENOSPC	There is not enough available disk space.

Remarks

Two channels, SNASM_STDOUT and SNASM_STDERR, are used to display information on the Log tab of CodeScape's Input / Output window by default.

debug_lseek

Syntaxheader required

```
int debug_lseek ( int file, int offset, int origin);#include <usrsnasm.h>
```

Description

Moves a file to a specific location.

Parameters

file is the handle to the file.

offset is the number of bytes from origin.

origin is the flag indicating the origin.

Return value

debug_lseek returns the offset, in bytes, of the new position from the beginning of the file. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The errno setting:	Means that the:
SNASM_EBADF	File handle is invalid.
SNASM_ENIVAL	Origin value is invalid; or the specified location is before the start of the file.

Remarks

The origin flag can be any of the following predefined values:

SNASM_SEEK_SET	From start of file position.
SNASM_SEEK_CUR	From current position.
SNASM_SEEK_END	From end of file.

debug_getcwd

Syntaxheader required

```
char * debug_getcwd ( const char *buffer, int maxlen);#include <usrsnasm.h>
```

Description

Gets current working directory.

Parameters

buffer is the allocated space in which to store the path.

maxlen is the number of bytes in buffer.

Return value

debug_getcwd returns a pointer to the buffer. If the return value is NULL an error occurred, refer to errno for the following setting.

The errno setting:	Means that the:
SNASM_ERANGE	Path is longer than maxlen characters.

Remarks

The working directory is specified in CodeScape's Set Fileserver Path dialog box.

debug_chdir

Syntaxheader required

```
int debug_chdir ( const char *dirname);#include <usrsnasm.h>
```

Description

Changes the current working directory.

Parameters

dirname is the path of the new working directory.

Return value

debug_chdir returns a value of 0. If the return value is -1 an error occurred, refer to errno for the following setting.

The errno setting:	Means the:
SNASM_ENOENT	Specified path could not be found.

Remarks

The working directory is specified in CodeScape's Set Fileserver Path dialog box. The directory set in the dirname parameter must exist. The function may be used to change the drive and working directory.

For example, to change the drive and working directory to:

```
C:\windows\temp
```

Enter:

```
debug_chdir("c:\\windows\\temp");
```

NOTE: Use "\\" to describe a single "\" in a C string literal.

debug_mkdir

Syntaxheader required

```
int debug_mkdir ( const char *dirname);#include <usrsnasm.h>
```

Description

Creates a new directory.

Parameters

dirname is the path of the new working directory.

Return value

debug_mkdir returns a value of 0. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The errno setting:	Means that the directory cannot be created because:
SNASM_EEXISTS	It already exists.
SNASM_ENOENT	The specified path does not exist.

Remarks

The function creates only one directory per call.

debug_rmdir

Syntaxheader required

```
int debug_rmdir ( const char *dirname);#include <usrsnasm.h>
```

Description

Deletes a new directory.

Parameters

dirname is the path of the new directory.

Return value

debug_rmdir returns a value of 0. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The errno setting: Means that the directory cannot be deleted because:	
SNASM_EACCESS	It does not exist; or it is not empty; or it is the current working directory; or it is the root directory.
SNASM_ENOENT	The specified path was not found.

Remarks

The function deletes the specified directory. The directory must be empty and it cannot be the root directory or the current working directory.

debug_findfirst

Syntax

```
int debug_findfirst ( const char *filespec,
                     struct SNASM_finddata_t * fileinfo);
```

header required

```
#include <usrsnasm.h>
```

Description

Provides information about the first instance of a filename.

Parameters

- filespec* the target file specification.
- fileinfo* a pointer to structure to hold file specification.

Return value

debug_findfirst returns a search handle. If the return value is -1 an error occurred, refer to errno for one of the following settings.

The errno setting:	Means that the file specification:
SNASM_ENOENT	Is invalid.
SNASM_EINVAL	Could not be found.

Remarks

The function returns information on the first file that matches the file specification. The file specification can contain wildcards; for example, the following command searches for C files in the current working directory:

```
int hSearchHandle = debug_findfirst("*.c", &FileSpecification);
```

The file information structure contains 3 parameters:

```
unsigned longm_ulSize; /* file size */
unsigned longm_ulAttributes; /* file attributes */
charm_szFilename[260]; /* file name */
```

The attributes will be one of the following values:

```
SNASM_A_NORMAL /* Normal. File can be read or written to without
restriction. */
SNASM_A_RDONLY /* Read-only. File cannot be opened for writing, and
a file with the same name cannot be created. */
SNASM_A_HIDDEN /* Hidden file. Not normally seen with the DIR
command, unless the /AH option is used. Returns
information about normal files as well as files with
this attribute.*/
SNASM_A_SYSTEM /* System file. Not normally seen with the DIR
command, unless the /A or /A:S option is used. */
SNASM_A_SUBDIR /* Subdirectory. */
SNASM_A_ARCH /* Archive. Set whenever the file is changed, and
cleared by the BACKUP command. */
```

debug_findnext

Syntax

```
int debug_findnext ( int handle,
                    struct SNASM_finddata_t * fileinfo);
```

header required

```
#include <usrsnasm.h>
```

Description

Provides information about the next instance of a filename.

Parameters

handle the search handle supplied by debug_findfirst.
fileinfo is the pointer to a structure to hold file specification.

Return value

debug_findnext returns 0. If the return value is -1 an error occurred, refer to errno for the following setting.

The errno setting:	Means that:
SNASM_ENOENT	No more files matched the file specification.

Remarks

The function returns information on the next file that matches the file specification.

The file information structure contains 3 parameters:

```
unsigned longm_ulSize; /* file size          */
unsigned longm_ulAttributes; /* file attributes */
charm_szFilename[260]; /* file name      */
```

The attributes field shows one of the following values:

```
SNASM_A_NORMAL /* Normal. File can be read or written to without
restriction. */
SNASM_A_RDONLY /* Read-only. File cannot be opened for writing, and
a file with the same name cannot be created. */
SNASM_A_HIDDEN /* Hidden file. Not normally seen with the DIR command,
unless the /AH option is used. Returns information
about normal files as well as files with this
attribute. */
SNASM_A_SYSTEM /* System file. Not normally seen with the DIR command,
unless the /A or /A:S option is used. */
SNASM_A_SUBDIR /* Subdirectory. */
SNASM_A_ARCH /* Archive. Set whenever the file is changed, and
cleared by the BACKUP command. */
```

debug_findclose

Syntaxheader required

```
int debug_findclose ( int handle);#include <usrsnasm.h>
```

Description

Closes a search handle.

Parameters

handle is the search handle supplied by debug_findfirst.

Return value

debug_findclose returns 0. If the return value is -1 an error occurred and the operation failed to close the handle.

Remarks

Free up resources allocated to the file search operations.

_ASSERT

Syntaxheader required

```
int _ASSERT ( int nFlag );#include <usrsnasm.h>
```

Description

Halt and inform the user.

Parameters

nFlag Test nFlag, if expression evaluates to zero an assert is generated on host.

Return value

Returns 0.

Remarks

When an `_ASSERT` occurs and the flag evaluates to zero the host is told. The host prompts for an instruction and the `_ASSERT()` encountered dialog box appears, select:

- Yes to stop the program and tell CodeScape to put the cursor on the `_ASSERT` statement.
- No to ignore the assert and continue running the program.
- Cancel to ignore this and all further asserts.

You can set how CodeScape responds to an `_ASSERT` in the Options dialog box.

- On the Action tab, select Process Fileserver ASSERTs to display a message box when an `_ASSERT` occurs describing where it occurred.

Some compilers generate code that cause CodeScape to stop on the instruction following an `_ASSERT`. The sample program supplied includes a macro that ensures that an `_ASSERT` will stop on the line that generated it. The file also shows how all asserts can be removed with a global definition.

```
/*
 * Macro Redefinition of _ASSERT to ASSERT. This is performed to
 * cause the compiler to insert at least one opcode after the jsr
 * _ASSERT has returned it also permits the ASSERT code to be
 * included / removed based on a compiler define.
 */
#ifdef _DEBUG_BUILD_
    /* Since _ASSERT always return zero the expression will only be
     * evaluated once
     */
    #define ASSERT(X) while(_ASSERT(X)) { ; }
#else
    #define ASSERT(X)
#endif /* _DEBUG_BUILD_ */
```

debug_printf

Syntaxheader required

```
int debug_printf(char *format, ...);#include <usrnsasm.h>
```

Description

Prints data to the Log tab.

Parameters

Format Format control.

Argument Optional arguments.

Return value

The return value is the number of characters printed to the Log tab. Returns a negative value if an error occurs.

Remarks

The function formats and prints data to the Log tab on the Input / Output window.

If arguments follow the format string, the format string must contain argument output format specifications. The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications.

debug_runscript

Syntax

header required

```
int debug_runscript(const char *filename,  
                    SCRIPTTYPE eScriptType);  
#include <usrsnasm.h>
```

Description

Runs a script from the target.

Parameters

filename Name of the file to run

SCRIPTTYPE is either: `SCRIPTTYPE_JSCRIPT`, or `SCRIPTTYPE_VBSCRIPT`

Return value

The return value is the output printed to the Scripts tab.

Remarks

The target is stopped when a script is run. You must issue a 'Run()' in the script to continue target execution when the script is complete.

Statistical Profiling

What is profiling?

Statistical profiling

The Statistical Profiler included with CodeScape is an analysis tool for examining the run-time behavior of programs written for Hitachi SuperH™ microprocessors. It shows where your program spends its time so you can identify inefficient sections of code.

When your program executes, the Profiler takes regular samples of the processor's program counter (PC) at a configurable rate from 1Hz to 1kHz. From the address returned with each PC sample it can resolve which function your program is in at that moment in time.

Over a period of execution time, a profile of these samples is built-up which shows the approximate percentage of time spent in each function, the source line, and instruction of your code. These values are used to identify hot spots where your program spends disproportionate amounts of time.

Statistical profiling is passive, which means it does not modify your code to gather performance data. Therefore your program's execution time is not noticeably affected when the Statistical Profiler is running.

Function trace profiling

Function trace profiling lets you generate function and trace profiles which provide more detail about how specific functions are called.

Contact enquiries@codescape.com for more information about function trace profiling with CodeScape.

Setting up the Profiler

Prerequisites

This guide assumes you are using CodeScape to debug a supported Hitachi SuperH™ SH2, SH3 or SH4 target microprocessor. It also assumes that you know how to use CodeScape and are familiar with its regular functions and regions.

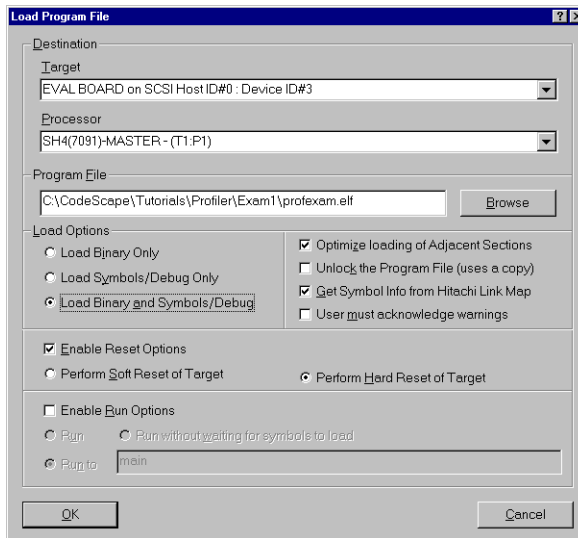
The example files used in this guide can be found in the CodeScape installation on your development computer under `Tutorials\Profiler\Exam1` and `Exam2`. If you did not select the Tutorials option during installation the files can be copied from the CodeScape CD.

NOTE: *You cannot use the Profiler while CodeScape's Simulator is running.*

Loading a program file

1. Start CodeScape.
2. In the Target window, right-click and click Load Program File...
3. Browse to the program file you want to profile.
4. Make the settings in the Load Program File dialog box as shown in *Figure* .

Load Program File options

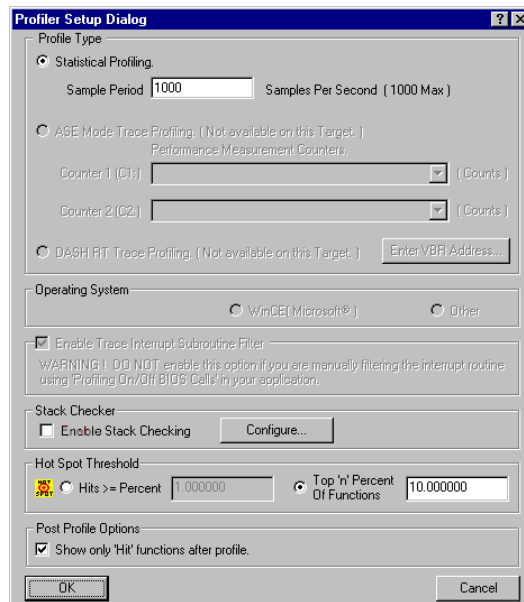


5. Click OK.
A progress box is displayed briefly and the program file is downloaded onto the target.

Setting the Profiler Setup options

1. On the Tools menu click Profiler.
The Profiler Setup dialog box appears in CodeScape, see *Figure* .
2. Select *Statistical Profiling*.
3. In the *Sample Period* text box, set the sample rate to 1000Hz.
This is the rate at which the Profiler samples the target's PC. There is no fixed rule to determine the optimum sample rate. The higher the sample rate, and the longer the duration of the profile, the more accurate the result will be. Normally the maximum sample rate of 1000Hz will not noticeably slow down the execution time of your program so we recommend that you always use this setting.
On some SH2 and SH3 processors running at slower clock speeds, the sample rate may have a greater effect on execution time but it should still be negligible.

Profiler Setup dialog



4. Select *Stack Checker* and click *Configure* if you want to specify a memory range for the stack overflow checker.



The Stack Check configuration dialog



You can add, remove, select or deselect memory ranges in the Add Range fields as shown in *Figure* . Either enter addresses directly into these fields or enter function names, an expression evaluator will resolve the correct addresses for you. The memory range you enter must be valid (for example, the *From* address must be before the *To* address).

See also “Checking for stack overflow” on page -349.

5. In the *Hot Spot Threshold* box select *Hits >= Percent* or *Top ‘n’ Percent of Functions* and enter a value for the hot spot threshold. This specifies the threshold at which functions will be tagged as a hot spot in the Profiler results window.
 - *Hits >= Percent* tags all functions that receive ‘n’ percent (or above) of the total number of hits counted during the profile.
 - *Top ‘n’ Percent of Functions* sorts the functions according to the number of times they were hit during the profile and tags the top ‘n’ percent as hot spots.In the example, a hot spot threshold of 10% is specified. This is a rather crude value because the example program has only a few functions. For larger, practical applications make use of the six decimal places provided to set up lower hot spot thresholds which have greater resolution.
6. Select *Show only ‘Hit’ functions after profile*.

With this selected the results of the Profile will show only those functions hit during the profile. All other functions are not displayed. This is useful feature profiling large programs to prevent the Profiler displaying large amounts of unwanted function data.
7. Click OK.
8. In the Profiler window click  and  to display Source and Disassembly regions as shown in *Figure* on page 340.

The Profiler window before performing a profile

List of functions in the example program. Only functions whose names can be resolved from debug information appear in the list.

Source region.

Disassembly region.

The screenshot shows the Profiler T1:P1 window. The top pane lists functions with their addresses and names. The middle pane shows the source code for `int main(void)`. The bottom pane shows the disassembly for the `main` function.

Function	Address	Disassembly
<code>long main()</code>	<code>0x0c010020</code>	
<code>void Initialise()</code>	<code>0x0c010034</code>	
<code>void BuildList()</code>	<code>0x0c010050</code>	
<code>void CleanUp()</code>	<code>0x0c010044</code>	
<code>struct LL_Node *GetNewNode()</code>	<code>0x0c010098</code>	
<code>struct LL_Node *GetLastNode()</code>	<code>0x0c0100ac</code>	
<code>void InsertNode(struct LL_Node *, struct L</code>		
<code>struct LL_Node *MallocNode()</code>	<code>0x0c010114</code>	
<code>void FreeHeap()</code>	<code>0x0c010148</code>	
TOTALS		

Address	Disassembly
<code>0x0c010016</code>	<code>0000 dc.w \$00</code>
<code>0x0c010018</code>	<code>0105 mov.w r0, 0</code>
<code>0x0c01001A</code>	<code>0000 dc.w \$00</code>
<code>0x0c01001C</code>	<code>001C mov.b 0(r0,r1), r</code>
<code>0x0c01001E</code>	<code>FF00 fadd r0, f</code>
<code>main</code>	<code>A006 bra \$0c010030</code>
<code>0x0c010022</code>	<code>0009 <s> nop</code>
<code>0x0c010024</code>	<code>E006 bsr Initialise</code>
<code>0x0c010026</code>	<code>0009 <s> nop</code>
<code>0x0c010028</code>	<code>E012 bsr BuildList</code>

Hits counter showing the number of hits on each function. This is also expressed as a percentage of the total number of hits on all the functions in the profile.

The “cl” column shows the cache line number for each function.

Setting Profiler breakpoints

This feature can be used to isolate a section of code for profiling. Due to the nature of statistical profiling it is usually preferable to profile the complete program to get a picture of how the program performs as a whole. For this reason we recommend that you do not use Profiler breakpoints when statistical profiling.

There are two types of Profiler breakpoints, start and stop. They are different to normal breakpoints. When a start breakpoint occurs an instruction is sent to the Profiler to start profiling. The Profiler runs until either a stop breakpoint occurs or until it is stopped manually by clicking the *Stop* button on the Profiler's toolbar. Unlike other breakpoints, on hitting a Profiler breakpoint the execution of the code does not stop, instead the program continues to run. You can set normal breakpoints in CodeScape that trigger different actions during profiling.

- You can only set one Profiler start breakpoint, but you can set multiple stop breakpoints.
- If the start breakpoint is at the beginning of a loop then you should set at least one stop breakpoint at the end of the loop otherwise a recursion can occur in the Profiler function call tree.
- Program execution does not stop when a Profiler breakpoint occurs.
- It takes a finite amount of time to process a Profiler breakpoint, therefore a frequently hit breakpoint increases the time it takes to profile the program but does not affect the processor time, so the Profiler's results remain accurate in terms of processor clock cycles.

Setting Profiler breakpoints in the Profiler window

1. In the Profiler's Disassembly region click on the function where you want profiling to start.
As a shortcut you can click on the required function in the Profiler's function list and the cursor in the Disassembly region will automatically move to the start of that function in the disassembled code.
2. Press F5 to insert a start breakpoint.
A green hand appears at the insertion point to indicate a Profiler start breakpoint. It is important that you do this in the Profiler's Disassembly region. In a normal Disassembly region in CodeScape you will have to set the trigger actions to define the breakpoint as a Profiler breakpoint, within the Profiler region this is automatic. See *"Setting Profiler breakpoints in CodeScape"* on page 343.

3. Scroll down to the function where you want profiling to stop.
Again, you can click on the required function in the Profiler's function list and the cursor in the Disassembly region will automatically move to the start of that function in the disassembled code.
4. Press F5 to insert a stop breakpoint.
A red hand appears at the insertion point to indicate a stop breakpoint. The Profiler automatically decides whether the breakpoint is a start or a stop based on your previous action.

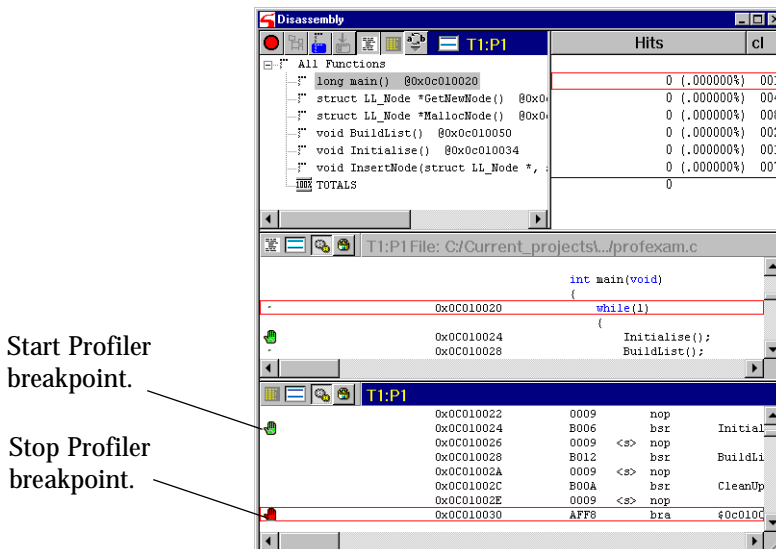
The Profiler window should now show the Profiler start and stop breakpoints in the Source and Disassembly regions as shown in *Figure* .

You can see that only one breakpoint is visible in the Source region. This is because the loop isolated in this example begins and ends on the same instruction in the source code.

In some circumstances the stop breakpoint can appear before the start breakpoint in the Source region. This is because the generated executable code can be different from the source code due to compiler optimizations.

NOTE: *Always check the Disassembly region when setting Profiler breakpoints to see exactly where the start and stop occur.*

Profiler start and stop breakpoints



Setting Profiler breakpoints in CodeScape

You can set up breakpoints in CodeScape that will trigger the Profiler to start. You insert the breakpoints in your program in the normal way using CodeScape and then configure the Trigger Actions to start or stop the Profiler upon hitting the breakpoints.



CAUTION: *If you use this method you must remember to enable the Profiler before running the program (stop/start button is green when the Profiler is enabled).*

1. In any region in CodeScape, right-click, point to Breakpoints and click Configure Breakpoint(s).
The Configure Breakpoint(s) dialog appears.
2. Add or select a breakpoint and specify any options you require.
3. On the Trigger Actions tab select *Cause processor profiling to* and specify whether the Profiler should start or stop when the breakpoint conditions are met.
4. Click OK.

NOTE: *Always check the Disassembly region when setting Profiler breakpoints to see exactly where the start and stop occur.*

Profiling a program and analyzing the results

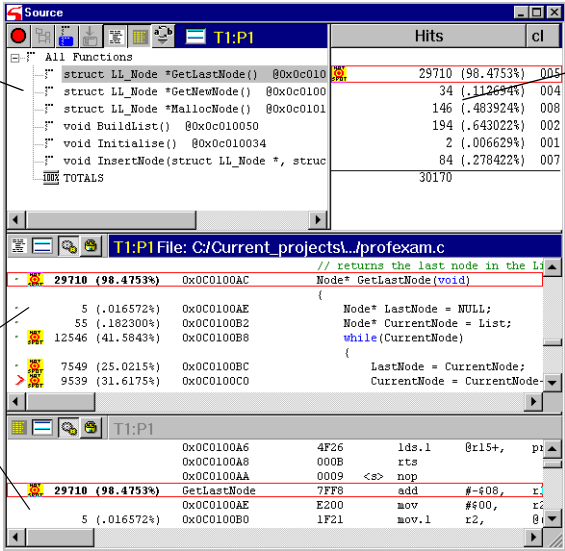
Running the Profiler

1. Click  (red) to start the Profiler.
The button is green when the Profiler is running and red when the Profiler is stopped.
2. Press F9 to run the program on the Target.
The Hits counter updates while the program is running.
The profile duration depends on the size of the program you are debugging. The longer you allow the program and the Profiler to run, the greater the number of samples taken, and therefore the greater the accuracy of the result. If the program executes once and then stops, you need to make sure that you run the Profiler for long enough to profile the whole program to get a complete picture of its performance.
3. Click  (green) to stop the Profiler.
The Profiler can take a few seconds or several minutes to resolve the data, depending on the size of the sample taken, after which you see the final results in the Profiler window as shown in *Figure on page 345*.

The Profiler results display

List of the functions hit during the profile. “Show only ‘hit’ functions” was selected on the Profiler Setup dialog.


Source and Disassembly regions showing the same hot spot.



Column showing the number of hits on each function. One hot spot is shown in this example.

Analyzing the profile data

The Hits column does **not** show the actual number of hits on each function during the profile, instead, statistical profiling gives the number of hits detected by a periodic sample. Therefore, the number of hits detected during statistical profiling is an indication of the time spent in each function rather than the actual number of hits on each function.

To locate the area in your code that causes a hot spot, click on  in the Hits column. The cursors in the Source and Disassembly regions automatically move to locate the hot spot in the code. A function tagged with a hot spot indicates that the number of hits detected on that function exceeds the threshold requirements specified on the Profiler Setup dialog.

NOTE: Sometimes the Profiler cannot resolve the function name from the address returned by the PC sample, usually because of insufficient debug data. In this case the hits are scored against “Other functions” in the function list. The number hits on “Other functions” can include hits from more than one function.

The example shown in *Figure* is a simple program that creates a single-ended linked list of ascending numbers from 1 to 1000, clears the list, and then starts again.

The hot spot in the Hits column shows that the program spent 98.5% of the time during the profile sample in the function `GetLastNode`. In this program spending 98% of the execution time in just one function is extremely inefficient.

In the function `GetLastNode`, the program locates the last entry in the list by searching from the start of the list until it reaches the last node. The problem occurs in a particular loop in `GetLastNode`, this is highlighted by another three hot spots beneath `GetLastNode` which are shown in the Source region. See *Figure* on page 347.

Analyzing the Profiler results

The loop shown in Source where the program spends a disproportionate amount of time.

The total number of hits (bold) shown next to a function is the sum of hits (non-bold) on all the source lines belonging to that function.

GetLastNode shown at instruction level in the Disassembly region.

Source		Hits	cl
All Functions			
struct LL_Node *GetLastNode() @0x0c0100		29710 (98.4753%)	005
struct LL_Node *GetNewNode() @0x0c0100		34 (.112694%)	004
struct LL_Node *MallocNode() @0x0c0101		146 (.483924%)	008
void BuildList() @0x0c010050		194 (.643022%)	002
void Initialise() @0x0c010034		2 (.006629%)	001
void InsertNode(struct LL_Node *, struct		84 (.278422%)	007
TOTALS		30170	

T1:P1 File: C:/Current_projects/_profexam.c	
29710 (98.4753%)	Node* GetLastNode(void)
5 (.016572%)	Node* LastNode = NULL;
55 (.182300%)	Node* CurrentNode = List;
12546 (41.5843%)	while(CurrentNode)
7549 (25.0215%)	{
9539 (31.6175%)	LastNode = CurrentNode;
	CurrentNode = CurrentNode;
	}

T1:P1	0x0C0100A6	4F26	lds.l	@t15+,	p1
	0x0C0100A8	000B	rts		
	0x0C0100AA	0009	<s>	nop	
29710 (98.4753%)	GetLastNode	7FF8	add	#-608,	r1
	0x0C0100AE	E200	mov	#600,	r2
5 (.016572%)	0x0C0100B0	1F21	mov.l	r2,	@t15+

Example 1

For GetLastNode, the number in bold in the Source region shows the total number of hits on that function, the numbers beneath show how many times each line in that function was hit during the profile.

You can see the number of hits at instruction level in the Disassembly region.

The Profiler's results show that rewriting GetLastNode so that it does not search from the start of the list each time will eliminate the hot spot and speed up the program. The program has been rewritten to show this in *Example 2*.

Profile results for example 2

Profiler: T1:P1		Hits	cl
All Functions			
long main() @0x0c010020		5 (.015923%)	001
struct LL_Node *GetNewNode() @0x0c01009A		4156 (13.2352%)	004
struct LL_Node *MallocNode() @0x0c01009C		7827 (24.9259%)	008
void BuildList() @0x0c010050		10861 (34.5880%)	002
void Initialise() @0x0c010034		12 (.038215%)	001
void InsertNode(struct LL_Node *, struct LL_Node *) @0x0c0100A8		8540 (27.1965%)	007
TOTALS		31401	
T1:P1 File: C:/Current_projects/.../profexam.c			
// returns a new Linked List			
Node* GetNewNode(void)			
{			
2767 (8.81182%) 0x0C01009C			return MallocNode();
0x0C0100A8			}
T1:P1			
1 (.003184%) 0x0C010092	7F10	add	#s10,
0x0C010094	4F26	lds.l	@r15+,
0x0C010096	000B	rts	
0x0C010098	0009 <s>	nop	
4156 (13.2352%) GetNewNode	4F22	sts.l	pt,
724 (2.30565%) 0x0C01009C	B03A	bsr	MallocN
0x0C01009E	0009 <s>	nop	
708 (2.25470%) 0x0C0100A0	4F26	lds.l	@r15+,

Example 2

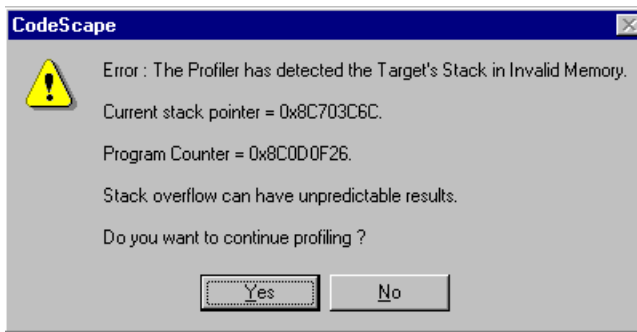
Here, the program has been rewritten to use a local pointer to remember the last inserted node. This means that GetLastNode does not appear in the profile results because the function is no longer used and has not been hit during the profile.

You can see from the hot spots that most of the work is shared fairly evenly between four functions, this is a much more efficient use of execution time.

Checking for stack overflow

If you selected *Enable Stack Checking* on the Profiler Setup dialog, the Profiler monitors the stack address during program execution when the Profiler is running. The stack address is sampled at the same rate that the PC is sampled by the Profiler.

If the Profiler detects that the stack pointer goes beyond the memory range you configured for the stack, the following message is displayed:



Current stack pointer identifies the offending out of range stack address.

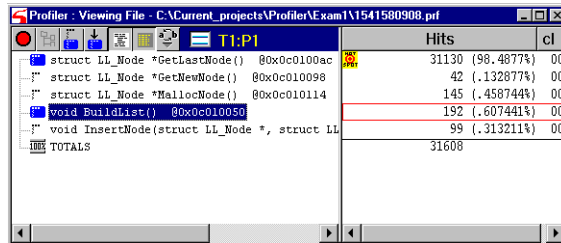
Program Counter gives the address of the current line of code when the stack overflow was detected.

A reported stack overflow is the first *detected* stack overflow, but because the stack pointer is sampled it might not be the first *actual* stack overflow.


Navigating in the Profiler window


Tagging functions

In the function list in the Profiler window you can double-click on a function to tag it. Tagged functions are indicated by a blue rectangle.




	Hits	cl
struct LL_Node *GetLastNode() @0x0c0100ac	31130 (98.4877%)	00
struct LL_Node *GetNewNode() @0x0c010098	42 (.132877%)	00
struct LL_Node *MallocNode() @0x0c010114	145 (.458744%)	00
void BuildList() @0x0c01005b	192 (.607441%)	00
void InsertNode(struct LL_Node *, struct LL	99 (.313211%)	00
TOTALS	31608	

The  button toggles the display to show tagged functions, untagged functions, or all functions.

In large function lists click  to show the next tagged function in the list.

Sorting functions

Click  to sort the functions alphabetically, either ascending or descending.

In the Hits column, click *Hits* or *cl* to sort the list either by Hits (ascending or descending) or by cache line number (ascending or descending).

Hits	cl
------	----

Finding functions

This feature is useful for large function lists. You can search for parts of words, whole words, parts of addresses, or whole addresses.

When the Profiler window is the active region, type in part of its name or address and an expression evaluator will locate the function. The search begins from the cursor and looks for exact matches first, then the nearest matches in descending order.

To continue the search press F3 or Enter.

Renaming functions

You can rename any function in the Profiler's function list. This does not change the function's name in the source code, only the Profiler's view changes.

1. Right-click on the function and click Rename Function...
The Rename Function dialog box appears.
2. In the dialog box type the new function name and click OK.
The function is renamed in the Profiler's function list.

Saving the profile data

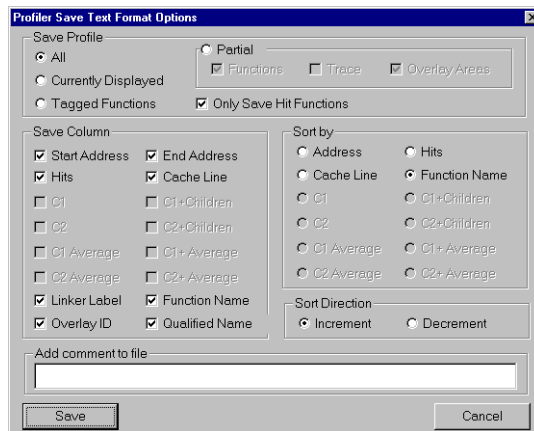
As profile data

1. Right-click in the Profiler window, point to File, Save Profile and click Save Profile (.PRF)...
2. In the dialog box browse to the location where you want to save the profile data, name the file and click Save.

You can close the session and return to it later and reload the profile results. Only the profile data and Profiler Breakpoints are saved, the Profiler display settings and hot spot thresholds are lost. The data structure of this file is given in detail in *“Appendix B: Profiler file format” on page 356.*

As text

1. Right-click in the Profiler window, point to File, Save Profile and click Save Profile (.TXT)...
2. Fill in the options that you want to save in the text file and click Save.
The options are described in *“Text format save options” on page 352.*



Text format save options

Save Profile	
--------------	--

All	Saves the profile data for all the functions listed in the Profiler results.
Currently Displayed	Saves the profile data for only the functions that are visible in the function list in the Profiler window.
Tagged Functions	Saves the profile data for only the functions that are tagged in the function list.
Only Save Hit Functions	Saves only the function that were hit during then profile. This prevents saving large amounts of unwanted data in large programs.
Partial	
Functions	Saves data associated only with functions.
Overlay Areas	Saves additional overlay information if the program contains overlay information.
Save Column	
Start Address	Saves the start address of each function, this is the address shown in the Profiler function list.
End Address	Saves the end address of each function. (Not shown in the function list.)
Hits, Cache Line	Saves the data in the Hits and Cache Line columns.
Linker Label, Function Name	Saves each function's name and linker label.
Overlay ID	Saves each function's overlay ID if overlays are in use.
Qualified Name	Saves the function's qualified name (function name + file name + class name) if the function is not unique.
Sort by	
Address, Hits, Cache Line or Function Name	Sorts the profile data by Address, Hits, Cache Line or Function Name.
Sort Direction	
Increment, Decrement	Direction in which the columns are to be sorted in the text file.
Add comment to file	Allows you to add comments to the header of the text file.
Save Profile	

Appendix A: Profiler's shortcut menu

Select...	To...
File	Load or save profile information. Profiles are saved using the extension .prf.
Enable Profiler	Start or stop the profiler.
One Pass Between Breakpoints	Set one pass between a Profiler start breakpoint and a profiler stop breakpoint.
Remove All Profiler Breakpoints	Remove all Profiler breakpoints.
Trace Tree Profile Display	Display a tree showing each function, the functions that called it, and the functions it called. Trace profiling only.
Function Profile Display	Display a simple list of the functions in your program in the Profiler window.
Function Profile Filter	Arrange the view to show one of the following: all functions, all tagged functions, or all untagged functions.
Untag All	Untag all currently tagged functions.
Sort	Arrange the column view of the active Function Profile Display.
Source Display	View your program's original source code.
Disassembly Display	View your program at instruction level (assembly code).
Rename Function...	Enter a new name for a specific function.
Profiler Display Setup...	Specify the profile display options.
Optimize...	Instruction cache optimization. Trace profiling only.

Select...	To...
Setup...	Specify options for statistical profiling or function trace profiling.

Appendix B: Profiler file format

This appendix describes the Profiler File Format that is created when you save profile data as a PRF file. This file can be subsequently reloaded into the Profiler so you can study its contents in the Profiler's results display.

The following tables of data structures that define the file format are in the order that they appear in the file. The file format consists of six structures:

1. Profiler file ID
This is the label at the start of the file which initially identifies it as a profiler file.
2. File header
This is the file's header which contains global information about the profile.
3. Overlay area data
This is a description of an overlay area.
4. Function data
This contains data of a function that was called during the profile.
5. Trace function call
This describes a function call which has its own function data unique to that particular call path.
6. Trace function return
This describes a function return.

NOTE: *Incorrect counter data (see Function Data Structure and Trace Function Call Structure) can occur when there is more than one Profile Event (JSR, RTS, Interrupt, Exception, RTE, BSR, BSRF) in the SH4's pipeline. This means there is no way of knowing which event incremented which counter and by what amount. This is an anomaly of the SH4's tracing hardware and the effect on the Profiler's results is usually negligible.*

Profiler file ID structure

Byte	Type	Profiler File ID
0	Byte	0x50 'P'
1	Byte	0x52 'R'
2	Byte	0x4f 'O'
3	Byte	0x46 'F'
4	Byte	0x49 'I'
5	Byte	0x4c 'L'
6	Byte	0x45 'E'
7	Byte	0x52 'R'
8	Byte	0x20 ' '
9	Byte	0x46 'F'
10	Byte	0x49 'I'
11	Byte	0x4c 'L'
12	Byte	0x45 'E'
13	Byte	0x3a ':'

File header structure

The file header indicates the type of profile. If it is a statistical profile the *Number Of Function Calls* and *Trace Information Offset* will be set to zero along with all the *Counter* data. In this case only the *Function Hits* data is valid. If the file represents a function trace profile all the data is valid.

Byte	Type	File Header
0 1 2 3	Unsigned Long	(LSB) Size of Header to follow. (0x00000055) (MSB)
4	Byte	File Version
5 6	2 bytes	Spare Bytes
7	Byte	Instruction Cache : 0x01 Enabled (8k) , 0x00 Disabled.
8	Byte	Day (0 - 31)
9	Byte	Month (0 - 12)
10	Byte	Year eg. 98 - 99 - 00 - 01
11	Byte	Hour (0 - 24)
12	Byte	Min (0 - 60)
13	Byte	Second (0 - 60)
14	Byte	Type of Profile 0x01 - Statistical Profile - in this mode only the Function Hits are valid. All the counter data in every structure will be zero. 0x02 - Trace Profile - all counters and hits are valid.
15 16 17 18 19 20 21 22	Unsigned Quadword	(LSB) Overall accumulative total number of Function Hits (MSB)

Byte	Type	File Header
23 24 25 26 27 28 29 30	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 (MSB)
31 32 34 35 36 37 38	Unsigned Quadword	(LSB) Overall accumulative total of Counter 1 minus the functions it called. (MSB)
39 40 41 42 43 45 46	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 (MSB)
47 48 49 50 51 52 53 54	Unsigned Quadword	(LSB) Overall accumulative total of Counter 2 minus the functions it called. (MSB)
55	Byte	What Counter 1 represents, see the Counter Description Table at the end of the document.
56	Byte	What Counter 2 represents, see the Counter Description Table at the end of the document.
57 58 59 60	Unsigned Long	(LSB) Number of Overlays (MSB)

Byte	Type	File Header
61 62 63 64	Unsigned Long	(LSB) Number of functions (MSB)
65 66 67 68	Unsigned Long	(LSB) Number of function calls. If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
69 70 71 72	Unsigned Long	(LSB) Overlay Information Offset Offset into this file from the End of the Header to the Overlay Information. This is only valid if the Number Of Overlays field is greater than zero. (MSB)
73 74 75 76	Unsigned Long	(LSB) Function Information Offset Offset into this file from the End of the Header to the Function Information. (MSB)
77 78 79 80	Unsigned Long	(LSB) Trace Information Offset Offset into this file from the End of the Header to the Trace Information (function calls/returns). If Byte 14 is set to 0x01 (for Statistical Profile) then this will be set to zero. (MSB)
81 82 83 84 85 86 87 88	8 bytes	Spare.

Overlay area data structure

The overlay data that follows is a list of Overlays that exist within the program. The data is repeated *Number Of Overlays* times. *Number Of Overlays* can be found in the file header.

Byte	Type	Overlay Area Data
0 1 2 3	Unsigned Long	(LSB) Size of Overlay Area Data to follow. (MSB)
4 5 6 7	Unsigned Long	(LSB) Overlay Area Index ID. (MSB)
8 9 10 11	Unsigned Long	(LSB) Start Address of Overlay Area. (MSB)
12 13 14 15	Unsigned Long	(LSB) End Address of Overlay Area. (MSB)
16 17 18 19	Unsigned Long	(LSB) Current Overlay ID (at the time of saving). (MSB)

Function data structure

The Function Data that follows is a list of functions called during the profile with accumulative totals for that function regardless of call path. The data is repeated *Number Of Functions* times. *Number Of Functions* can be found in the file header.

Byte	Type	Function Data
0 1 2 3	Unsigned Long	(LSB) Size Of Function Data to Follow. (MSB)
4	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR + ?). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
5 6	Unsigned Word	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. 0x00000002 - This function was Tagged (Selected) by the user. All other bit-fields will be 0.
7 8	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k) , 0xFFFF Non Cacheable (MSB)
9 10 11 12	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)
13 14 15 16	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
17 18 19 20	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of an overlay. (MSB)

Byte	Type	Function Data
21 22 23 24	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an Overlay. (MSB)
25 26 27 28	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
29 30 31 32 33 34 35 36	8 Bytes	Spare.
37 38 39 40 41 42 43 44	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function. (MSB)
45 46 47 48 49 50 51 52	Unsigned Quadword	(LSB) Accumulative total of Counter 1 for this function. (MSB)
53 54 55 56 57 58 59 60	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function. (MSB)

Byte	Type	Function Data
61 62 63 64 65 66 67 68	Unsigned Quadword	(LSB) Accumulative total of Counter 2 for this function. (MSB)
69 70 71 72 73 74 75 76	Unsigned Quadword	(LSB) Accumulative total of Counter 2 minus the functions it called for this function. (MSB)
77 78	Word	(LSB) The size of the following function name. ('n1') (MSB)
79	String	Function Name. The string is of the size specified in the previous field.
79 +'n1'	Word	(LSB) The size of the following Linker Label. ('n2') (MSB)
79 +'n1' + 2	String	Linker Label. The string is of size specified in the previous field. The Linker Label is the label that was used to link this function.
79 +'n1' +2+'n2'	Word	(LSB) The size of the following Qualified Name. (MSB)
79 +'n1' +2 +'n2' +2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace function call structure

The position of the *Trace Function Call/Return* structures in the file defines the function's call/return relationship. If the file represents a statistical profile as opposed to a function trace profile there will be no *Trace Function Call* or *Trace Function Return* structures in the file.

The function trace data (function calls/returns) consists of two data structures *Trace Function Call* and *Trace Function Return*. To distinguish between them the first byte after the data size is 0xFF for a function call and 0x00 for a return. There is a file offset to this data that is stored in the File Header. Also stored in the File Header is the *Number of Function Calls*. The number of following Trace Function Calls is coupled with an equal amount of Trace Function Returns.

Byte	Type	Trace Function Call
0 1 2 3	Unsigned Long	(LSB) Size of Trace Function Call Data to follow. (MSB)
4	Byte	0xFF Function Call identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is present in the matching 'Return From Function' data. (MSB)
9	Byte	Function Description. 0x00 - Unknown. 0x01 - Function. 0x02 - Interrupt/Exception (unknown VBR +). 0x03 - Exception VBR + 0x100. 0x04 - Exception VBR + 0x400. 0x05 - Interrupt VBR + 0x600. 0x06 - Exception DBR. 0x07 - User Defined Block.
10 11	Unsigned Long	Attributes (Bit Field) 0x00000001 - This function has possible incorrect counter data. All other bit-fields will be 0.
12 13	Unsigned Word	(LSB) Cache Line Number. 0 - 256 (256 lines * 32 Bytes = 8k), 0xFFFF(-1) Non Cacheable (MSB)

Byte	Type	Trace Function Call
14 15 16 17	Unsigned Long	(LSB) Minimum Program Counter of Function Scope. (MSB)
18 19 20 21	Unsigned Long	(LSB) Max Program Counter of Function Scope. (MSB)
22 23 24 25	Unsigned Long	(LSB) Overlay Area Index ID ID 0xFFFFFFFF = Not part of any overlay. (MSB)
26 27 28 29	Unsigned Long	(LSB) Overlay ID ID 0xFFFFFFFF = Not an overlay. (MSB)
30 31 32 33	Unsigned Long	(LSB) Function Group ID ID 0xFFFFFFFF = No Group. (MSB)
34 35 36 37 38 39 40 41	8 Bytes	Spare.
42 43 44 45 46 47 48 49	Unsigned Quadword	(LSB) Accumulative total number of Function Hits for this function with this call path. (MSB)

Byte	Type	Trace Function Call
50 51 52 53 54 55 56 57	Unsigned Quadword	(LSB) Accumulative total of Counter 1 for this function with this call path. (MSB)
58 59 60 61 62 63 64 65	Unsigned Quadword	(LSB) Accumulative total of Counter 1 minus the functions it called for this function with this call path. (MSB)
66 67 68 69 70 71 72 73	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 for this function with this call path. (MSB)
74 75 76 77 78 79 80 81	Unsigned Quadword	(LSB) Accumulative Total of Counter 2 minus the functions it called for this function with this call path. (MSB)
82 83	Word	(LSB) The Size of the Following Function Name. ('n1') (MSB)
84	String	Function Name. The string is of size specified in the previous field.
84 + 'n1'	Word	(LSB) The Size of the following Linker Label. ('n2') (MSB)

Byte	Type	Trace Function Call
84 + 'n1' + 2	String	Linker Label. The String is of size specified in the previous field. The Linker Label is the label that was used to link this function.
84 + 'n1' + 2 + 'n2'	Word	(LSB) The Size of the following Qualified Name. (MSB)
84 + 'n1' + 2 + 'n2' + 2	String	Qualified Name. The string is of size specified in the previous field. The Qualified Name is used to specify a function in greater detail (filename, class etc.) in the event of a duplicate function name.

Trace function return structure

Byte	Type	Trace Function Return
0 1 2 3	Unsigned Long	(LSB) Size Of Trace Function Return Data to Follow. (0x00000005) (MSB)
4	Byte	0x00 Function Return Identifier.
5 6 7 8	Unsigned Long	(LSB) Unique Call Number. This number is to ensure that the return Unique Number matches the Call Unique Number. (MSB)

Counter description table

The *Counter Description Table* describes all the counters available in the SH4's ASE mode debug facility.

Counter Description Number	Counter Description	Count/Cycles
0x01	Operand access (read/with cache)	Count
0x02	Operand access (write/with cache)	Count
0x03	UTLB miss	Count
0x04	Operand cache read miss	Count
0x05	Operand cache write miss	Count
0x06	Instruction fetch (with cache) - 2 instructions fetched simultaneously	Count
0x07	Instruction TLB miss.	Count
0x08	Instruction cache miss	Count
0x09	All operand access	Count
0x0a	All instruction access - 2 instructions fetched simultaneously	Count
0x0b	On-chip RAM operand access	Count
0x0d	On-chip I/O space access	Count
0x0e	Operand access (read + write/with cache)	Count
0x0f	Operand cache read + write miss	Count
0x10	Branch instruction	Count
0x11	Branch taken	Count
0x12	BSR/BSRF/JSR	Count
0x13	Instruction execution	Count
0x14	2- instruction simultaneous execution	Count
0x15	FPU instruction execution	Count

Counter Description Number	Counter Description	Count/Cycles
0x16	Interrupt (normal)	Count
0x17	Interrupt (NMI)	Count
0x18	TRAPA instruction execution	Count
0x19	UBC-A match	Count
0x1a	UBC-B match	Cycles
0x21	Instruction cache fill	Cycles
0x22	Operand cache fill	Cycles
0x23	Elapsed time	Cycles
0x24	Pipeline freeze (by cache miss/instruction)	Cycles
0x25	Pipeline freeze (by cache miss/data)	Cycles
0x27	Pipeline freeze (by branch instruction)	Cycles
0x28	Pipeline freeze (by CPU register)	Cycles
0x29	Pipeline freeze (by FPU)	Cycles

DashScript

Before you start

This document tells you how to establish communications between the DASH and your computer, and the DASH and the target for first use.

It supports:

- DASH2 - 7615, 7047, and 7622.
- DASH3 - 7709A, 7729R, 7729, and 7727.
- DASH4 - 7750, 7750S, and 7751.

DashScript COM object scripting

DashScript is a COM object that supports Microsoft® JScript® and VBScript macro scripts for controlling the DASH. The DashScript commands let you communicate with a DASH and connected target via a windows scripting host. You can use the commands available in either script language to add commands of your own.

The DashScript debugging commands include reading and writing memory and registers, loading and running programs, and setting breakpoints. You communicate with the application running on the target system via the DASH's comms channels. The channels are supported at one end by DashScript, and at the other by the 'ASE BIOS' calls available via the debug mechanism on the target system.

For details about the Microsoft Windows Script Host, JScript, and VBScript commands, connect to: <http://msdn.microsoft.com/scripting>

Configuring the communications

Complete the following steps to establish communications between the DASH and your computer, and the DASH and the target for first use.

1. Install then create a registry entry for DashScript. See *“Installing DashScript” on page 377*.
2. Install TCP/IP. See *“Installing TCP/IP” on page 378*.
3. Configure the DASH’s IP address using NetFlash, and confirm that NetFlash reports the DASH is present. See *“Assigning an IP address and installing firmware” on page 379*.
4. Create and run a DashScript that initializes the target’s RAM and loads the Extended debug stub. See *“Initializing the DASH for debugging” on page 382*. Example scripts are supplied with this release, for more information see *“Examples for debugging a new target board” on page 385*.

NOTE: *DashScript will fail to run if it cannot find the DASH or target. See “Troubleshooting” on page 388.*

Installing DashScript

1. Insert the Release CD into your computer's CD drive.

2. Follow the instructions on screen.

After installation DashScript and it's associated applications can be accessed on the Start menu in the CodeScape program group.

Installing TCP/IP

For correct communication with the DASH you must have the TCP/IP suite installed as part of your computer's network settings.

Network

If your computer is connected to a network you will probably have TCP/IP already installed and you can continue with the section *“Assigning an IP address and installing firmware” on page 379*. Ask your network administrator if you are not certain about this.

Peer-to-peer

If you are connecting peer-to-peer you do not need to know any network details but you do need a network card installed in your computer. Refer to the documentation supplied with your network card for details of how to install and configure it. Refer to Windows online help documentation for details of how to install TCP/IP on your computer and then configure the IP settings. Below are typical public values you can use for the IP address and subnet mask in a peer-to-peer connection:

- Your computer's IP address: 192.168.0.1
- Your computer's subnet mask: 255.255.255.0
- The DASH's IP address 192.168.0.2.

The IP address space 192.168.0.x is allocated for private use by the Internet Assigned Numbers Authority. In this example the last digits (1 and 2) are the device IDs assigned to your computer and the DASH respectively. You can assign any number from 1 to 254 as device IDs. Do not assign 0 or 255, these numbers are reserved.

Assigning an IP address and installing firmware

Run NetFlash to:

- Assign or change the IP address of a DASH on your network.
You can either allocate a static IP or you can use DHCP to request a dynamic address. Ask your network administrator which method you should use to assign your DASH an IP.
To use a static IP you need the subnet mask or netmask address for your network, and a spare, valid IP address on the network to assign to the DASH.
- Upgrade the firmware on a DASH target or revert to an older version.

The following versions of DASH firmware support DHCP:

- DASH4 v6.4.5 or later.
- DASH3 v5.7.5a or later.
- DASH2 v6.0.2a or later.

Notes

1. *If the DASH firmware supports DHCP, it is active by default.*
2. *If your DASH does not have firmware that supports DHCP, you can get an upgrade from www.codescape.com*
3. *When you upgrade the firmware on a DASH the state of DHCP remains as it was before you changed the firmware. To enable DHCP for a DASH after a firmware upgrade, remove the DASH from the target list then add it with DHCP enabled.*

Searching for a DASH and assigning it an IP address

1. Run NetFlash.
2. Click *Search/Add*.
The Search For DASH IP dialog appears.
3. In the *Hardware No* box, enter the last five digits of the DASH's serial number. The serial number is shown on the base of the DASH.
4. Click *Search*.
If the DASH is on your network its current network settings appear in the Add Selected Target dialog.
5. Do one of the following:
 - Click OK if you want to accept the current settings.
 - Select *DHCP Enabled* then click OK.
The *IP No.* field becomes inactive and the DASH uses DHCP to request a dynamic address.
 - In the *IP No.* field enter a valid spare IP address on your network then click OK.
The new target appears in the *DA ID* box and DHCP use, model identification, product serial number, firmware version and type appear in the fields below.

Changing a DASH's IP address

1. Run NetFlash
2. Select a DASH from the *DA ID* box.
The current state of DHCP usage, model identification, product serial number, current firmware version, and firmware type appear in the fields below.
3. Click *Change IP*.
The Change IP Number dialog appears.
4. Do one of the following:
 - Select *DHCP Enabled*.
The *IP No.* field becomes inactive and the DASH uses DHCP to request a dynamic address.
 - In the *IP No.* field enter a valid spare IP address on your network.
5. Click OK.
The target's new IP address appears in the *DA ID* box next to the DASH hardware number.

Upgrading the DASH firmware

1. Run NetFlash.
2. Select a DASH from the DA ID box.
The current state of DHCP usage, model identification, product serial number, current firmware version, and firmware type appear in the fields below.
3. Click Reflash.
The Open dialog appears.
4. Browse for the version of firmware you want to upgrade to, select the *.fsh file and click Open.
5. Click OK.
A progress bar appears and indicates the time remaining for the firmware change. When the firmware change is complete, power cycle the DASH.

CAUTION: Do not power cycle the DASH while the firmware is being upgraded, this may permanently damage the target.

CAUTION: *Make sure that you use the correct version of the firmware for your target processor. Using the wrong firmware for your target may permanently damage it.*

Removing a DASH from the target list

1. Run NetFlash.
2. Select a DASH from the DA ID box.
The model identification, product serial number, current firmware version, and firmware type appear in the fields below.
3. Click Remove.
4. You are asked to confirm that you want to remove the selected target. Do one of the following:
 - Click Yes to remove the target and continue.
 - Click No to leave the target as it is and continue.

Initializing the DASH for debugging

Initializing for 7615, 7622, 7709A, 7729R, 7729, 7727, 7750, 7750S, and 7751

Once the DASH is communicating with the ASE stub you have access to basic debugging information. To get full debugging information the Extended debug stub must be loaded into external RAM on the target.

The DashScript commands let you create and run a script to:

- Specify how the target's RAM initializes.
You can initialize the target's RAM with a boot ROM, or write a script to program the control memory mapped registers directly.
- Specify the reset method. (SH4 processors only.)
- Set the JTAG clock frequency.
- Load the Extended debug stub.
You can load the Extended debug stub with a BIOS call request from the boot ROM to an address specified in the boot ROM, or force it to load a specified address.

For information about the ASE and Extended debug stubs and example boot ROM code for SH3 and SH4 processors refer to the appropriate "Debug Interface" section.

The 7047 has a single debug stub that resides in ASE RAM on the target processor. This is loaded each time the DASH resets the target. For information see "Initializing for 7047" on page 387.

Configuring the target

The *ConfigureTarget* command lets you:

- Initialize the target's RAM with a boot ROM via the *runBootRom* parameter. You can also initialize the target's RAM with a script.
- Specify the reset method with the *resetMethod* parameter. Refer to “*DASH Connector Configurations*”.
- Set the JTAG clock frequency with the *JTagFreq* parameter.

Do not use the WriteRegister command in a RAM initialization script or unexpected results may occur.

Specifying the reset method (SH4 only)

If you are configuring an SH4 you can specify how the DASH generates a target reset. *Pin Reset* is preferable to *UDI Reset* because it enables the entire target hardware to be reset. Refer to “*DASH Connector Configurations*” for pin details.

- Pin Reset uses the DBG_RST# line. For this to operate the target system must have a functional DBG_RST# and RESETIN# line available to the DASH. The DASH uses the RESETIN# line as confirmation that the DBG_RST# line assertion occurred correctly.
- UDI Reset can be used if there is no hard-wired reset mechanism on the target hardware. With this specified, on a hard reset, the DASH sends a serial command on the UDI TDO line to reset the target.

Setting the JTAG clock frequency

The JTAG clock frequency (TCK) is preset to 20MHz which is suitable for most SH2, SH3 and SH4 applications. If you have specific debugging requirements you may want to change the JTAG clock frequency to suit your target hardware.

The following general rules apply:

- SH4: the JTAG clock frequency must be less than the target hardware's peripheral clock frequency.
- SH2 and SH3: the JTAG clock frequency must be less than half the target hardware's CPU clock frequency.

Notes

1. *Refer to Hitachi documentation to determine the correct JTAG clock frequency to use for your target hardware.*
2. *The JTAG clock must remain stable during use, do not attempt to single step over an instruction that changes the contents of the Frequency Control Register, FRQCR.*

Loading the Extended debug stub

The *LoadExtendedStub* command loads the Extended debug stub to the memory set up by a script or boot ROM.

There are two methods of loading the Extended debug stub:

- As a result of a BIOS call request from the boot ROM to an address specified in the boot ROM.
- Force it to load at a specified address using the *LoadExtendedStub* command.

Examples for debugging a new target board

On the CD are two example scripts that demonstrate how to configure the DASH Settings. The scripts are called `example.js` and `example.vbs`. The function examples are from `example.js`.

Additional JScript and VBScript examples are supplied on the release CD including example boot ROM code.

The JScript defines variables and the VBScript defines constants for:

- Breakpoint types, actions, script types, expression types, address masks, access sizes, and access types.
- *ConfigureTarget* as:

```
TARGETBOOT_RUN           = 1;
TARGETBOOT_NORUN         = 0;
TARGETRESET_HALT         = 1;
TARGETRESET_RUN          = 0;
TARGETRESET_PIN          = 0;
TARGETRESET_HUDI         = 1;
TARGETJFRQ_20MHZ         = 0;
TARGETJFRQ_10MHZ         = 1;
TARGETJFRQ_5MHZ          = 2;
```

Each script then:

Creates the object

```
dash = WScript.CreateObject ( "DashScript.Dash" );
```

Finds and configures the current target

```
function TestTargets()//gets current target and configures it
{
    var target = dash.GetCurrentTarget();
    WScript.Echo( "Current Target = " + target );
    dash.ConfigureTarget(target,
                                TARGETBOOT_RUN,
                                TARGETRESET_HALT,
                                TARGETRESET_PIN,
                                TARGETJFRQ_20MHZ );
    WScript.Echo( "Current target is:- " + dash.GetTargetInfo( target
    ) );
}
```

Loads a program file

```
function LoadProgram( filename ) //Loads a program file
{
    if (dash.LoadProgramFile(filename))
        WScript.Echo("Program file \""+filename+"\" loaded");
    else
        WScript.Echo ("Failed to load program file
\""+filename+"\"");
}
```

Sets a breakpoint

```
function TestBreakpoints() //Adds a breakpoint
{
    // Use your own test program here!
    LoadProgram("F:\\Dashscript\\sample.elf");

    dash.ClearAllBreakpointS();
    var BreakpointID1 = dash.CreateBreakpoint( BPTYPE_WATCH,
"0x0c000010" );
    dash.SetWatchBreakpointParameters( BreakpointID1,

false, "",

true,

BPACCESSSIZE_ANY,

BPACCESSTYPE_RW);
    dash.Run();
    if (dash.IsRunning())
        WScript.Echo ("TestBreakpoints failed");
    else
        WScript.Echo( "TestBreakpoints succeeded" );
}
```

Initializing for 7047

The 7047 has a single debug stub that resides in ASE RAM on the target processor. This is loaded each time the DASH resets the target.

The DashScript commands let you create and run a script to:

- Initialize the vector table.
- Initialize the target's RAM. (Optional.)

For more information see “Scripting commands” on page 390.

Initializing the 7047 vector table

On reset the DASH reads the first 1kbyte of the 7047's application flash from 0x00000000. If its contents are FFFF (new unused target board) you must write a valid vector table to flash from 0x00000000 to be used by the debug stub. Use the *Reflash* command to do this.

If the first 1kbytes of flash contain anything other than FFFF, the DASH assumes the contents are valid and does not write the vector table. The vector can be reprogrammed at any time using the *Reflash* command.

Initializing RAM with a script

To initialize the target's RAM write a script to program the control memory mapped registers directly.

Do not use the WriteRegister command in a RAM initialization script or unexpected results may occur.

Troubleshooting

DashScript will fail to run if it cannot find the DASH or target.

Check that the:

- DASH is not in use by software on another computer on the network.
- DASH serial number in the NetFlash *DA ID* box is correct.
- Network cabling between your computer, the DASH, and the target is OK.

Check on the DASH that the:

- Green *LNK* LED (ethernet link OK) is illuminated on the DASH.
- Yellow *T/R* LED (ethernet data) flashes occasionally.

If you have checked all of the above and are still experiencing problems contact Technical Support for assistance.

Scripting commands

GetTargetInfo

Syntax

```
GetTargetInfo(target)
```

Supported by

Running scripts, Script region.

Description

Returns an identifier with information about the specified target.

Parameters and Remarks

target identifies the target to find out about.

Examples

VBScript

```
dash.GetTargetInfo(target)
```

JScript

```
dash.GetTargetInfo(target);
```

ConfigureTarget

Syntax

```
ConfigureTarget(target,
               runBootRom,
               haltAfterReset,
               resetMethod,
               JTagFreq)
```

Description

Lets you do the following for the specified target:

- Run a boot ROM.
- Halt the target after it is reset.
- Set the reset method.
- Set JTAG clock frequency.

Parameters and Remarks

target identifies the target to configure.

runBootRom. Set to 0 if you do not want to run the boot ROM after a target reset, set any other value to run the boot ROM after a target reset. You cannot initialize the RAM of a 7047 with a boot ROM.

haltAfterReset. Set to 0 to continue running the target after it is reset, set any other value to halt the target after it is reset.

resetMethod, the available values are: 0 = Pin Reset, 1 = HUDI reset. All other values are reserved.

JTagFreq, the available values are: 0 = 20MHz, 1 = 10MHz, 2 = 5MHz. All other values reserved.

Examples

VBScript

```
call dash.ConfigureTarget(target,0,1,0,0)
```

JScript

```
dash.ConfigureTarget(target,0,1,0,0);
```

LoadExtendedStub

Syntax

```
LoadExtendedStub(address)
```

Description

Loads the Extended debug stub to the memory set up by a script or boot ROM.
Returns non-zero on success else zero.

Parameters and Remarks

address specifies where to load the Extended stub.

Examples

VBScript

```
dash.LoadExtendedStub(address)
```

JScript

```
dash.LoadExtendedStub(address);
```

NOTE: *You cannot load the Extended debug stub to a 7047.*

Reflash

Syntax

```
Reflash(target)
```

Description

Writes a valid vector table to flash from 0x00000000 for use by the debug stub.

Parameters and Remarks

target identifies the target to reflash.

On reset the DASH reads the first 1kbyte of the SH7047's application flash from 0x00000000. If its contents are FFFF (new unused target board) you must write a valid vector table to flash from 0x00000000 to be used by the debug stub.

If the first 1kbytes of flash contain anything other than FFFF, the DASH assumes the contents are valid and does not write the vector table. The vector can be reprogrammed at any time.

Examples

VBScript

```
dash.Reflash(target)
```

JScript

```
dash.Reflash(target);
```

NOTE: *The SH7047 has a single debug stub that resides in ASE RAM on the target processor. This is loaded each time the DASH resets the target.*

General and control

LoadProgramFile

Syntax

```
LoadProgramFile(pathname)
```

Supported by

Running scripts, Script region.

Description

Loads the specified program file and debug information. Returns 1 if a file is loaded, otherwise it returns 0.

Parameters and Remarks

pathname specifies the directory path and file to load. For example, C:\dashscript\load.j.

Examples

VBScript

```
dash.LoadProgramFile("e:\projects\maketest\hello.elf")
```

JScript

```
dash.LoadProgramFile("e:\\projects\\maketest\\hello.elf")
```

LoadProgramFileEx

Syntax

```
LoadProgramFileEx(filename, hardReset, binaryOnly)
```

Supported by

Running scripts, Script region.

Description

Loads the specified program file and debug information. Returns 1 if a file is loaded, otherwise it returns 0.

Parameters and Remarks

filename is the name of the file to load.

hardReset is a boolean which will perform a hard reset when set else none.

binaryOnly. Set to: 0 to load binary and symbols, 1 to load only binary, and 2 to load only symbols.

Examples

VBScript

```
dash.LoadProgramFileEx("e:\projects\maketest\hello.elf", false, 1)
```

JScript

```
dash.LoadProgramFileEx("e:\\projects\\maketest\\hello.elf", false, 1)
```

LoadBinaryFile

Syntax

```
LoadBinaryFile(pathname,  
               Numeric binary location)
```

Supported by

Running scripts, Script region.

Description

Loads a binary file from the specified location into target memory. Returns 1 on success, otherwise it returns 0.

Parameters and Remarks

pathname identifies the file.

Numeric binary location specifies where to load the file from.

Examples

VBScript

```
Sub LoadSomeBinary()  
    call dash.LoadBinaryFile "d:\projects\codescape\example.bin", "201392128"  
    call dash.LoadBinaryFile "d:\projects\codescape\example.bin", 201392128  
    call dash.LoadBinaryFile "d:\projects\codescape\example.bin", "0xc010000"  
    call dash.LoadBinaryFile "d:\projects\codescape\example.bin", "main"  
End Sub
```

JScript

```
function LoadSomeBinary()  
{  
    dash.LoadBinaryFile("d:\\projects\\codescape\\example.bin", "201392128");  
    dash.LoadBinaryFile("d:\\projects\\codescape\\example.bin", 201392128);  
    dash.LoadBinaryFile("d:\\projects\\codescape\\example.bin", "0xc010000");  
    dash.LoadBinaryFile("d:\\projects\\codescape\\example.bin", "main");  
}
```

HardReset

Syntax

```
HardReset ( )
```

Supported by

Running scripts, Script region.

Description

Resets the target board.

Parameters and Remarks

None.

Examples

VBScript

```
dash.HardReset ( )
```

JScript

```
dash.HardReset ( ) ;
```

SoftReset

Syntax

```
SoftReset()
```

Supported by

Running scripts, Script region.

Description

Resets the debug system.

Parameters and Remarks

None.

Examples

VBScript

```
dash.SoftReset()
```

JScript

```
dash.SoftReset();
```

Run

Syntax

`Run ()`

Supported by

Running scripts, Script region.

Description

Runs the target processor.

Parameters and Remarks

None.

Examples

VBScript

```
dash.Run ( )
```

JScript

```
dash.Run ( ) ;
```

IsRunning

Syntax

```
IsRunning()
```

Supported by

Running scripts, Script region.

Description

Queries the run state of the target processor. Returns 1 if running, 0 if not running.

Parameters and Remarks

None.

Examples

VBScript

```
Dim Running
Running = 1
Do
    Running = dash.IsRunning
Loop Until Running = 0
```

JScript

```
Run();
while(dash.IsRunning() != 0);
```

Stop

Syntax

```
StopTarget( )
```

Supported by

Running scripts, Script region.

Description

Stops the current target. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

None

Examples

VBScript

```
if dash.Stop() <> 0 then
    WScript.echo("Target has stopped")
else
    WScript.echo("Target has not stopped")
end if
```

JScript

```
if(dash.Stop())
{
    WScript.echo("Target has stopped");
}
else
{
    WScript.echo("Target has not stopped");
}
```

SymbolExists

Syntax

```
SymbolExists(string Symbol)
```

Supported by

Running scripts, Script region.

Description

Looks up the specified symbol in the symbol table. Returns 0 if the parameter specified is not a string with an alphabetic first character, or if the symbol could not be found.

Parameters and Remarks

string Symbol is the symbol to search for in the specified string.

Examples

VBScript

```
if dash.SymbolExists("main") <> 0 then
    WScript.echo("Symbol exists")
else
    WScript.echo("Symbol does not exist")
end if
```

JScript

```
if(dash.SymbolExists("main"))
{
    WScript.echo("Symbol exists");
}
else
{
    WScript.echo("Symbol does not exist");
}
```

EvaluateSymbol

Syntax

```
EvaluateSymbol(string Symbol)
```

Supported by

Running scripts, Script region.

Description

Searches for the symbol specified in the string. Returns the value of the symbol from the symbol table on success.

Parameters and Remarks

string symbol is a string that includes a symbol to search for.

The string must have an alphabetic first character. The string can be written as a whole expression.

Examples

VBScript

```
var main = dash.EvaluateSymbol("main")
```

JScript

```
var main = dash.EvaluateSymbol("main");
```

Basic read/write

ReadByte

Syntax

```
ReadByte(Numeric address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a byte (8 bits) from a specified target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    WScript.echo("Byte at main = " & dash.ReadByte("main"))  
    WScript.echo("Word at main + 4 = " & dash.ReadWord("main + 4"))  
    WScript.echo("Long at main + 8 = " & dash.ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    WScript.echo("Byte at main = " + dash.ReadByte("main"));  
    WScript.echo("Word at main + 4 = " + dash.ReadWord("main + 4"));  
    WScript.echo("Long at main + 8 = " + dash.ReadLong("main + 8"));  
}
```

ReadWord

Syntax

```
ReadWord(address)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a word (16 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    WScript.echo("Byte at main = " & dash.ReadByte("main"))  
    WScript.echo("Word at main + 4 = " & dash.ReadWord("main + 4"))  
    WScript.echo("Long at main + 8 = " & dash.ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    WScript.echo("Byte at main = " + dash.ReadByte("main"));  
    WScript.echo("Word at main + 4 = " + dash.ReadWord("main + 4"));  
    WScript.echo("Long at main + 8 = " + dash.ReadLong("main + 8"));  
}
```

ReadLong

Syntax

```
ReadLong ( address )
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a long (32 bits) from a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

Examples

VBScript

```
Sub ReadSomeMemory()  
    WScript.echo("Byte at main = " & dash.ReadByte("main"))  
    WScript.echo("Word at main + 4 = " & dash.ReadWord("main + 4"))  
    WScript.echo("Long at main + 8 = " & dash.ReadLong("main + 8"))  
End Sub
```

JScript

```
function ReadSomeMemory()  
{  
    WScript.echo("Byte at main = " + dash.ReadByte("main"));  
    WScript.echo("Word at main + 4 = " + dash.ReadWord("main + 4"));  
    WScript.echo("Long at main + 8 = " + dash.ReadLong("main + 8"));  
}
```

WriteByte

Syntax

```
WriteByte(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a byte (8 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call dash.WriteByte ("main", 255)  
    call dash.WriteWord ("main + 4", "0xabcd")  
    call dash.WriteLine ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    dash.WriteByte("main", 255);  
    dash.WriteWord("main + 4", "0xabcd");  
    dash.WriteLine("main + 8", "0xfedcba");  
}
```

WriteWord

Syntax

```
WriteWord(address,  
           value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a word (16 bits) to a target memory location.

Parameters and Remarks

address is a number or string expression that specifies a location in memory.

value is a number or string expression that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call dash.WriteByte ("main", 255)  
    call dash.WriteWord ("main + 4", "0xabcd")  
    call dash.WriteLine ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    dash.WriteByte("main", 255);  
    dash.WriteWord("main + 4", "0xabcd");  
    dash.WriteLine("main + 8", "0xfedcba");  
}
```

WriteLong

Syntax

```
WriteLong(Numeric address,  
          Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a long (32 bits) to a target memory location.

Parameters and Remarks

Numeric address is a number that specifies a location in memory.

Numeric value is a number that specifies the data to be written to.

Examples

VBScript

```
Sub WriteSomeMemory()  
    call dash.WriteByte ("main", 255)  
    call dash.WriteWord ("main + 4", "0xabcd")  
    call dash.WriteLong ("main + 8", "0xfedcba")  
End Sub
```

JScript

```
function WriteSomeMemory()  
{  
    dash.WriteByte("main", 255);  
    dash.WriteWord("main + 4", "0xabcd");  
    dash.WriteLong("main + 8", "0xfedcba");  
}
```

WriteRegister

Syntax

```
WriteRegister(Register name,  
              Numeric value)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes to a target processor register.

Parameters and Remarks

Register name identifies the register.

Numeric value is a number to assign to that register.

Examples

VBScript

```
fred=dash.WriteRegister ("fr0", 3.14159)
```

JScript

```
dash.WriteRegister("fr0", 3.14159);
```

ReadRegister

Syntax

```
ReadRegister(RegisterName)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads from a target processor register.

Parameters and Remarks

Register Name specifies the register.

Examples

VBScript

```
Sub ReadSomeRegisters()  
    WScript.echo("Value of pc = " & dash.ReadRegister("pc"))  
    WScript.echo("Value of r0 = " & dash.ReadRegister("r0"))  
End Sub
```

JScript

```
function ReadSomeRegisters()  
{  
    WScript.echo("Value of pc = " + dash.ReadRegister("pc"))  
    WScript.echo("Value of r0 = " + dash.ReadRegister("r0"))  
}
```

ReadString

Syntax

```
ReadString(address,  
            length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated ASCII string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
call dash.ReadString(address, 100)
```

JScript

```
if(dash.ReadString(address, 100) != "Hello John")  
{  
    throw "Read/Write String failed"  
}
```

ReadWString

Syntax

```
ReadWString(address,  
            length)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Reads a null terminated UNICODE string from the current target's memory.

Parameters and Remarks

address is an address in memory.

length specifies the maximum memory length to read.

Examples

VBScript

```
fred=dash.ReadWString(address, 100)
```

JScript

```
if(dash.ReadWString(address, 100) != "GoodBye")  
{  
    throw "Read/Write String failed"  
}
```

WriteString

Syntax

```
WriteString(address,  
            string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call dash.WriteString(address, "Hello John")
```

JScript

```
dash.WriteString(address, "Hello John");
```

WriteWString

Syntax

```
WriteWString(address,  
             string)
```

Supported by

DASH boot scripts, Running scripts, Script region.

Description

Writes a null terminated string (including the null terminator) as unicode to memory on the target.

Parameters and Remarks

address is a memory location.

string is the string to write.

Examples

VBScript

```
call dash.WriteWString(address, "GoodBye")
```

JScript

```
dash.WriteWString(address, "GoodBye");
```

Breakpoints

CreateBreakpoint

Syntax

```
CreateBreakpoint(type,  
                 location)
```

Supported by

Running scripts, Script region.

Description

Creates a breakpoint of the given type at the given address. Returns a breakpoint identifier on success, otherwise 0.

Parameters and Remarks

To create a Code Breakpoint set *Type* to 0.

To create a Watch Breakpoint set *Type* to 1.

location specifies where to set the breakpoint.

Examples

In the examples, the breakpoint types are represented by the following:

```
BPTYPE_CODE=0;  
BPTYPE_WATCH=1;
```

VBScript

```
Sub CreateWatchBP()  
    breakID= dash.CreateBreakpoint(BPTYPE_WATCH, "main")  
    SetWatchBreakpointParameters(breakID,  
                                true,  
                                "14",  
                                BPEXPR_C,  
                                BPACCESSSIZE_BYTE,  
                                BPACCESSTYPE_WRITE)  
End Sub
```

JScript

```
function CreateWatchBP()  
{  
    breakID= dash.CreateBreakpoint(BPTYPE_WATCH, "main");  
    SetWatchBreakpointParameters(breakID,  
                                true,  
                                "14",  
                                BPEXPR_C,  
                                BPACCESSSIZE_BYTE,  
                                BPACCESSTYPE_WRITE);  
}
```

ClearBreakpoint

Syntax

```
ClearBreakpoint( ID )
```

Supported by

Running scripts, Script region.

Description

Clears the specified breakpoint created by CreateBreakpoint on success.

Parameters and Remarks

ID is the identifier returned by CreateBreakpoint.

Examples

VBScript

```
dash.ClearBreakpoint( "ID" )
```

JScript

```
dash.ClearBreakpoint( "ID" ) ;
```

ClearAllBreakpoints

Syntax

```
ClearAllBreakpoints()
```

Supported by

Running scripts, Script region.

Description

Removes all breakpoints.

Parameters and Remarks

None.

Examples

VBScript

```
dash.ClearAllBreakpoints()
```

JScript

```
dash.ClearAllBreakpoints();
```

EnableBreakpoint

Syntax

```
EnableBreakpoint(identifier,  
                 boolean enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables a breakpoint. Returns 1 on successfully enabling or disabling the breakpoint, otherwise it returns 0.

Parameters and Remarks

identifier specifies the breakpoint.

Set the *boolean enable* value to:

- 1 to enable the specified breakpoint.
- 0 to disable the specified breakpoint.

Examples

VBScript

```
call dash.EnableBreakpoint (breakID, true)
```

JScript

```
dash.EnableBreakpoint(breakID, true);
```

EnableAllBreakpoints

Syntax

```
EnableAllBreakpoints(enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables all breakpoints. Returns 1 on successfully enabling or disabling the breakpoint, otherwise it returns 0.

Parameters and Remarks

Set *enable* to:

- 1 to enable all breakpoints.
- 0 to disable all breakpoints.

Examples

VBScript

```
call dash.EnableBreakpoints (1)
```

JScript

```
dash.EnableBreakpoints(1);
```

SetBreakpointAction

Syntax

```
SetBreakpointAction(identifier,  
                    numeric action,  
                    boolean enable)
```

Supported by

Running scripts, Script region.

Description

Enables or disables specific actions when a breakpoint has been hit. The allowed actions are:

- Stop the target.
- Remove the breakpoint.
- Display a message box.
- Beep.

Returns 1 on success, otherwise it returns 0.

NOTE: *You must call SetBreakpointAction separately for each action you require. The examples demonstrate how to do this.*

Parameters and Remarks

identifier specifies the breakpoint.

Set the *boolean enable* value to:

- 1 to enable the specified breakpoint action.
- 0 to disable the specified breakpoint action.

Set the *numeric action* value to:

- 0 to stop the target when the breakpoint when it is hit.
- 1 to remove the breakpoint after it has been hit.
- 2 to display a message box prompt when the breakpoint has been hit.
- 3 to beep when the breakpoint has been hit.

Examples

VBScript

```
call dash.SetBreakpointAction (breakID, BPACTION_HALT, true)
    call dash.SetBreakpointAction (breakID, BPACTION_ONESHOT, false)
    call dash.SetBreakpointAction (breakID, BPACTION_PROMPT, false)
    call dash.SetBreakpointAction (breakID, BPACTION_BEEP, true)
```

JScript

```
dash.SetBreakpointAction(breakID, BPACTION_HALT, true);
    dash.SetBreakpointAction(breakID, BPACTION_ONESHOT, false);
    dash.SetBreakpointAction(breakID, BPACTION_PROMPT, false);
    dash.SetBreakpointAction(breakID, BPACTION_BEEP, true);
```

SetBreakpointCondition

Syntax

```
SetBreakpointCondition(identifier,  
                        string expression,  
                        numeric expression type,  
                        numeric trigger count,  
                        boolean incOn,  
                        boolean breakWhen)
```

Supported by

Running scripts, Script region.

Description

Sets a conditional expression for a breakpoint. Returns 1 on success, otherwise it returns 0.

Parameters and Remarks

identifier specifies the breakpoint.

string expression is a string representing the condition.

Set *numeric expression type* to 0 for C/C++, or to non-zero for assembly.

numeric trigger count is the number of hits before breakpoint actions are performed.

incOnTrue derements the trigger count only when conditions are true.

incOnFalse always decrements the trigger count.

Set *boolean breakWhen* as false to break either when the trigger count reaches 0 or when condition is true. Set *breakWhen* as true to break when trigger count reaches zero and the condition is true.

Examples

VBScript

```
call dash.setBreakpointCondition (breakID,  
                                "index == 375",  
                                BPEXPR_C,  
                                37,  
                                true,  
                                true)
```

JScript

```
dash.setBreakpointCondition(breakID,  
                            "index == 375",  
                            BPEXPR_C,  
                            37,  
                            true,  
                            true);
```

SetWatchBreakpointParameters

Syntax

```
BOOL SetWatchBreakpointParameters(Identifier,  
                                   boolean incDataCondition,  
                                   string dataCondition,  
                                   numeric expressionType,  
                                   numeric accessSize,  
                                   numeric accessType)
```

Supported by

Running scripts, Script region.

Description

Sets the parameters for a watch breakpoint.

Parameters and Remarks

identifier specifies the breakpoint.

boolean incDataCondition is set to true to include a data condition otherwise false.

string dataCondition is a numeric expression that sets the data condition.

numeric expressionType is a numeric expression that sets the expression type. To set the expression syntax to:

- C/C++, enter the value 0.
- Assembly, enter the value 1.

numeric accessSize is a numeric expression that sets the size. To set the size to:

- Any, enter the value 0.
- Byte, enter the value 1.
- Word, enter the value 2.
- Long, enter the value 4.
- Quad, enter the value 8.

numeric accessType is a numeric expression that sets the type. To set the type to:

- Read, enter the value 1.
- Write, enter the value 2.

- Read, or Write enter the value 3.

Examples

VBScript

```
call dash.SetWatchBreakpointParameters (breakID,  
                                         true,  
                                         "14",  
                                         BPEXPR_C,  
                                         BPACCESSSIZE_BYTE,  
                                         BPACCESSTYPE_WRITE)
```

JScript

```
dash.SetWatchBreakpointParameters(breakID,  
                                   true,  
                                   "14",  
                                   BPEXPR_C,  
                                   BPACCESSSIZE_BYTE,  
                                   BPACCESSTYPE_WRITE);
```

SetBreakpointDataMask

Syntax

```
SetBreakpointDataMask(identifier,  
                        data mask)
```

Supported by

Running scripts, Script region.

Description

Sets the data mask for a watch breakpoint.

Parameters and Remarks

identifier specifies the breakpoint.

data mask must be a 32-bit number.

Examples

Each example is to mask 10 bits.

VBScript

```
call dash.SetBreakpointDataMask (breakID,  
                                0x000003FF)
```

JScript

```
dash.SetBreakpointDataMask(breakID,  
                            0x000003FF);
```

SetBreakpointLocationMask

Syntax

```
SetBreakpointLocationMask(breakID,  
                           maskSelect)
```

Supported by

Running scripts, Script region.

Description

Masks a location for the breakpoint. The identifier specifies the breakpoint, and mask. Select is a value that specifies the bits to mask.

To mask:

- No bits, enter the value 1.
- The lower 10 bits, enter the value 2.
- The lower 12 bits, enter the value 3.
- The lower 16 bits, enter the value 4.
- The lower 20 bits, enter the value 5.
- All bits, enter the value 6.

Returns true 1 on success, otherwise it returns 0.

NOTE: *This command is for SH4 targets.*

Parameters and Remarks

breakID is the identifier that specifies the breakpoint.

maskSelect specifies the bits to mask.

Examples

VBScript

```
call dash.SetBreakpointLocationMask (breakID,  
                                     BPLOCMASK_LOW10)
```

JScript

```
dash.SetBreakpointLocationMask(breakID,  
                               BPLOCMASK_LOW10);
```

SetBreakpointLocationMaskEx

Syntax

```
SetExtendedBreakpointLocationMask(breakID,  
                                   mask)
```

Supported by

Running scripts, Script region.

Description

Masks a location for the breakpoint.

Returns true 1 on success, otherwise it returns 0.

NOTE: *This command is for SH2 targets allowing full 32-bit masking.*

Parameters and Remarks

breakID is the identifier that specifies the breakpoint.

maskSelect specifies the bits to mask.

Examples

VBScript

```
call dash.SetExtendedBreakpointLocationMask(breakID,  
                                              BPLOCMASK_LOW10)
```

JScript

```
dash.SetExtendedBreakpointLocationMask(breakID,  
                                         BPLOCMASK_LOW10);
```

Multiple Target Support

GetCurrentTarget

Syntax

```
GetCurrentTarget()
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the current target.

Parameters and Remarks

The strings returned from `GetCurrentTarget` should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = dash.GetCurrentTarget()
```

JScript

```
target = dash.GetCurrentTarget();
```

GetFirstTarget

Syntax

```
GetFirstTarget()
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the first target.

Parameters and Remarks

The strings returned from `GetFirstTarget` should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = dash.GetFirstTarget()
```

JScript

```
target = dash.GetFirstTarget();
```

GetNextTarget

Syntax

```
GetNextTarget ( current )
```

Supported by

Running scripts, Script region.

Description

Returns an identifier for the next target. The return string is empty if all targets have been identified.

Parameters and Remarks

current identifies the current target.

The strings returned from GetNextTarget should be treated as magic cookies and the format is subject to change.

Examples

VBScript

```
target = dash.GetNextTarget (current)
```

JScript

```
target = dash.GetNextTarget (current);
```

SelectTarget

Syntax

```
SelectTarget ( target )
```

Supported by

Running scripts, Script region.

Description

Selects the chosen target as the current target. Returns true if successfully selected, otherwise false.

Parameters and Remarks

target identifies the target to use.

Examples

VBScript

```
dash.SelectTarget ( target )
```

JScript

```
dash.SelectTarget ( target ) ;
```

CopyData

Syntax

```
CopyData(target,  
         address,  
         size,  
         numElems,  
         destTarget,  
         destAddress)
```

Supported by

Running scripts, Script region.

Description

Copies data from one target to another.

Parameters and Remarks

target specifies the target to copy from.

address specifies the address to copy from.

size specifies the size of data (1, 2, 4, and 8 are valid sizes).

numElems specifies the number of elements of the given size to copy.

destTarget specifies the target to copy to.

destAddress specifies the address to copy to.

The maximum size allowed in one transfer is 65536, so $\text{size} * \text{numElems}$ must be less than 65536.

The command lets you transfer data from a big endian target to a little endian target (and vice versa) correctly. This command should not be used to transfer large amounts of data.

Examples

VBScript

```
call dash.CopyData(target, address, size, numElems, destTarget, destAddress)
```

JScript

```
dash.CopyData(target, address, size, numElems, destTarget, destAddress);
```

RunTarget

Syntax

```
RunTarget(target)
```

Supported by

Running scripts, Script region.

Description

Runs the target specified. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

target identifies the target to run.

Examples

VBScript

```
if dash.RunTarget(target) <> 0 then
    WScript.echo("Target is running")
else
    WScript.echo("Target is not running")
end if
```

JScript

```
if(dash.RunTarget(target))
{
    WScript.echo("Target is running");
}
else
{
    WScript.echo("Target is not running");
}
```

StopTarget

Syntax

```
StopTarget(target)
```

Supported by

Running scripts, Script region.

Description

Stops the target specified. Returns non-zero if the target was not running previously, returns zero if the target was running previously.

Parameters and Remarks

target identifies the target to stop.

Examples

VBScript

```
if dash.StopTarget(target) <> 0 then
    WScript.echo("Target has stopped")
else
    WScript.echo("Target has not stopped")
end if
```

JScript

```
if(dash.StopTarget(target))
{
    WScript.echo("Target has stopped");
}
else
{
    WScript.echo("Target has not stopped");
}
```

RunAllTargets

Syntax

```
RunAllTargets( )
```

Supported by

Running scripts, Script region.

Description

Runs all targets that are connected. Returns the number of targets successfully started.

Parameters and Remarks

None.

Examples

VBScript

```
dash.RunAllTargets( )
```

JScript

```
dash.RunAllTargets( ) ;
```

StopAllTargets

Syntax

```
StopAllTargets()
```

Supported by

Running scripts, Script region.

Description

Stops all the connected targets. Returns the number of targets successfully stopped.

Parameters and Remarks

None.

Examples

VBScript

```
dash.StopAllTargets()
```

JScript

```
dash.StopAllTargets();
```

IsTargetRunning

Syntax

```
IsTargetRunning(target)
```

Supported by

Running scripts, Script region.

Description

Queries the run status of a specific target. Returns true if the target is running, otherwise false.

Parameters and Remarks

target identifies the target to test.

Examples

VBScript

```
dash.IsTargetRunning(target)
```

JScript

```
WScript.echo("Target is " + (dash.IsTargetRunning(target) ? " " : "not ") +  
"running");
```

Channel Support

The Channel commands are all for communicating via CPDIAL's communication channels.

ChannelReserve

Syntax

```
ChannelReserve(channel id)
```

Supported by

Running scripts, Script region.

Description

Reserves a specified channel for exclusive use. Returns non-zero on success else zero.

To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Serial spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id a number specifying the reserved channel.

Examples

VBScript

```
function WriteToChannel(channel, a)
    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            text = ("Message " & a)
            for j = 1 to Len (text) step 1
                Dim MGSChar
                MGSChar = Mid(text , j , 1)
                if dash.ChannelWrite (channel, Char) Then
                    WScript.EchoWrite("write to channel, success")
                Else
                    WScript.EchoWrite("Failed to write to channel")
                End If
            Next
        End If
    End If
    dash.ChannelRelease(channel)
End Function
```

JScript

```
function WriteToChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            text = "Message " + a;
            for(j = 0; j < text.length; ++j)
            {
                if(!dash.ChannelWrite(channel, text.charCodeAt(j)))
                {
                    WScript.Echo("Failed to write to channel");
                }
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

ChannelRelease

Syntax

```
ChannelRelease(channel id)
```

Supported by

Running scripts, Script region.

Description

Releases a previously reserved channel. Returns non-zero on success else zero.

To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id is number specifying the reserved channel specified to release.

Examples

VBScript

```
function WriteToChannel(channel, a)
    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            text = ("Message " & a)
            for j = 1 to Len (text) step 1
                Dim MGSChar
                MGSChar = Mid(text , j , 1)
                if dash.ChannelWrite (channel, Char) Then
                    WScript.Echo("write to channel, success")
                Else
                    WScript.EchoWrite("Failed to write to channel")
                End If
            Next
        End If
    End If
    dash.ChannelRelease(channel)
End Function
```

JScript

```
function WriteToChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            text = "Message " + a;
            for(j = 0; j < text.length; ++j)
            {
                if(!dash.ChannelWrite(channel, text.charCodeAt(j)))
                {
                    WScript.EchoWrite("Failed to write to channel");
                }
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

ChannelValidate

Syntax

```
ChannelValidate(channel id)
```

Supported by

Running scripts, Script region.

Description

Checks that the specified channel exists on the current target and is reserved by this process. Returns non-zero on success else zero.

To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id is a number specifying the channel to validate.

Examples

VBScript

```
function WriteToChannel(channel, a)
    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            text = ("Message " & a)
            for j = 1 to Len (text) step 1
                Dim MGSChar
                MGSChar = Mid(text , j , 1)
                if dash.ChannelWrite (channel, Char) Then
                    WScript.Echo("write to channel, success")
                Else
                    WScript.Echo("Failed to write to channel")
                End If
            Next
        End If
    End If
    dash.ChannelRelease(channel)
End Function
```

JScript

```
function WriteToChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            text = "Message " + a;
            for(j = 0; j < text.length; ++j)
            {
                if(!dash.ChannelWrite(channel, text.charCodeAt(j)))
                {
                    WScript.Echo("Failed to write to channel");
                }
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

ChannelDataReady

Syntax

```
ChannelDataReady(channel id)
```

Supported by

Running scripts, Script region.

Description

Checks that there is data to read on the specified channel. Returns non zero if data is available else zero.

To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id is a number specifying the channel to check for data.

Examples

VBScript

```
function ReadFromChannel(channel, a)

    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            msg = ""
            numRead = 0
            while(dash.ChannelDataReady(channel))
                ch = dash.ChannelRead(channel)
                temp = String.fromCharCode(ch)
                msg = msg + temp
                numRead = numRead + 1
            Wend
            WScript.Echo(dash.ChannelDataReady(channel))
            if(numRead) Then
                WScript.Echo(msg)
            Else
                WScript.Echo("msg ")
            End If
            dash.ChannelRelease(channel)
        End If
    End If
End Function
```

JScript

```
function ReadFromChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            msg = "";
            numRead = 0;
            while(dash.ChannelDataReady(channel))
            {
                var ch = dash.ChannelRead(channel);
                temp = String.fromCharCode(ch);
                msg += temp;
                ++numRead;
            }
            if(numRead)
            {
                WScript.Echo(msg);
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

ChannelRead

Syntax

```
ChannelRead(channel id)
```

Supported by

Running scripts, Script region.

Description

Reads data from the specified channel. To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id is a number specifying the channel to read data from.

Examples

VBScript

```
function ReadFromChannel(channel, a)

    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            msg = ""
            numRead = 0
            while(dash.ChannelDataReady(channel))
                ch = dash.ChannelRead(channel)
                temp = String.fromCharCode(ch)
                msg = msg + temp
                numRead = numRead + 1
            Wend
            WScript.Echo(dash.ChannelDataReady(channel))
            if(numRead) Then
                WScript.Echo(msg)
            Else
                WScript.Echo("msg ")
            End If
            dash.ChannelRelease(channel)
        End If
    End If
End Function
```

JScript

```
function ReadFromChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            msg = "";
            numRead = 0;
            while(dash.ChannelDataReady(channel))
            {
                var ch = dash.ChannelRead(channel);
                temp = String.fromCharCode(ch);
                msg += temp;
                ++numRead;
            }
            if(numRead)
            {
                WScript.Echo(msg);
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

ChannelWrite

Syntax

```
ChannelWrite(channel id,  
             character)
```

Supported by

Running scripts, Script region.

Description

Writes the specified character to the specified channel.

To use the:

- Operating System debugging interface enter 0. (Currently this channel is spare.)
- File Server enter 2. (Debugging File Server Protocol.)
- Sound spare enter 3.
- Spare enter 4.

For more information refer to the Debug Interface guide.

Parameters and Remarks

channel id is a number specifying the channel to write data to.

character is the character to write to the channel.

Examples

VBScript

```
function WriteToChannel(channel, a)
    if(dash.ChannelReserve(channel)) Then
        if(dash.ChannelValidate(channel)) Then
            text = ("Message " & a)
            for j = 1 to Len (text) step 1
                Dim MGSChar
                MGSChar = Mid(text , j , 1)
                if dash.ChannelWrite (channel, Char) Then
                    WScript.Echo("write to channel, success")
                Else
                    WScript.Echo("Failed to write to channel")
                End If
            Next
        End If
    End If
    dash.ChannelRelease(channel)
End Function
```

JScript

```
function WriteToChannel(channel, a)
{
    if(dash.ChannelReserve(channel))
    {
        if(dash.ChannelValidate(channel))
        {
            text = "Message " + a;
            for(j = 0; j < text.length; ++j)
            {
                if(!dash.ChannelWrite(channel, text.charCodeAt(j)))
                {
                    WScript.Echo("Failed to write to channel");
                }
            }
        }
        dash.ChannelRelease(channel);
    }
}
```

Hardware Reference

SH2 Hardware Reference

ASE and Extended debug stubs (SH7615, SH7622)

The debug stub has two parts, the ASE debug stub and the Extended debug stub. Two stubs are required because ASE memory is limited to 1kbyte and is insufficient to provide all the required functionality. The Extended debug stub, due to its size, must be resident in external RAM.

The two stubs and their environment requirements are discussed in this chapter.

The ASE debug stub

The ASE debug stub resides in the SH2's 1kbyte of ASE memory. This memory can only be accessed in ASE mode by the debug tools, it cannot be physically read or written to by the application code. The ASE debug stub is a minimal debug stub loaded into ASE RAM after a target system reset. The ASE debug stub facilitates basic debugging, system initialization and the loading of the larger, functionally complete, Extended debug stub. Once the Extended debug stub is loaded the ASE debug stub is used simply to chain ASE exceptions to handlers within the Extended debug stub. The ASE debug stub and ASE RAM are transparent to the application.

The Extended debug stub

The Extended debug stub resides in RAM and requires 16kbytes of memory. The Extended debug stub provides comprehensive debugging of the target. The debug stub's memory is not protected from corruption by the application code, thus the application must not attempt to use the memory space allotted to the Extended debug stub. An errant piece of application code can write over the Extended debug stub causing it to fail. Under these circumstances the debug stub (and debug session in progress) must be reloaded and restarted. The Extended debug stub has its own private stack within the boundaries of the allotted 16kbytes of RAM and thus does not require use of the application program's stack.

The location of the Extended debug stub is defined either by the boot ROM code, or by the settings in the DA Start-up dialog in CodeScape.

The Extended debug stub always has the same three instructions at the start of the 16kbyte section, BRK, RTE, NOP with the instruction codes 0x00, 0x000B, and 0x0009.

Byte sequence:

- 0x00, 0x00, 0x00, 0x0B, 0x00, 0x09 - big endian Extended debug stub.

Functional differences of the ASE and Extended debug stubs

The table below lists the differences in functionality of the two stubs. **Optimization of the stubs**

The ASE debug stub must fit entirely into 1kbyte (512 instructions), for this reason all of the ASE debug stub's functions are optimized for size. In some cases functionality is delegated from the ASE debug stub to the DASH.

An example of this is the 'Read Context UBC' command. This command is issued by CodeScape to request that the values of all of the UBC's (user break controller) memory mapped registers are returned.

In the Extended debug stub, a protocol exists for this purpose. The DASH requests the UBC's context and the Extended debug stub replies by reading the required registers and returning them in a single transaction.

In the ASE debug stub no such protocol exists between the DASH and the ASE debug stub. However, the same result is achieved by the DASH issuing a series of 'Read Memory' commands to read each of the individual registers one by one.

The DASH and Extended debug stub performs the UBC context read in the most efficient (fastest) method possible, whereas the DASH and the ASE debug stub performs the UBC context read using the minimum amount of code in the ASE debug stub. The net result is that the Extended debug stub runs much faster than the ASE debug stub.

Exception handling in the ASE debug stub

Exception handling in the Extended debug stub

The Extended debug stub handles exceptions and interrupts by installing a default exception handler which sets up the VBR register accordingly. Alternatively, you can use your own application exception handler, see *“Interrupts and Exceptions (SH7615, SH7622)”* on page 464.

Interrupts and Exceptions

(SH7615, SH7622)

This section deals with issues concerning exception and interrupt handling on the target and the interaction and interrupt requirements of the debug stub.

Exception handling with or without a boot ROM

Once the Extended debug stub has loaded, the DASH instructs the debug stub to either resume execution of the boot ROM (if one is installed), or set up a default environment and wait for the user to perform some action such as downloading a program. This is controlled by the option: *Halt after stub load* on the *DA Start-up options* dialog in CodeScape.

The debug stub implements exceptions using different methods depending on this condition.

Resume (debugging with a boot ROM)

This mode exists to allow program execution to resume back to the boot ROM after the Extended debug stub is loaded. This is achieved with the minimum of disruption to the state of the target microprocessor context.

Halt (debugging without a boot ROM)

This mode exists to allow the Extended debug stub to be loaded and then the context to be set up with default settings to allow a debugging session to commence. In this mode CodeScape does not pass program control back to the boot ROM, instead, control remains within the debug stub monitor.

- The VBR register is loaded with the default debug stub exception handler.
- The stack pointer is loaded

Exceptions during debugging with a boot ROM

When *Resume after stub load* mode is selected on the DA Start-up dialog in CodeScape, the debug stub does not implement an exception or interrupt handler of its own and it does not alter or set up the VBR register. Thus the debug stub makes no demands of the application code and does not introduce any time penalties by running debug stub exception handler code. However, this does mean that the application code must implement its own handler if exceptions or interrupts are to be used by the application code itself.

For the debug stub to process exceptions (not interrupts) in this mode, it incorporates a passback facility to allow unhandled exceptions to be caught and processed by the debug stub. To implement the passback facility the application code exception handler must call the debug stub using the sequence of instructions below.

For exceptions the passback call takes the form:

```
BRK
RTE
RTE
NOP
```

This special sequence causes the debug stub to be entered via the 'BRK' instruction and the double 'RTE' immediately after the break indicates to the debug stub that this is an exception passback call.

This sequence should be added to the end of the application code exception handler to allow the debug stub to process any unhandled exceptions. The 'NOP' at the end is not strictly required but can be added to inhibit compiler or assembler errors. You should use the passback sequence where you would normally put code to deal with unhandled exceptions. Where an exception is handled correctly by the application code, an 'RTE' must be executed rather than the passback sequence.

Notes:

1. *After the debug stub processes an unhandled exception, program execution is returned to the interrupted code, control is not returned to the application exception handler so no attempt is made to execute the 'RTE RTE' element of the passback sequence.*
2. *Some assemblers and compilers may not allow an RTE opcode following an RTE opcode. In this event the following sequence could be used:*

```
BRK
RTE
DC .W      0x002b
NOP
```

3. *Unhandled exceptions passed back to the debug stub are reported to CodeScape and this in turn is interpreted and reported as specified by CodeScape.*

Exceptions during debugging without a boot ROM

When “Halt after stub load” mode is selected on the DA Start-up dialog in CodeScape, the debug stub implements a default exception and interrupt handler by initializing the VBR register to point at the debug stub default handler.

The default handler allows all exceptions and interrupts to be caught by the debug stub and in turn reported by CodeScape. This allows application code to be debugged without the need for an application code handler to catch exceptions such as 'address error'.

The application code can at any time install its own handler by changing the contents of the VBR. Once this is done the debug stub default handler will no longer function but application code can utilise the passback method as described in the previous section to allow unhandled exceptions to be handled by the debug stub.

If the VBR is changed by the application code, the debug stub will make no attempt to restore the VBR to point at the debug stub default handler. However, the application code can restore the VBR to the default value to make the default handler functional again.

Interrupts during debugging

An additional specific passback sequence is available to allow unhandled interrupts (not exceptions) to be captured and reported to CodeScape. To implement this passback facility the application code interrupt handler must call the debug stub using the following sequence of instructions.

```
BRK
RTE
RTS
NOP
```

The above sequence should be implemented in a similar manner to the exception passback sequence described above.

Caveats and limitations

Use of the BC registers

CodeScape makes use of the BC unit to implement hardware breakpoints, for this reason the application code must not attempt to access the BC's registers.

Use of the BSC register

The SH7615 moves the address of the BSC registers from 0xFFFFFE0 to 0xFFFFFDE0 When in ASE Break Mode. Accessing the BSC register using the DASH (eg. from a script) must use the 0xFFFFFDE0 address.

Use of the FMR register

It is not possible to modify the FMR register in ASE Break Mode.

Debug stub and exception handling (SH7047)

The SH7047 debug stub and default vector table

The SH7047's debug stub is loaded into the 2kbytes of ASE RAM after a target system reset. This memory can only be accessed in ASE mode by the debug tools, it cannot be physically read or written to by the application code. The debug stub does not require use of the application program's stack.

The debug stub handles exceptions and interrupts by installing a default exception handler at 0x0000 in the first 1kbyte of flash. Alternatively, you can use your own application exception handler.

The vector table passes exceptions via software breaks at 0x30e - 0x3f4 as follows (0x30e - 0x3f4 is a reserved area at the end of the first 1kbytes of flash):

```
; 0x3e0->0x3f5 BRKs for passing exceptions up via software breaks
org 0x3e0
_exp_manual_reset
dc.w 0x0000 ; BRK
_exp_general_illegal
dc.w 0x0000 ; BRK
_exp_slot_illegal
dc.w 0x0000 ; BRK
_exp_cpu_addr_err
dc.w 0x0000 ; BRK
_exp_dma_addr_err
dc.w 0x0000 ; BRK
_exp_nmi
dc.w 0x0000 ; BRK
_exp_ubc
dc.w 0x0000 ; BRK
_exp_udi
dc.w 0x0000 ; BRK
_exp_trap32
```

```
dc.w 0x0000 ; BRK
_exp_trap33
dc.w 0x0000 ; BRK
_catch_default_exception
dc.w 0x0000 ; BRK

; A safe place to branch to 0x3fa
org 0x3f6
nop
nop
_safe_code
bra _safe_code ; 0x3fa
nop
nop
```

|

SH2e Hardware Reference

Equipment covered and terminology

This document applies to the Hitachi EVB7055F development board. Further information can be found in the EVB7055F Low-cost Evaluation Board User Manual and the Hitachi specification for the SH7055F.

- The EVB7055F low-cost evaluation board is referred to as the EVB.
- The standard production SH7055F processor is referred to as the SH7055F.
- The DASH2e debug adapter is referred to as the DASH.

Although this document refers specifically to the EVB board, the DASH can be used with any target board based around the SH7055F if provision is made for:

- connection via the supplied DASH/7055 Adapter Board
- facility to set the mode pins

Full connection details are given in *Connection details for the SH2e EVB (SH7055) on page 63.*

The EVB7055F evaluation board (EVB)

SH7055F memory map



Operating Modes

Boot mode

The SH7055F incorporates boot mode. This allows its flash memory to be programmed via the SCI1 serial port using some hidden code on the silicon provided by Hitachi for this purpose. The DASH connects to SCI1 for flash programming.

Normal debugging mode

SH7055F debugging is normally performed with your application resident in flash memory from addresses 0x00000000 to 0x0007BFFF (the debug stub resides from 0x0007C000 to 0x0007FFFF). Codescape handles all the programming issues so that you can simply load your application into flash memory as if it is RAM. You can overlay the vector table with any required vectors as long as the vectors reserved by the debug stub are retained. See *Reserved vectors on page 479* for details of which vectors you can safely overwrite and how to modify the vector table.

- To load your application into Application Flash use the Load Program File... or Load Binary... command in CodeScape (File menu > Load Program File or Load Binary).

Debugging on the SH7055F is limited because software breakpoints cannot be used in flash memory and only one hardware breakpoint is available on the SH7055F.

Your application can be loaded and run in RAM from the locations 0xFFFF6000 to 0xFFFFDE7F where software breakpoints can be used. However, with a under 32kbytes of RAM available, only small applications that will fit in this space can be debugged.

Codescape software breakpoints

Software breakpoints are implemented on both targets using a TRAPA #32 instruction.

Codescape provides a button (Tools menu > Options > Debug tab) to select between the TRAPA #32 and the BRK instruction. The BRK instruction provides software breakpoints with the following features that differ from the TRAPA #32.

- The BRK does not require a valid VBR pointer.
- The BRK does not use any user stack when in use and therefore does not need a valid user stack pointer.

NOTE: *If the special BRK instruction is used, all other vectors still require a valid VBR and valid user stack pointer, for example the address error vector.*

Final application run mode (debug available)

This facility provides an additional level of confidence since your application is executed as a result of vectoring from the final PC and SP, but with the ability to suspend and debug your application from CodeScape in the normal way.

The debug stub reserves use of particular vectors in the vector table for full operation (such as the H-UDI vector) but does not exclusively require the PC and SP vectors for the debug stub to function correctly. See *Reserved vectors on page 479* for details of which vectors you can safely overwrite and how to modify the vector table.

To run the processor in in final application run mode with debug available:

1. Prepare a version of your application code that overwrites the debug stub's default PC and SP vectors with your own reset PC and SP (PC = 0x00000000, SP = 0x00000004).
2. Load the final code into Application Flash (SH7055F) or Emulation RAM (SH7055MCM) using the Load Program File... or Load Binary... command in CodeScape (File menu > Load Program File or Load Binary).
3. Uncheck *Halt After Target Reset* on the DA Start-up tab and click OK.
The processor loads and runs your application making use of the PC and SP from the vector table.

NOTE: *During normal debugging using the debug stub's default PC and SP, Halt After Target Reset should always be checked.*

Stand-alone run mode (no debug facility)

When debugging, Application Flash on the SH7055F contains the debug stub and some reserved vectors. To test your application in stand alone run mode you should prepare a final version of your application with any debug components removed. The basic procedure is as follows:

1. Prepare a final version of your code that fully specifies the vector table with vectors pointing to your application code.

2. Program the final code into Application Flash using the Load Program File... or Load Binary... command in CodeScape (File menu > Load Program File or Load Binary).
3. Do one of the following:
 - Power off the EVB, remove the 'D' connector from the DASH/7055 Adapter Board and power on the EVB.
 - OR-
 - Uncheck the *Debug Support Enabled* check box in CodeScape and click OK (Tools menu > Target/Communication > Configure ... > DA Start-up tab).

The SH7055F resets and runs your application without intervention by the debug tools.

To return to normal debugging reconnect the 'D' connector or check the *Debug Support Enabled* check box in CodeScape and click OK. You must also reprogram the debug stub, see *Programming the debug stub for the SH7055F on page 483*.

Operating mode selection

SH7055F operating modes are set by pins MD0, MD1, MD2 and FWE as shown below. The mode setting pins should not be changed when the processor is running.

SH7055F Operating Modes

Operating Mode	FWE	MD2	MD1	MD0	Hitachi Mode Name	On-chip ROM	Area 0 Bus Width
Mode 0	0	1	0	0	MCU Expanded Mode	Disabled	8 bits
Mode 1	0	1	0	1			16 bits
Mode 2	0	1	1	0		Enabled	Set by BCR1
Mode 3	0	1	1	1	MCU single-chip mode	Enabled	
Mode 4	1	1	0	0	Boot mode	Enabled	Set by BCR1
Mode 5	1	1	0	1			
Mode 6	1	1	1	0	User program mode	Enabled	Set by BCR1
Mode 7	1	1	1	1			

- FWE is linked to VCC via the 'D' connector from the DASH and is logic 1 when the DASH cable is inserted into the DASH/7055 Adapter Board and logic 0 when disconnected.
- MD0 is set to logic 0 on the EVB by jumper J2.
- MD1 is set to logic 1 (soft pull-up) by jumper J1 when the DASH is not connected to the EVB. When the DASH is connected to the EVB, control of MD1 is surrendered to the MDY line from DASH via J17.
- Pin MD2 is fixed at logic 1 on the EVB.

With MD2 and MD0 fixed, this gives the possible mode pin configurations as shown by the shaded areas in the table above. These operating modes are utilized by the CodeScape debugging tools and are selected under control of the DASH and the 'D' connector. Further details are given in *Connection details for the SH2e EVB (SH7055) on page 63*.

Reserved vectors

The SH7055F reserves various vectors for use with the debug stub. They are listed in below.

These vectors are set up when the debug stub is programmed into flash. All other vectors are configured so that the debug stub handles them and subsequently reports via CodeScape that an Unhandled Exception has occurred. This is to make sure that exceptions occurring without a user handler are reported in CodeScape.

You can modify the reserved vectors to be used by your application, however the functionality of the debug tools will be compromised to a greater or lesser amount depending on which of the reserved vectors are modified. In general you should avoid using reserved vectors.

Vector Number	Description	Usage
0	Power-on reset	Jumps to a simple spin loop (dummy user application).
2	Manual reset	To report that a manual reset has occurred.
4	General illegal instruction	To report that an illegal instruction has occurred.
6	Slot illegal instruction	To report that a slot illegal instruction has occurred.
7	BRK instruction	Used for software breakpoints.
9	CPU address error	To report that a CPU address error has occurred.
10	DMA address error	To report that a DMA error has occurred.
11	NMI	To report that a NMI has occurred.
12	UBC	Used for hardware breakpoints.
13	FPU error	To report that a FPU error has occurred.
14	UDI interrupt	Used to force entry into the stub and for statistical profiling.
32	TRAPA #32 exception	Used for software breakpoints
33	TRAPA #33 exception	Used for BIOS calls.

The power-on reset PC and SP vectors

The power-on reset vector and power-on SP are used to set up a default SP (r15) and to jump to a spin loop which is a dummy user application. The debug stub requires that a user application is running so that it can interrupt it (the debug stub acts as a high priority interrupt handler). You can change the power-on reset vector to point to your own application code to be run on a reset. You must make sure that a valid SP is set up and that the interrupt mask is not set to level 15. The debug stub requires space on the user stack and an interrupt mask less than 15 if hardware breakpoints are being used. It is envisaged that in the early stages of debugging you would simply load code into a section of memory, set the PC to the beginning of the code and start debugging.

Modifying the vector table

You can modify individual vectors to suite your application.

The example below specifies a start PC and SP (vectors 0 and 1) and two IRQ vectors (vectors 64 and 65).

```
; Vector table start at 0x00000000.
    org 0x00000000
    dc.l my_app_start ;Application start PC.
    dc.l 0xfffffe000  ;Application initial SP.

; Let the debug stub handle intervening vectors such
; as H-UDU, Address error, etc.

; Application uses IRQ0 and IRQ1.
    org 0x00000100
    dc.l my_IRQ0_handler;Application IRQ0
    dc.l my_IRQ1_handler;Application IRQ1
```

When an application containing the above example code is loaded using Codescape, only vectors 1, 2, 64 and 65 are modified (2 to 63, 65 to 255 are unchanged). This is achieved by the DASH's flash programming management system which modifies vectors without corrupting previously programmed vectors or vectors used by the debug stub, *see Flash programming management system on page 486*.

The final application code image should handle all the vectors in the vector table.

The 7055F debug stub

The target relies on a debug stub being resident in memory for the debug tools to work.

DASH firmware target detection

After power-up or a reset the DASH performs a simple test to detect if the debug stub is present on the connected target. If the stub is not present the DASH notifies Codescape which displays the message:

- *Cannot find stub, do you want to load the stub now?*

At this point the debug stub can be reprogrammed into the target's flash memory via CodeScape.

The SH7055F debug stub

Since the SH7055F has a limited amount of RAM and a significant amount of flash memory, the debug stub is located in flash. The location of the debug stub is at 0x0007C000 and requires 16kbytes of memory (0x0007C000 to 0x0007FFFF). This area of flash memory is no longer available for your application when debugging, this is a the compromise when debugging on the SH7055F. The debug stub also requires 384bytes of RAM for workspace. This is located at between 0xFFFFDE80 to 0xFFFFDFFF at the end of the 32kbytes of User RAM. *See SH7055F memory map on page 474.*

The debug stub also requires that some of the vectors in the vector table are reserved and set up to point to the debug stub, *see Watchdog timer on page 484*. When the debug stub is initially programmed into flash the whole of the 4kbyte vector table is programmed to vector into the debug stub. Some vectors are reserved and must not be altered when the debug tools are in operation, however you can change the non-reserved vectors as and when required.

If the debug stub or reserved vectors are accidentally overwritten by your application, debugging will be compromised or no longer available until the debug stub is reprogrammed, see *Programming the debug stub for the SH7055F*.

If the debug stub workspace (RAM) is corrupted by your application, the debugging tools will not function correctly until you initiate a target reset.

Programming the debug stub for the SH7055F

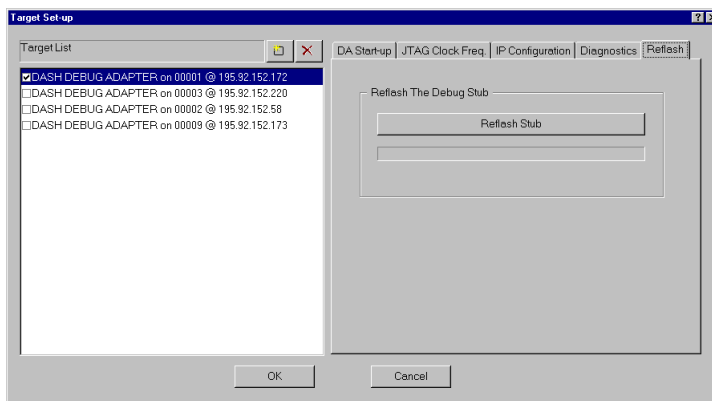
You will be required to program the debug stub in the SH7055F when:

- The SH7055F has never been programmed before.
- The debug stub has been overwritten by your application (addresses 0x0007C000 to 0x0007FFFF).
- Some or all of the reserved vectors have been overwritten.
- There is a version mismatch between the DASH firmware and the detected debug stub.

The most common reason to reprogram the debug stub is that it has been overwritten. This occurs if you have programmed final code into the SH7055F for testing without the debug tools. Final code need not contain debug stub code located at 0x0007C000 to 0x0007FFFF or any of the reserved vectors.

You can reprogram the debug stub and default vectors back into flash using the Reflash Stub command in CodeScape (Tools menu > Target/Communications > Configure... > Reflash tab. This uses the SH7055F's boot mode which is always available since it is native to the SH7055F.

Reflash Stub button in CodeScape



Watchdog timer

The watchdog timer is turned off whenever the debug stub is entered. The watchdog timer is restored to its previous state (turned on if required) when the debug stub returns program execution to your application.

When entering the debug stub the watchdog timer is turned off. The debug stub saves a copy of the TCNT (Watchdog Timer Counter) and the TCSR (Watchdog Timer Control/Status Register). The TME (Timer Enable) bit in the TCSR register is then set to zero.

When exiting the debug stub, the TCNT (Watchdog Timer Counter) and the TCSR (Watchdog Timer Control/Status Register) are restored to the previously saved values.

Debug stub requirements

User stack requirements

The debug stub requires 24 bytes to be made available in the user stack. The user stack must have a valid stack pointer at all times, that is register R15 must be long word aligned and pointing at RAM.

UBC registers

The MSTCR (module standby control register) MSTOP0 (bit 0) and MSTOP2 (bit 2) must remain clear at all times to allow the H-UDI and UBC modules to function, these are required by the debug stub.

Coding limitations

The interrupt mask (bits I3, I2, I1, I0) must be kept below 0xF (15). This allows the debug stub exceptions such as the UBC, and UDI interrupts to be accepted.

Flash memory programming

Flash programming management system

NOTE: Flash programming performed by the DASH is intended as a means to program the SH2e device with prototype code for debugging and development purposes only, Imagination Technologies do not guarantee the integrity of flash programming for use in final products or critical components.

Flash memory programming can be problematic because it is not possible to erase and program to flash on a byte-by-byte basis as might be required. Flash can only be erased in large fixed blocks between 4kbytes to 64kbytes. Flash programming is done in 128byte blocks on the SH7055F.

The debug tools improve the efficiency of flash programming by performing all the necessary reading, erasing and programming required to seamlessly allow flash to be programmed as if it is RAM.

Flash programming is achieved by using the Load Program File... or Load Binary... commands in CodeScape (File menu > Load Program File or Load Binary).

When Codescape issues either of these commands and the destination address of the code is in flash memory the following happens:

1. The DASH copies the entire contents of the target's flash memory into a memory buffer in the DASH.
2. Codescape issues a series of Write Memory commands to the DASH. These writes can be large or small to any area of flash you have requested in your application's address mapping. It can take many Write Memory commands to load a new program file or binary from Codescape.
3. The DASH compares the incoming data with its internal copy of the target's flash memory and overwrites its internal copy where there are differences, keeping track of any changes.
4. The DASH creates a list of erase blocks required to be erased on the target prior to reprogramming. This list is based on the changes tracked in the DASH's copy of the target's flash memory. The DASH then erases the target's flash memory block by block and subsequently reprograms each erased block with either new data or a combination of new data and previously stored data. Areas that have not changed are not erased or reprogrammed.

This method issues as few erase and program cycles as possible to preserve the life of the flash memory.

NOTE: *You cannot edit flash memory from a Memory region in CodeScape as you can with RAM.*

Minimizing program load times on the SH7055F

Since debugging on the SH7055F is normally performed in flash memory, there is a potential for long load times of up to one minute if your application is large. The load time can be greatly reduced with small amount of preparation.

The flash programming cycle consists of a load/erase/program cycle. The time consuming part of this process is the erasing and programming of the flash memory, the program load overhead is minimal in comparison. The key to minimizing the overall program load time is to minimize the amount of erasing and programming of the flash.

Flash is erased in large blocks from 4kbytes to 64kbytes as shown on *page 488*. Therefore, for example, reprogramming a small section (say two bytes) within block EB9 would require that the entire block of 64kbytes be erased. The erased flash would then need to be reprogrammed as a series of 128byte blocks.

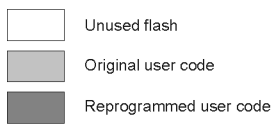
Permitted erase blocks in SH7055F Application Flash

Block Name	Block Size (kbytes)	Block Address Range
EB0	4	0x00000000 - 0x00000FFF
EB1	4	0x00001000 - 0x00001FFF
EB2	4	0x00002000 - 0x00002FFF
EB3	4	0x00003000 - 0x00003FFF
EB4	4	0x00004000 - 0x00004FFF
EB5	4	0x00005000 - 0x00005FFF
EB6	4	0x00006000 - 0x00006FFF
EB7	4	0x00007000 - 0x00007FFF
EB8	32	0x00008000 - 0x0000FFFF
EB9	64	0x00010000 - 0x0001FFFF
EB10	64	0x00020000 - 0x0002FFFF
EB11	64	0x00030000 - 0x0003FFFF
EB12	64	0x00040000 - 0x0004FFFF
EB13	64	0x00050000 - 0x0005FFFF
EB14	64	0x00060000 - 0x0006FFFF
EB15	64	0x00070000 - 0x0007FFFF

The flash programming management system avoids erasing and reprogramming entire erase blocks if no changes are made in that block. So for a large program that occupies several blocks, load times can be dramatically reduced by fragmenting your application into individual blocks that are partially filled. An edit in a particular block now does not cause the contents of other blocks to change.

Diagram *User code before optimizing* shows an application occupying a single contiguous block of memory from EB2 to EB6. Adding a small amount of code in EB2 causes every byte above to move up by the amount added, so blocks EB2, EB3, EB4, EB5 and EB6 are erased and reprogrammed to accommodate the change.

User code before optimizing



The effect of small change to user code before optimizing

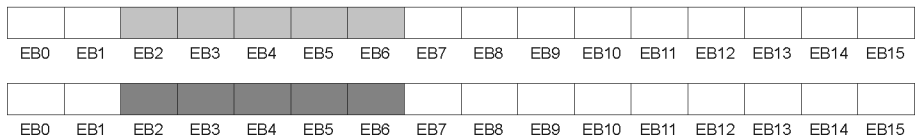
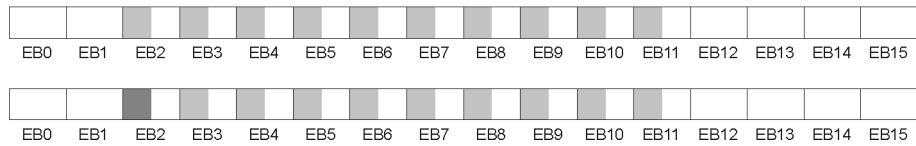


Diagram *User code after optimizing* the same code is broken up into segments. Now making the same change in EB2 results in only EB2 being erased and reprogrammed by the DASH saving considerable program load time.

User code after optimizing

The effect of a small change to user code after optimizing



Anomalies

Exception anomalies

The SH2e exhibits unexpected behavior when certain exceptions occur. These anomalies effect how the debug tools process and display exception events. These anomalies occur due to the design of the UBC controller and peculiarities of handling multiple exceptions within the SH2e's pipeline.

For a concise definition of the limitations imposed on exceptions refer to the *Hitachi SH7055 Hardware manual Rev. 1.0. Section 8.5*.

The following is a list of these anomalies as seen when debugging using Codescape.

CPU address error exception

If a CPU address error is encountered when running your application, your application is halted and the debug stub reports the exception to Codescape. The actual address at which your application is halted will probably not be at the instruction that actually caused the exception, but one or two instructions after the errant instruction. You must deduce which instruction caused the exception. This applies for applications running in both flash memory and RAM.

FPU error

The FPU error exhibits similar properties to the CPU address error exception. The address of the FPU error reported can be one or two instructions after the errant instruction.

DMA address error exception

The DMA error exhibits similar properties to the CPU address error exception. In this case the address of the error reported can be as much as three instructions after the errant instruction.

Inability to trace LDC, LDS, STC, STS and floating point instructions

When tracing through an application in flash memory (using the UBC) the instructions LDC, LDS, STC, STS and floating point instructions cannot be stopped at. On encountering one of these instructions when tracing, that instruction is executed and then the next instruction is traced. In this instance a double trace is performed. If more than one of the above instructions occurs in succession then multiple traces occur until an instruction that can be halted at is encountered.

Inability to breakpoint BT, BF instructions

The UBC mechanism does not work if a breakpoint is placed at the jump address of a BT and BF instruction (when the jump is made). This problem is corrected when tracing in Codescape as the errant instructions are simulated in software.

Non-delay branch instruction TRAPA

There is a problem with the UBC that prevents a break occurring on the instruction that is the destination of the branch. Consequently it is not possible to trace into TRAPA routines or to breakpoint the first address in TRAPA routine.

Non-maskable interrupts

Non-maskable interrupts cannot be used since they cause the UBC to fail. Consequently the debug system reserves the NMI and if the NMI is generated the debug stub is entered.

Interrupts

The first instruction of an interrupt cannot be breakpointed using the UBC.

SH3 Hardware Reference

ASE and Extended debug stubs

The debug stub has two parts, the ASE debug stub and the Extended debug stub. Two stubs are required because ASE memory is limited to 1kbyte and is insufficient to provide all the required functionality. The Extended debug stub, due to its size, must reside in external RAM.

The two stubs and their environment requirements are discussed in this chapter.

The ASE debug stub

The ASE debug stub resides in the SH3's 1kbyte of ASE memory. This memory can only be accessed in ASE mode by the debug tools, it cannot be physically read or written to by the application code. The ASE debug stub is a minimal debug stub loaded into ASE RAM after a target system reset. The ASE debug stub facilitates basic debugging, system initialization and the loading of the larger, functionally complete, Extended debug stub. Once the Extended debug stub is loaded, the ASE debug stub is used simply to chain ASE exceptions to handlers within the Extended debug stub. The ASE debug stub and ASE RAM are transparent to the application.

The Extended debug stub

The Extended debug stub resides in RAM and requires 16kbytes of memory. The Extended debug stub provides comprehensive debugging of the target. The debug stub's memory is not protected from corruption by the application code, thus the application must not attempt to use the memory space allotted to the Extended debug stub. An errant piece of application code can write over the Extended debug stub causing it to fail. Under these circumstances the debug stub (and debug session in progress) must be reloaded and restarted. The Extended debug stub has its own private stack within the boundaries of the allotted 16kbytes of RAM and thus does not require use of the application program's stack.

The location of the Extended debug stub is defined either by the boot ROM code, or by the settings in the DA Start-up dialog in CodeScape.

The Extended debug stub always has the same three instructions at the start of the 16kbyte section, BRK, RTE, NOP with the instruction codes 0x0000, 0x000B, and 0x0009.

Byte sequence:

- 0x00, 0x00, 0x0B, 0x00, 0x09, 0x00 - little endian Extended debug stub.
- 0x00, 0x00, 0x00, 0x0B, 0x00, 0x09 - big endian Extended debug stub.

Functional differences of the ASE and Extended debug stubs

The table below lists the differences in functionality of the two stubs.

Function	Extended debug stub	ASE debug stub	Comments
Read/Write memory	yes	yes	Extended: Optimized for speed. ASE: Optimized for size.
Read/Write General purpose registers.	yes	yes	
Read/Write DSP registers (on SH7729)	yes	yes	
Read/Write UBC registers	yes	yes	Extended: Optimized for speed. ASE: Achieved by multiple memory R/W accesses (much slower).
Read/Write MMU registers.	yes	yes	Extended: Optimized for speed. ASE: Achieved by multiple memory R/W accesses (much slower). ASE: Store queue info not available.
Default exception handler at VBR.	yes	no	Extended: Located wholly within the boundaries of the 16kbytes allotted.
Catch exceptions at VBR + 0x100.	yes	yes	Extended: Uses default exception handler. ASE: Uses system exception handler.
Catch exceptions at VBR + 0x400.	yes	yes	Extended: Uses default exception handler. ASE: Uses system exception handler.

Function	Extended debug stub	ASE debug stub	Comments
Catch interrupts at VBR + 0x600.	yes	yes	Extended: Uses default exception handler. ASE: Uses system exception handler.
Exception passback to stub.	yes	no	
Profiling, statistical sampling.	yes	no	
BIOS calls for virtual channels.	yes	no	
BIOS calls for utility functions.	yes	no	
Supports little endian systems.	yes	yes	Target endianness is tested on power up and the appropriate stub is loaded.
Supports big endian systems	yes	yes	Target endianness is tested on power up and the appropriate stub is loaded.
Require system memory to be present and configured.	yes	no	Extended: Requires 16kbytes (anywhere in RAM). ASE: Required no RAM at all.
Facilitates debugging of code in ROM.	yes	yes	
Facilitates debugging of code in RAM.	yes	yes	
Facilitates the use of software breakpoints	yes	yes	
Facilitates the use of hardware breakpoints	yes	yes	
ROMless system 'bring up'.	yes	yes	
RAMless system 'investigation'.	no	yes	
ASE mode used for debugging.	yes	yes	Extended: Vectored into from ASE RAM. ASE: Used inherently.
Stub memory protected for application code.	no	yes	

Optimization of the stubs

The ASE debug stub must fit entirely into 1kbyte (512 instructions), for this reason all of the ASE debug stub's functions are optimized for size. In some cases functionality is delegated from the ASE debug stub to the DASH.

An example of this is the 'Read Context UBC' command. This command is issued by CodeScape to request that the values of all of the UBC's (user break controller) memory mapped registers are returned.

In the Extended debug stub, a protocol exists for this purpose. The DASH requests the UBC's context and the Extended debug stub replies by reading the required registers and returning them in a single transaction.

In the ASE debug stub no such protocol exists between the DASH and the ASE debug stub, However, the same result is achieved by the DASH issuing a series of 'Read Memory' commands to read each of the individual registers one by one.

The DASH and Extended debug stub performs the UBC context read in the most efficient (fastest) method possible, whereas the DASH and the ASE debug stub performs the UBC context read using the minimum amount of code in the ASE debug stub. The net result is that the Extended debug stub runs much faster than the ASE debug stub.

Exception handling in the ASE debug stub

When using the ASE debug stub for debugging, CodeScape will be limited to using hardware and software breakpoints and basic trace functions such as single stepping. When using exceptions with the ASE debug stub, you must provide your own exception handlers and there is no exception passback facility as with the Extended debug stub.

Exception handling in the Extended debug stub

The Extended debug stub handles exceptions and interrupts by installing a default exception handler which sets up the VBR register accordingly. Alternatively, you can use your own application exception handler, see *"Interrupts and Exceptions" on page 498*.

Caching during BIOS calls

When a BIOS call is made the debug stub runs from the SH3's P1 area (cacheable). If the SH3's cache is enabled then the speed of the BIOS call is improved due to the debug stub being in a cacheable area. The debug stub does not implicitly enable the cache during a BIOS call.

Interrupts and Exceptions

This section deals with issues concerning exception and interrupt handling on the target and the interaction and interrupt requirements of the debug stub.

Exception handling with or without a boot ROM

Once the Extended debug stub has loaded, the DASH instructs the debug stub to either resume execution of the boot ROM (if one is installed), or set up a default environment and wait for the user to perform some action such as downloading a program. This is controlled by the option: *Halt after stub load* on the *DA Start-up options* dialog in CodeScape.

The debug stub implements exceptions using different methods depending on this condition.

Resume (debugging with a boot ROM)

This mode exists to allow program execution to resume back to the boot ROM after the Extended debug stub is loaded. This is achieved with the minimum of disruption to the state of the target microprocessor context.

Halt (debugging without a boot ROM)

This mode exists to allow the Extended debug stub to be loaded and then the context to be set up with default settings to allow a debugging session to commence. In this mode CodeScape does not pass program control back to the boot ROM, instead, control remains within the debug stub monitor.

- The VBR register is loaded with the default debug stub exception handler.
- The stack pointer is loaded with the default stack pointer.

-
- The status register block bit is cleared to 0.

Exceptions during debugging with a boot ROM

When *Resume after stub load* mode is selected on the DA Start-up dialog in CodeScape, the debug stub does not implement a exception or interrupt handler of its own and it does not alter or set up the VBR register. Thus the debug stub makes no demands of the application code and does not introduce any time penalties by running debug stub exception handler code. However, this does mean that the application code must implement its own handler if exceptions or interrupts are to be used by the application code itself.

For the debug stub to process exceptions (not interrupts) in this mode, it incorporates a passback facility to allow unhandled exceptions to be caught and processed by the debug stub. To implement the passback facility the application code exception handler must call the debug stub using the sequence of instructions below.

For exceptions (VBR + 0x100 and VBR + 400) the passback call takes the form:

```
BRK
RTE
RTE
NOP
```

This special sequence causes the debug stub to be entered via the 'BRK' instruction and the double 'RTE' immediately after the break indicates to the debug stub that this is an exception passback call.

This sequence should be added to the end of the application code exception handler to allow the debug stub to process any unhandled exceptions. The 'NOP' at the end is not strictly required but can be added to inhibit compiler or assembler errors. You should use the passback sequence where you would normally put code to deal with unhandled exceptions. Where an exception is handled correctly by the application code, an 'RTE' must be executed rather than the passback sequence.

Notes:

1. *After the debug stub processes an unhandled exception, program execution is returned to the interrupted code, control is not returned to the application exception handler so no attempt is made to execute the 'RTE RTE' element of the passback sequence.*
2. *Some assemblers and compilers may not allow an RTE opcode following an RTE opcode. In this event the following sequence could be used:*

```
BRK
```

```
RTE
DC.W      0x002b
NOP
```

3. *Unhandled exceptions passed back to the debug stub are reported to CodeScape and this in turn is interpreted and reported as specified by CodeScape.*
4. *The application code must take care to ensure the SR.BL block bit is handled correctly to ensure exceptions are accepted (refer to the Hitachi SH3 hardware manual for further details).*
5. *For the passback facility to function correctly the application code must not change the contents of the EXPEVT register prior to issuing the passback call.*

Exceptions during debugging without a boot ROM

When “Halt after stub load” mode is selected on the DA Start-up dialog in CodeScape, the debug stub implements a default exception and interrupt handler by initializing the VBR register to point at the debug stub default handler.

The default handler allows all exceptions and interrupts to be caught by the debug stub and in turn reported by CodeScape. This allows application code to be debugged without the need for an application code handler to catch exceptions such as 'address error'.

The application code can at any time install its own handler by changing the contents of the VBR. Once this is done the debug stub default handler will no longer function but application code can utilise the passback method as described in the previous section to allow unhandled exceptions to be handled by the debug stub.

If the VBR is changed by the application code, the debug stub will make no attempt to restore the VBR to point at the debug stub default handler. However, the application code can restore the VBR to the default value to make the default handler functional again.

Note:

If 'halt after stub load' debugging mode is selected the SR.BL block bit defaults to 0.

Interrupts during debugging

An additional specific passback sequence is available to allow unhandled interrupts (not exceptions) to be captured and reported to CodeScape. To implement this passback facility the application code interrupt handler must call the debug stub using the following sequence of instructions.

BRK
RTE
RTS
NOP

The above sequence should be implemented in a similar manner to the exception passback sequence described above. This sequence must only be used to passback unhandled interrupts for the interrupt handler at the address $VBR + 0x600$.

Caveats and limitations

Use of the UBC registers

CodeScape makes use of the UBC unit to implement hardware breakpoints, for this reason the application code must not attempt to access the UBC's registers.

Debugging exception handlers using CodeScape

It is possible to 'lose' hardware breakpoints when debugging application exception handlers. This occurs if the handler routine being debugged does not allow for nested exceptions to occur. A nested exception is required in this instance to allow the UBC exception to be accepted and the debug stub default handler to be called. It is recommended that software breakpoints are used where possible when debugging exception handlers.

For information on allowing nested interrupts to occur refer to the Hitachi SH3 hardware manual.

The following notes outline the basic methodology. On entering the application exception handler:

1. Save the SPC to a temporary location.
2. Save the SSR to a temporary location.
3. Clear the SR.BL bit to 0 (this now allows exceptions or interrupts to occur).
4. Execute the application exception handler code.
5. Set the SR.BL bit to 1. (this inhibits exceptions or interrupts from occurring when restoring the SPC and SSR).
6. Restore the SSR to the original value from the temporary location.
7. Restore the SPC to the original value from the temporary location.
8. RTE

Notes:

- 1. If an interrupt occurs within the exception handler prior to the SR.BL bit being cleared, then it will be held until the SR.BL bit is cleared.*
- 2. If an exception occurs within the exception handler prior to the SR.BL bit being cleared or after it has been set, then it will cause a manual reset to occur causing program execution to transfer to the reset vector (highly undesirable). This situation will occur if the exception handler contains a programming error in the code such as an instruction address error. Care must be taken with writing application exception handlers to ensure no such errors exist, specifically when the SR.BL bit is set.*

SH4 Hardware Reference

ASE and Extended debug stubs

The debug stub has two parts, the ASE debug stub and the Extended debug stub. Two stubs are required because ASE memory is limited to 1kbyte and is insufficient to provide all the required functionality. The Extended debug stub, due to its size, must be resident in external RAM.

The two stubs and their environment requirements are discussed in this chapter.

The ASE debug stub

The ASE debug stub resides in the SH4's 1kbyte of ASE memory. This memory can only be accessed in ASE mode by the debug tools, it cannot be physically read or written to by the application code. The ASE debug stub is a minimal debug stub loaded into ASE RAM after a target system reset. The ASE debug stub facilitates basic debugging, system initialization and the loading of the larger, functionally complete, Extended debug stub. Once the Extended debug stub is loaded the ASE debug stub is used simply to chain ASE exceptions to handlers within the Extended debug stub. The ASE debug stub and ASE RAM are transparent to the application.

The Extended debug stub

The Extended debug stub resides in RAM and requires 16kbytes of memory. The Extended debug stub provides comprehensive debugging of the target. The debug stub's memory is not protected from corruption by the application code, thus the application must not attempt to use the memory space allotted to the Extended debug stub. An errant piece of application code can write over the Extended debug stub causing it to fail. Under these circumstances the debug stub (and debug session in progress) must be reloaded and restarted. The Extended debug stub has its own private stack within the boundaries of the allotted 16kbytes of RAM and thus does not require use of the application program's stack.

The location of the Extended debug stub is defined either by the boot ROM code, or by the settings in the DA Start-up dialog in CodeScape.

The Extended debug stub always has the same three instructions at the start of the 16kbyte section, BRK, RTE, NOP with the instruction codes 0x00, 0x000B, and 0x0009.

Byte sequence:

- 0x, 0x00, 0x0B, 0x00, 0x09, 0x00 - little endian Extended debug stub.
- 0x00, 0x, 0x00, 0x0B, 0x00, 0x09 - big endian Extended debug stub.

Functional differences of the ASE and Extended debug stubs

The table below lists the differences in functionality of the two stubs. **Optimization of the stubs**

The ASE debug stub must fit entirely into 1kbyte (512 instructions), for this reason all of the ASE debug stub's functions are optimized for size. In some cases functionality is delegated from the ASE debug stub to the DASH.

An example of this is the 'Read Context BC' command. This command is issued by CodeScape to request that the values of all of the BC's (break controller) memory mapped registers are returned.

In the Extended debug stub, a protocol exists for this purpose. The DASH requests the BC's context and the Extended debug stub replies by reading the required registers and returning them in a single transaction.

In the ASE debug stub no such protocol exists between the DASH and the ASE debug stub. However, the same result is achieved by the DASH issuing a series of 'Read Memory' commands to read each of the individual registers one by one.

The DASH and Extended debug stub performs the BC context read in the most efficient (fastest) method possible, whereas the DASH and the ASE debug stub performs the BC context read using the minimum amount of code in the ASE debug stub. The net result is that the Extended debug stub runs much faster than the ASE debug stub.

Exception handling in the ASE debug stub

Exception handling in the Extended debug stub

The Extended debug stub handles exceptions and interrupts by installing a default exception handler which sets up the VBR register accordingly. Alternatively, you can use your own application exception handler, see *“Interrupts and Exceptions” on page 510*.

Caching during BIOS calls

When a BIOS call is made the debug stub runs from the SH4's P1 area (cacheable). If the SH4's cache is enabled then the speed of the BIOS call is improved due to the debug stub being in a cacheable area. The debug stub does not implicitly enable the cache during a BIOS call.

Interrupts and Exceptions

This section deals with issues concerning exception and interrupt handling on the target and the interaction and interrupt requirements of the debug stub.

Exception handling with or without a boot ROM

Once the Extended debug stub has loaded, the DASH instructs the debug stub to either resume execution of the boot ROM (if one is installed), or set up a default environment and wait for the user to perform some action such as downloading a program. This is controlled by the option: *Halt after stub load* on the *DA Start-up options* dialog in CodeScape.

The debug stub implements exceptions using different methods depending on this condition.

Resume (debugging with a boot ROM)

This mode exists to allow program execution to resume back to the boot ROM after the Extended debug stub is loaded. This is achieved with the minimum of disruption to the state of the target microprocessor context. *Halt (debugging without a boot ROM)*

This mode exists to allow the Extended debug stub to be loaded and then the context to be set up with default settings to allow a debugging session to commence. In this mode CodeScape does not pass program control back to the boot ROM, instead, control remains within the debug stub monitor.

- The VBR register is loaded with the default debug stub exception handler.
- The stack pointer is loaded
- The status register block bit is cleared to 0.

Exceptions during debugging with a boot ROM

When *Resume after stub load* mode is selected on the *DA Start-up* dialog in CodeScape, the debug stub does not implement an exception or interrupt handler of its own and it does not alter or set up the VBR register. Thus the debug stub makes no demands of the application code and

does not introduce any time penalties by running debug stub exception handler code. However, this does mean that the application code must implement its own handler if exceptions or interrupts are to be used by the application code itself.

For the debug stub to process exceptions (not interrupts) in this mode, it incorporates a passback facility to allow unhandled exceptions to be caught and processed by the debug stub. To implement the passback facility the application code exception handler must call the debug stub using the sequence of instructions below.

For exceptions (VBR + 0x100 and VBR + 400) the passback call takes the form:

```
BRK
RTE
RTE
NOP
```

This special sequence causes the debug stub to be entered via the 'BRK' instruction and the double 'RTE' immediately after the break indicates to the debug stub that this is an exception passback call.

This sequence should be added to the end of the application code exception handler to allow the debug stub to process any unhandled exceptions. The 'NOP' at the end is not strictly required but can be added to inhibit compiler or assembler errors. You should use the passback sequence where you would normally put code to deal with unhandled exceptions. Where an exception is handled correctly by the application code, an 'RTE' must be executed rather than the passback sequence.

Notes:

- 1. After the debug stub processes an unhandled exception, program execution is returned to the interrupted code, control is not returned to the application exception handler so no attempt is made to execute the 'RTE RTE' element of the passback sequence.*
- 2. Some assemblers and compilers may not allow an RTE opcode following an RTE opcode. In this event the following sequence could be used:*

```
BRK
RTE
DC.W      0x002b
NOP
```

- 3. Unhandled exceptions passed back to the debug stub are reported to CodeScape and this in turn is interpreted and reported as specified by CodeScape.*
- 4. The application code must take care to ensure the SR.BL block bit is handled correctly to ensure exceptions are accepted (refer to the Hitachi SH4 hardware manual for further details).*

5. *For the passback facility to function correctly the application code must not change the contents of the EXPEVT register prior to issuing the passback call.*

Exceptions during debugging without a boot ROM

When “*Halt after stub load*” mode is selected on the DA Start-up dialog in CodeScape, the debug stub implements a default exception and interrupt handler by initializing the VBR register to point at the debug stub default handler.

The default handler allows all exceptions and interrupts to be caught by the debug stub and in turn reported by CodeScape. This allows application code to be debugged without the need for an application code handler to catch exceptions such as 'address error'.

The application code can at any time install its own handler by changing the contents of the VBR. Once this is done the debug stub default handler will no longer function but application code can utilise the passback method as described in the previous section to allow unhandled exceptions to be handled by the debug stub.

If the VBR is changed by the application code, the debug stub will make no attempt to restore the VBR to point at the debug stub default handler. However, the application code can restore the VBR to the default value to make the default handler functional again.

Note:

If 'halt after stub load' debugging mode is selected the SR.BL block bit defaults to 0.

Interrupts during debugging

An additional specific passback sequence is available to allow unhandled interrupts (not exceptions) to be captured and reported to CodeScape. To implement this passback facility the application code interrupt handler must call the debug stub using the following sequence of instructions.

```
BRK
RTE
RTS
NOP
```

The above sequence should be implemented in a similar manner to the exception passback sequence described above. This sequence must only be used to passback unhandled interrupts for the interrupt handler at the address $VBR + 0x600$.

Caveats and limitations

Use of the BC registers

CodeScape makes use of the BC unit to implement hardware breakpoints, for this reason the application code must not attempt to access the BC's registers.

Debugging exception handlers using CodeScape

It is possible to 'lose' hardware breakpoints when debugging application exception handlers. This occurs if the handler routine being debugged does not allow for nested exceptions to occur. A nested exception is required in this instance to allow the BC exception to be accepted and the debug stub default handler to be called. It is recommended that software breakpoints are used where possible when debugging exception handlers.

For information on allowing nested interrupts to occur refer to the Hitachi SH4 hardware manual.

The following notes outline the basic methodology. On entering the application exception handler:

1. Save the SPC to a temporary location.
2. Save the SSR to a temporary location.
3. Clear the SR.BL bit to 0 (this now allows exceptions or interrupts to occur).
4. Execute the application exception handler code.

5. Set the SR.BL bit to 1. (this inhibits exceptions or interrupts from occurring when restoring the SPC and SSR).
 6. Restore the SSR to the original value from the temporary location.
 7. Restore the SPC to the original value from the temporary location.
-

8. RTE

Notes:

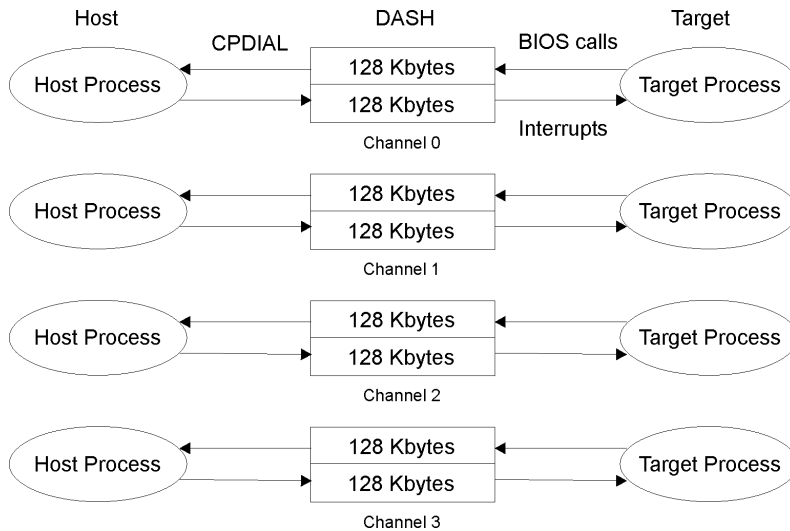
1. *If an interrupt occurs within the exception handler prior to the SR.BL bit being cleared, then it will be held until the SR.BL bit is cleared.*
2. *If an exception occurs within the exception handler prior to the SR.BL bit being cleared or after it has been set, then it will cause a manual reset to occur causing program execution to transfer to the reset vector (highly undesirable). This situation will occur if the exception handler contains a programming error in the code such as an instruction address error. Care must be taken with writing application exception handlers to ensure no such errors exist, specifically when the SR.BL bit is set.*

Communications Channels

About channels

Channels provide a means of fast communication (500kbytes/s) between programs running the development computer and application code running on the target. There are four bi-directional channels that may be read from and written to independently either a byte at a time or in blocks.

The DASH has a 128kbyte input/output buffers for each channel to ensure uninterrupted communication.



Channels are implemented at the target end using ASE BIOS calls and interrupts, and from the development computer using the Debug Interface Adapter Library (CPDIAL).

Channels are reserved for use as follows:

Channel Number	Use
0	Spare.
1	File Server (Debugging File Server Protocol).
2	Spare.
3	Spare.

ASE BIOS services

At the target end, channel access is implemented using ASE BIOS calls made by the target application software to the debug stub, or interrupts issued by the debug stub to the target application interrupt handler.

ASE BIOS calls

Six BIOS calls are available for use by the application software. The BIOS calls are non-blocking to allow the target application to issue a BIOS call and receive a reply immediately. The return values of the reply indicate the success or failure to carry out the requested task.

- Read byte, INCHR.
- Write byte, OUTCHR.
- Read buffer, RDBF.
- Write buffer, WRBF.
- Read channel buffer status, RDSTAT.
- Set and read channel interrupts, CHISR.

Interrupts

The DASH issues an interrupt to the target application interrupt handler when a specified channel buffer condition is met, see *“Set and Read Channel Interrupts, CHISR” on page 529*.

Accessing ASE BIOS services

NOTE: The examples given here are for SH2 and SH4 processors. For SH3 processors substitute registers R4, R5, R6, R7, R0 with R10, R11, R12, R13, R8 respectively.

In privileged mode the ASE BIOS services are accessed by:

1. Loading registers R4, R5, R6, R7 with the required parameters.
2. Executing the following sequence:
BRK
RTS
RTE
NOP

In user mode (not applicable to SH2 processors) the ASE BIOS services are accessed by:

1. Loading registers R4, R5, R6, R7 with parameters.
2. TRAPA #02.

Notes:

1. ASE BIOS services are provided by the Extended debug stub resident in target system RAM. The ASE debug stub provides just one service, Instruct the DASH where to load the Extended stub.
2. The ASE BIOS accepts any value for the destination address, therefore calling code should validate the supplied parameters in the TRAPA handler.
3. If an application exception handler is in use, the debug stub exception passback must be implemented to allow the TRAPA #02 to be handled by the debug stub. The application exception handler must not alter the values of EXPEVT and TRA prior to issuing the passback call.
4. In privileged mode the TRAPA #02 instruction can also be used to issue a BIOS call subject to the same validation as above.
5. BIOS calls can be issued using TRAPA #02 when the debug stub's default exception handler is in use. In this instance the default handler will issue the BIOS call in a way that is transparent to the application code.
6. ASE BIOS services should only be used when the target is started in debug mode (not applicable to SH4).

Setup registers

To use the ASE BIOS services, set up the registers as shown in the following table.

Name	Description	R4	R5	R6	R7	LOC
INCHR	Read byte	0x0A00 + Channel Number.	Not used.	Not used.	Not used.	Not used.
OUTCHR	Write byte.	0x0B00 + Channel Number.	Byte to write.	Not used.	Not used.	Not used.
RDBF	Read buffer.	0x0A08 + Channel Number.	Maximum read size.	Destination address.	Address of LOC.	Number of bytes read.
WRBF	Write buffer.	0x0B08 + Channel Number.	Number of bytes to write.	Source address.	Address of LOC.	Number of bytes written.
RDSTAT	Read channel buffer status.	0x0C00 + Channel Number.	Not used.	Not used.	Address of LOC.	Status.

Results

After the BIOS call, R0 and LOC hold the results as follows:

Operation	R0	LOC
INCHR	Byte read in low word, error code in high word.	Not used.
OUTCHR	Error code in high word.	Not used.
RDBF	Error code in high word.	Number of bytes read.
WRBF	Error code in high word.	Number of bytes written.
RDSTAT	Error code in high word.	Data available, space available.

Error codes

There are nine possible error codes, stored in the high order 16 bits of R0:

Name	Value	Meaning
CONAOK	0	No error, the operation was successful.
CONERR	1	Fatal error. The DASH suffered an internal error.
CONBAD	2	Unknown command.
CONPRM	3	Parameter error. A buffer count of zero was requested.
CONADR	4	Bad address.
CONCNT	5	Bad count. A buffer count exceeding the maximum allowable value was requested. The current maximum value is 65536 bytes.
CONCBF	6	Channel buffer full. If there is insufficient space to write all the requested number of bytes, then as much data will be written as possible. When the buffer is full the application is informed that no further data was written because there was no more space available.
CONCBE	7	Channel buffer empty. The application is informed that there was no data to read.
CONBSY	8	Channel busy.

NOTE: *An ASE BIOS call may use a subset of the above codes for its return codes.*

ASE BIOS calls

The examples given here are for SH2 and SH4 processors. For SH3 processors substitute registers R4, R5, R6, R7, R0 with R10, R11, R12, R13, R8 respectively.

Read byte, INCHR

Reads a single byte of data from the specified channel buffer to the target.

Call parameters

R4	0xA00 + channel (0,1,2,3)
R5	Not used.
R6	Not used.
R7	Not used.
LOC	Not used.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBE, channel buffer empty. CONERR, fatal error. Byte read in low order 8 bits.
LOC	Not used.

Example BIOS call

```
; OS (Channel 0) issue a 'read byte non-blocking' BIOS call.
mov.l    #(INCHR + OS),r4 ;BIOS command + channel (0xA00 + 0).
dc.w    0x003b             ;BRK
dc.w    0x000b             ;RTS
dc.w    0x002b             ;RTE
nop                      ;NOP
; After call:
; Byte Read returned in low word R0.
; Error code returned in high word R0.
```

Write byte, OUTCHR

Writes a single byte of data from the target to the specified channel buffer.

Call parameters

R4	0xB00 + channel (0,1,2,3).
R5	Byte of data to write.
R6	Not used.
R7	Not used.
LOC	Not used.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBF, Channel buffer full. CONERR, Fatal error.
LOC	Not used.

Example BIOS call

```
; OS (Channel 0) issue a 'write byte non-blocking' BIOS call.
mov.l #(OUTCHR + OS),r4    ;BIOS command + channel (0xB00 + 0).
mov.l #0xa5,r5             ;Byte to write (e.g. 0xa5).
dc.w 0x003b                ;BRK
dc.w 0x000b                ;RTS
dc.w 0x002b                ;RTE
nop                        ;NOP
; After call:
; Error code returned in high word R0.
```

Read buffer, RDBF

Reads a number of bytes of channel data, up to a specified maximum.

Call parameters

R4	0xA08 + channel (0,1,2,3).
R5	Byte Count. The maximum transfer size of data to be read from the specified channel buffer, up to a maximum of 65536 bytes.
R6	Destination address of data read from the specified channel buffer.
R7	Zero, or address of LOC to put Byte Count.
LOC	Zero or a 32-bit address to store the actual number of bytes read from the specified channel buffer.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONPRM, Bad parameter. CONCNT, Bad count. CONCBE, Channel buffer empty. CONERR, Fatal error.
LOC	The number of bytes actually read (32-bits) or zero if the channel buffer was empty.

Example BIOS call

```
; OS (Channel 0) issue a 'read buffer non-blocking' BIOS call.
mov.l #(RDBF + OS),r4 ;BIOS command + channel (0xA08 + 0).
mov.l #200,r5          ;Max number of bytes to read.
mov.l #buffer,r6        ;Pointer to destination buffer area.
mov.l #loc,r7           ;Location to report actual number
                        ;of bytes read after BIOS call.

dc.w 0x003b            ;BRK
dc.w 0x000b            ;RTS
dc.w 0x002b            ;RTE
nop                    ;NOP

; After call:
; Error code returned in high word R0.
; 'Loc' contains the number of bytes read.
; 'buffer' contains 'loc' number of data bytes read.
```


Write buffer, WRBF

Writes a number of bytes, up to a specified maximum, to the specified channel buffer.

Call parameters

R4	0xB08 + channel (0,1,2,3).
R5	Byte Count. The number of bytes to be written to the specified channel buffer, up to a maximum of 65536 bytes.
R6	Source address. The address in target memory to read data from.
R7	Zero, or address of LOC to put the Byte Count. If this value is non-zero, then the value points to a 32-bit location to store the actual number of bytes written to the DASH buffer.
LOC	32-bit location to hold the number of bytes transferred.

Returned data

R0	In high order 16 bits, one of: CONAOK, No error. CONCBF, Channel buffer full. CONERR, Fatal error.
LOC	Number of bytes actually written (32 bits) or zero if the specified channel buffer was full.

Example BIOS call

```
; OS (Channel 0) issue a 'write buffer non-blocking' BIOS call.
mov.l #(WRBF + OS),r4 ;BIOS command + channel (0xB08 + 0).
mov.l #200,r5         ;Max number of bytes to write.
mov.l #buffer,r6      ;Pointer to source buffer area.
mov.l #loc,r7         ;Location to report actual number
                     ;of bytes written after BIOS call.
dc.w 0x003b           ;BRK
dc.w 0x000b           ;RTS
dc.w 0x002b           ;RTE
nop                  ;NOP
; After call:
; Error code returned in high word R0.
; 'Loc' contains the number of bytes written.
```

Read channel buffer status, RDSTAT

Informs the target application of amount of data available to be read from and space available to be written to in the specified channel buffer.

Call parameters

R4	0xC00 + channel (0,1,2,3).
R5	Not used.
R6	Not used.
R7	Address of LOC to buffer status information.
LOC	64-bit location to hold bytes of data available (long), bytes of space available (long).

Returned data

R0	CONAOK.
LOC	64-bit location to hold bytes of data available (long), followed by bytes of space available (long).

Example BIOS call

```
; OS (Channel 0) issue a 'read channel buffer status' BIOS call.
mov.l #(RDSTAT + OS),r4    ;BIOS command + channel (0xC00 + 0).
mov.l #loc,r7              ;Location to report the actual
                           ;status information
dc.w 0x003b                ;BRK
dc.w 0x000b                ;RTS
dc.w 0x002b                ;RTE
nop                        ;NOP
; After call:
; Error code returned in high word R0.
; 'Loc' (long) contains the number of bytes available to be read.
; 'Loc'+4 (long) contains the space in bytes available for writing.
```

Set and Read Channel Interrupts, CHISR

Use CHISR to service buffers:

- Request an interrupt if a transmitting buffer is empty or a receiving buffer is full.
- Determine if data is present in a channel buffer.
- Acknowledge receipt of interrupts by the application interrupt handler.

Call Parameters

R4	CHISR = 0x900
R5	Address of a long word LOC, must be non-zero.
R6	Not used.
R7	Not used.
LOC	Four 8-bit fields, as described below.

Name	0	1	Channel 2	3	Set bit to:
IPISSET ¹	0 (LSB)	8	16	24	Change input buffer interrupts (bits 1-2).
IPSIE ⁵	1	9	17	25	Enable a single input buffer interrupt, clear to disable. Valid only if bit 0 is set.
IPMIE ³	2	10	18	26	Enable multiple interrupts, clear to disable. Valid only if bit 0 is set.
IPACK	3	11	19	27	Acknowledge receipt of interrupt when data is present in the input buffer.
OPISET ²	4	12	20	28	Change output buffer interrupts (bits 5-6).
OPSIE ⁵	5	13	21	29	Enable a single output buffer interrupt, clear to disable. Valid only if bit 4 is set.
OPMIE ⁴	6	14	22	30	Enable multiple interrupts, clear to disable. Valid only if bit 4 is set.
OPACK	7	15	23	31(MSB)	Acknowledge receipt of an interrupt when space is available in the output buffer.

Returned data

R0	One of CONAOK, CONERR, CONPRM in the high order 16 bits.
LOC	Four 8-bit fields, as described below.

Name	Channel				Meaning when bit set
	0	1	2	3	
IPINT	0 (LSB)	8	16	24	An input buffer interrupt has occurred.
IPSIE	1	9	17	25	Input buffer interrupts are enabled.
IPMIE	2	10	18	26	Multiple interrupts are enabled.
IPRDY	3	11	19	27	There is data in the input buffer.
OPINT	4	12	20	28	An output buffer interrupt has occurred.
OPSIE	5	13	21	29	Output buffer interrupts are enabled.
OPMIE	6	14	22	30	Multiple interrupts are enabled.
OPRDY	7	15	23	31 (MSB)	There is space in the output buffer.

Bits 0-7 apply to channel 0, 8-15 to channel 1, 16-23 to channel 2, and 24-31 to channel 3.

Notes

1. *IPSIE and IPMIE are ignored if IPISSET is set to 0.*
2. *OPSIE and OPMIE are ignored if OPISET is set to 0.*
3. *IPMIE is ignored if IPSIE is set to 0.*
4. *OPMIE is ignored if OPSIE is set to 0.*
5. *IPSIE and OPSIE are cleared automatically on acknowledgment of an interrupt.*

Servicing buffers using CHISR

There are two ways to use the CHISR service to inform the application software there is a buffer for it to service:

- **Interrupt control**
Trigger an interrupt when channel buffer data becomes available or channel space is available.
- **Polling**
Poll a channel to determine if a transmitting buffer is empty or a receiving buffer is non-empty.

Interrupt control

The CHISR BIOS call allows the DASH to interrupt the target if one of these conditions are met:

- Data is in the input buffer for the application software to read and process.
- Space is available in the output buffer for the application software to store data in the buffer, ready for the host to read and process.

You can generate single or multiple interrupts when a condition is met. Single interrupts are generated once-only when an interrupt condition is met. Multiple interrupts are generated each time the channel condition is met.

Notes:

1. *The interrupt is not an ASE mode interrupt.*
2. *The interrupt vectors to the application software interrupt handler. It is the responsibility of the application software to ensure this is present.*
3. *CHISR can return with more than one 'interrupt occurred' bit set.*

Example BIOS call

This example shows a typical BIOS call to set up and clear interrupts for the four channels.

```
; Use the 'set and read interrupts' BIOS call to set up interrupts.
;
;   mov.l  #loc,r5           ;Location for set up parameters.
;
; Channel 0. Operating system. Set for single interrupt on input
; buffer (data available).
;   mov.b  #(IPISET + IPSIE),r0
```

```
        mov.b  r0,@r5
;
; Channel 1. File server. Set up for multiple interrupts on input
; buffer (data available) and output buffer (space available).
        mov.b  #(IPSET + IPMIE + OPISET + OPMIE),r0
        mov.b  r0,@(1,r5)
;
; Channel 2. Sound tools. Turn off interrupts.
        mov.b  #(IPISET + OPISET),r0
        mov.b  r0,@(2,r5)
;
; Channel 3. Unused. Make no changes to channel 3.
        mov.b  #0x00,r0
        mov.b  r0,@(3,r5)
;
; Issue a 'Set and read channel interrupts' BIOS call.
        mov.l  #CHISR,r4          ;BIOS command 0x900.
        dc.w  0x003b              ;BRK
        dc.w  0x000b              ;RTS
        dc.w  0x002b              ;RTE
        nop                       ;NOP
; After call:
; Error code returned in high word R0.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
; status.
```

Polling

The CHISR BIOS call can be used to determine the state of individual channel buffers. This allows polling to be implemented instead of interrupt control.

Example BIOS call

This example shows a typical BIOS call to interrogate the four channels' buffers.

```
; Use the 'set and read interrupts' BIOS call to read channel status.
        mov.l  #loc,r5            ;Location for set up parameters.
        mov.b  #0,r0
        mov.b  r0,@r5            ;No change channel 0
        mov.b  r0,@(1,r5)        ;No change channel 1
        mov.b  r0,@(2,r5)        ;No change channel 2
        mov.b  r0,@(3,r5)        ;No change channel 3
;
; Issue a 'Set and read channel interrupts' BIOS call.
        mov.l  #CHISR,r4          ;BIOS command 0x900.
        dc.w  0x003b              ;BRK
        dc.w  0x000b              ;RTS
        dc.w  0x002b              ;RTE
        nop                       ;NOP
; After call:
; Error code returned in high word R0.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
```

```

; status.
    mov.l #loc,r5          ;Location for returned parameters.
    mov.b @r5+,r1          ;Current Channel 1 status.
    mov.b @r5+,r2          ;Current Channel 2 status.
    mov.b @r5+,r3          ;Current Channel 3 status.
    mov.b @r5+,r4          ;Current Channel 4 status.
; The registers r1,r2,r3,r4 now contain the details for each channel
; for the following items:
;   Input buffer data available/no data available.
;   Output buffer space available/no space available.
; and also:
;   Input buffer occurred/not occurred.
;   Input buffer interrupts enabled/disabled.
;   Input buffer multiple interrupts enabled/disabled.
;   Output buffer occurred/not occurred.
;   Output buffer interrupts enabled/disabled.
;   Output buffer multiple interrupts enabled/disabled.

```

Acknowledging receipt of an interrupt

You can use CHISR to acknowledge the receipt of an interrupt. The application interrupt handler must include the functionality to acknowledge interrupts for the following reasons:

- The interrupt cannot be guaranteed to be accepted by the processor.
- The interrupt can be lost if the debug stub is entered at the same time as an interrupt is generated, for example when executing a software BRK breakpoint.

The DASH will repeatedly issue interrupts (when interrupts are enabled) until the application software has acknowledged the interrupt by calling CHISR.

When the application software interrupt handler is entered due to an interrupt, the CHISR command with the relevant acknowledge bits set should be issued. This will inform the DASH that the interrupt has been accepted and prevent it from re-issuing the interrupt again.

Example BIOS call

This example shows a typical BIOS call to acknowledge an input interrupt on channel 1.

```

; Use the 'set and read interrupts' BIOS call to acknowledge an
interrupt.
    mov.l #loc,r5          ;Location for set up parameters.
    mov.b #0,r0
    mov.b r0,@r5           ;No change channel 0
    mov.b r0,@(2,r5)       ;No change channel 2
    mov.b r0,@(3,r5)       ;No change channel 3
    mov.b #IPACK,r0        ;Acknowledge interrupt.
    mov.b r0,@(1,r5)
;

```

```
; Issue a 'Set and read channel interrupts' BIOS call.
    mov.l  #CHISR,r4          ;BIOS command 0x900.
    dc.w  0x003b              ;BRK
    dc.w  0x000b              ;RTS
    dc.w  0x002b              ;RTE
    nop                       ;NOP
; After call:
; Error code returned in high word R0.
; 'Loc' contains eight 4-bit fields containing interrupt and buffer
; status.
```


Interrupt handler considerations

HUDI interrupt handler

The interrupt mechanism associated with channel usage generated by the DASH is the HUDI interrupt. This causes a HUDI interrupt category at an offset of 0x600 from the VBR. The application code must have a suitable exception handler routine in place to allow the exception to be serviced.

The HUDI interrupt has an associated priority to allow masking of interrupts by using the status register interrupt level mask bits (SR.IMASK).

For the SH3 this priority for the interrupt is fixed at 15.

For the SH4, this priority must be set so that the interrupt is not masked. The IPRC (interrupt priority register C) facilitates the setting up of the priority for the interrupt. The application must ensure that the priority for the interrupt is high enough to allow interrupts through. It is recommended that the IPRC value be set to its highest level (15) to ensure unplanned masking of this interrupt does not occur.

For the SH4 it is essential that the on-chip peripheral module interrupt priority level setting is only performed when the status register block bit SR.BL is set.

Within the application code interrupt handler, the INTEVT register must be checked to ascertain the source of the interrupt.

On SH4 INTEVT will contain 0x600.

On SH3 INTEVT will contain 0x5E0 and EXPEVT will contain 0x620.

Implementing BIOS calls within the interrupt handler

The recommended method for implementing an interrupt handler is:

1. Issue a CHISR BIOS call to ascertain which channel sources caused the interrupt (and whether it was on input and/or output).
2. For each of the interrupting sources, either:
 - Handle the interrupt immediately by issuing further BIOS channel commands to move data between the development computer and the target as applicable.
 - OR-
 - Set flags or signals to inform 'background code' of the interrupt occurring.
3. Issue a CHISR BIOS call to acknowledge all channel sources that caused the interrupt.

CPDIAL Library Reference

Host channel access

The CPDIAL communications library provides access to channels from the host. CPDIAL is organized as a C++ class called *CDial* containing objects that each hold an API for a different device supported by the library. There are four header files required to use the library with channels, the principle header file is *dial.h*.

Header file:	Contains:
Dial.h	The <i>CDial</i> class definition and definition of <i>DIALDEVICE</i> .
DialChannel.h	The definition of <i>CDialChannel</i> , the base class for the six channel-specific classes. Data structures used by <i>CDIALChannel</i> are contained in <i>GenChannel.h</i> .
GenChannel.h	The data structures used by <i>DialChannel</i> .
error.h	The DIAL API error code definitions.

Connecting and disconnecting

Two functions, *Connect* and *Disconnect* establish and release a connection between an application and a device. Depending on the type of the device, the nature of the connection can be very important. For example, a network device may be accessed by multiple users but an active connection provides a user with exclusive access to the device for the duration of that connection.

NOTE: *When a device is first found (using `FindNextDevice`) it is recommended that a connection is made and kept until the device is no longer required.*

Using CPDIAL

To initialize and use the library acquire a pointer to the library using *InitializeDial* function.

Calling a function

CPDIAL requires a ‘magic cookie’ (denoted by *DIAL_ID*) that the CPDIAL DLL uses to identify the device. The *DIAL_ID* is usually obtained using the *FindNextDevice* function.

Example

NOTE: *This example illustrates the calling convention only; for more examples, see the `TESTDIAL` diagnostic utility and source code provided with CPDIAL.*

```
// Instantiate the DIAL library once only for any given application
CDial * pDial = InitializeDial();
pDial = GetDial;

DIAL_EC ecRetCode = DIAL_EC_NOERROR;
DIAL_ID DeviceID = DIAL_ID_UNDEF;

ecRetCode = DIAL->FindNextDevice(    DeviceID,
                                   DIALDEVICE::DEVTYPE_KATANA_DA);

// a DIAL root method

if (ecRetCode == DIAL_EC_NOERROR)
{
    ecRetCode = pDial->DA.xxx(DeviceID, ...) // control the DA
    ecRetCode = pDIAL->Console.xxx(DIAL_ID, deviceID, ...) // control the console.
    ... // access the channels.
    ecRetCode = DIAL->FServer.Reserve (DeviceID, ...);
    ecRetCode = DIAL->FServer.Read(DeviceID, ...);
    ecRetCode = DIAL->FServer.Release(DeviceID, ...);
}
```

CDial functions

The *CDial* class contains these functions:

Function name	Purpose
InitializeDial	Initializes DIAL and returns a pointer to the Dial interface.
GetVersion	Returns the library's version number.
SetTimeout	Sets the timeout period.
GetTimeout	Returns the current timeout value.
FindNextDevice	Locate devices on the bus.
GetDeviceDetails	Supplies details of a specified device.
ValidateDevice	Tests the validity of a specified device.
GetErrorText	Converts an error code to a strings.
Connect	Makes a connection to a device.
Disconnect	Disconnects from a device.

Error codes

The majority of *CDial* class methods return a 32-bit error code (of type *DIAL_EC*). The higher 16 bits is a group code indicating the module, the lower 16 bits is an error code indicating the error. Error codes are defined in the file *error.h*; the construct of the error is not usually required to be known. Passing the error code to the *GetErrorText* function returns a textual description of the category and specific error encountered.

CDial class definition

The *CDial* class definition has the public form. The class *CChannel* is defined externally to *CDial* in its own definition and implementation files and contains just the routines specific to that module.

```
class CDial
{
public:
    enum DEBUGMODE
    {
        DEBUGMODE_OS,
        DEBUGMODE_CPU
    };
    CDialDAEx&      DA;
    CDialConsoleEx& Console;

    // Use these channel wrapper classes in preference to using
    // CChannel directly
    CTypedChannelEx& VSerial;
    CTypedChannelEx& DebugOS;
    CTypedChannelEx& FServer;
    CTypedChannelEx& Sound;
    CTypedChannelROEx& TraceProfile;
    // Direct access to the CChannel class.
    // Needs specification of channel identifier CHANTYPE enumerated
    // parameter defined in channel.h
    // N.B. CHANTYPE_TRACEPROFILE is *Read Only*, as specified in
    // CTraceProfile API.
    CDialChannelEx& Channel;

    CDial( CDialDAEx& da,
           CDialConsoleEx& console,
           CTypedChannelEx& vserial,
           CTypedChannelEx& debugOS,
           CTypedChannelEx& fileServer,
           CTypedChannelEx& sound,
           CTypedChannelROEx& traceProfile,
           CDialChannelEx& channel
         ) : DA( da ),
            Console( console ),
            VSerial( vserial ),
            DebugOS( debugOS ),
            FServer( fileServer ),
            Sound( sound ),
            TraceProfile( traceProfile ),
            Channel( channel )
    {
```

```

    }
    virtual BOOL      GetVersion( DWORD&, DWORD& ) const = 0;

    virtual void      SetTimeOut( DWORD ) = 0;
    virtual DWORD     GetTimeOut() const = 0;

    virtual DIAL_EC    FindNextDevice( DIAL_ID&,
                                       DIALDEVICE::DEVTYPE ) = 0;

    virtual DIAL_EC    GetDeviceDetails( DIAL_ID, DIALDEVICE& ) = 0;

    virtual DIAL_EC    ValidateDevice( DIAL_ID,
                                       DIALDEVICE::DEVTYPE = DIALDEVICE::DEVTYPE_UNDEF ) = 0;

    virtual const char*GetErrorText( DIAL_EC ) = 0;

    virtual DIAL_EC    Connect( DIAL_ID ) = 0;

    virtual void      Disconnect( DIAL_ID ) = 0;
};

#ifdef DIAL_DEF_
extern "C" __declspec( dllexport ) CDial * InitializeDial();
extern "C" __declspec( dllexport ) DIAL_EC GetDialDebugMode(CDial *,
                                                         DIAL_ID,
                                                         CDial::DEBUGMODE& );
extern "C" __declspec( dllexport ) DIAL_EC SetDialDebugMode(CDial *,
                                                         DIAL_ID,
                                                         CDial::DEBUGMODE );
#else
extern "C" __declspec( dllimport ) CDial * InitializeDial();
extern "C" __declspec( dllimport ) DIAL_EC GetDialDebugMode(CDial *,
                                                         DIAL_ID,
                                                         CDial::DEBUGMODE& );
extern "C" __declspec( dllimport ) DIAL_EC SetDialDebugMode(CDial *,
                                                         DIAL_ID,
                                                         CDial::DEBUGMODE );

typedef      CDialDAEx      CDA;
typedef      CDialConsoleEx CConsole;
typedef      CDialChannelEx CChannel;
typedef      CDialMirageEx  CMirage;
#endif

```

InitializeDial

Initializes CPDIAL and returns a pointer to the Dial interface.

```
CDial * InitializeDial();
```

Return value

A pointer to the Dial interface.

Parameters

None.

GetVersion

Returns the library's version number.

```
BOOL CDial::GetVersion (    WORD& Major,  
                           WORD& Minor);
```

Return value

Always returns *True*.

Parameters

Major

The major version number.

Minor

The minor version number.

SetTimeout

Sets the timeout, in milliseconds, for the high-level commands.

```
VOID SetTimeout(DWORD timeout);
```

Return value

None.

Parameters

timeout

The timeout value in milliseconds, the default is 5000 milliseconds.

Remarks

Sets the timeout value for the channel-specific commands. If a command reports a non-serious error, such as *BUSY* or *PENDING*, CPDIAL retries the command or waits for it to be completed until the timeout period is reached.

GetTimeOut

Returns the current timeout value for CPDIAL high-level commands.

```
DWORD GetTimeOut (VOID);
```

Return value

The current timeout value in milliseconds.

Parameters

None.

FindNextDevice

Locate devices on the bus.

```
DIAL_EC FindNextDevice(    DIAL_ID& device,  
                           DIALDEVICE::DEVTYPE type = DIALDEVICE::DEVTYPE_UNDEF);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

The most recently found device. Use *DIAL_ID_UNDEF* to find the first device.

type

The type of device to find. The default value is *ALL*. See *DIALDEVICE* in *dial.h* for a list of device types.

Remarks

Use *FindNextDevice* to find DIAL devices. The caller supplies a device identifier (or *DIAL_ID_UNDEF* to find the first device). On successful return the device identifier of the next matching device is set.

NOTE: *Device identifiers are passed by reference and will be modified.*

GetDeviceDetails

Supplies details for a specified device.

```
DIAL_EC GetDeviceDetails(    DIAL_ID device,  
                             DIALDEVICE& details);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

details

A *DIALDEVICE* structure where the results are stored.

Remarks

See *DIALDEVICE* in *dial.h* for supported device types.

NOTE: *The details of a particular device are passed by reference and will be modified.*

ValidateDevice

Tests the validity of a specified device.

```
DIAL_EC ValidateDevice(    DIAL_ID device,  
                           DIALDEVICE::DEVTYPE type = DIALDEVICE::DEVTYPE_UNDEF);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

The type of device to validate. The default is *DIALDEVICE::DEVTYPE_UNDEF*.

Remarks

See *DIALDEVICE* in *dial.h* for a list of device types.

GetErrorText

Converts an error code to a string for display purposes.

```
const char * GetErrorText(DIAL_EC error)
```

Return value

Returns a pointer to a string containing a textual representation of *error*.

Parameters

error

The error code to convert.

Remarks

None.

Connect

Makes a connection to a device.

```
DIAL_EC Connect(DIAL_ID device)
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

The identifier of the device to connect to.

Remarks

The device will not be available to other users until it is released using the *Disconnect* function. Each use of *Connect* must be matched by a corresponding *Disconnect* call.

NOTE: *Always use the Connect and Disconnect functions connecting to a device. This is the most efficient method and guarantees the correct state of the device.*

Disconnect

Disconnects from a device.

```
void Disconnect(DIAL_ID device)
```

Return value

None.

Parameters

device

The identifier of the device to disconnect from.

Remarks

After disconnecting, the device is then available to other users. Each use of the *Connect* function must be matched by a corresponding *Disconnect* call.

NOTE: *Always use the Connect and Disconnect functions connecting to a device. This is the most efficient method and guarantees the correct state of the device.*

CDial::CDialChannelEx functions

Typed channels

CPDIAL's channel functions are defined by three classes:

CDialChannelEx
CTypedChannelROEx
CTypedChannelEX

CDialChannelEx represents a generic channel, whereas *CTypedChannelROEx* and *CTypedChannelEX* represent channels of a particular type.

CDialChannelEx class definition

```
class CDialChannelEx : public CGenChannel
{
public:
    virtual DIAL_EC Reserve( DIAL_ID,
                             CHANTYPE,
                             DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
    virtual DIAL_EC Release( DIAL_ID, CHANTYPE ) = 0;
    virtual DIAL_EC Validate( DIAL_ID, CHANTYPE ) = 0;
    virtual DIAL_EC DataReady( DIAL_ID, CHANTYPE, BOOL&, DWORD& ) = 0;
    virtual DIAL_EC Read( DIAL_ID,
                          CHANTYPE,
                          DWORD,
                          DWORD&,
                          void*,
                          BOOL = FALSE,
                          DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
    virtual DIAL_EC Write( DIAL_ID,
                           CHANTYPE,
                           DWORD,
                           DWORD&,
                           const void*,
                           BOOL = FALSE,
                           DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```

Reserve

Reserves the channel specified by *type* and must be used before any reading or writing is performed.

```
DIAL_EC Reserve(          DIAL_ID device,
                          CHANTYPE type,
                          DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

timeout

The length of time to wait for the channel if it is in use by another process.

Remarks

When the channel is no longer required, for example when a program exits, a corresponding Release call must be made.

If the channel is currently in use, the length of time *Reserve* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

Release

Releases the channel specified by *type* on the device specified by *device* for use by other processes.

```
DIAL_EC Release(          DIAL_ID device,  
                        CHANTYPE type);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration specifying the channel.

Validate

Tests for the presence of the channel specified by *type* on the device specified by *device* and is reserved for use by this process.

```
DIAL_EC Validate(      DIAL_ID device,  
                      CHANTYPE type);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

Remarks

None.

DataReady

Determines whether there is any data to read on the specified channel.

```
DIAL_EC DataReady(    DIAL_ID device,
                      CHANTYPE type,
                      BOOL& isData,
                      DWORD& status);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

isData

Set to *True* if data is available in the buffer, *False* otherwise.

status

Holds additional status information on return.

Remarks

None.

Read

Reads a specified amount from the specified channel and returns the actual amount read.

```
DIAL_EC Read(          DIAL_ID device,
                      CHANTYPE type,
                      DWORD size,
                      DWORD& sizeRead,
                      const void* buffer,
                      BOOL blocking = FALSE,
                      DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

size

The amount of data to read.

sizeRead

The amount of data actually read.

buffer

A pointer to a buffer to store the result.

blocking

Set to *True* for blocking, *False* for non-blocking.

timeout

The delay before a timeout.

Remarks

By default *Read* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Read* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify a value of *INFINITE*.

Write

Writes from the specified channel to the specified address and returns the actual number of bytes read.

```
DIAL_EC Write(    DIAL_ID device,
                  CHANTYPE type,
                  DWORD size,
                  DWORD& sizeWritten,
                  const void* buffer,
                  BOOL blocking = FALSE,
                  DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

type

A member of the *CChannel::CHANTYPE* enumeration, specifying the channel.

size

The amount of data to write.

sizeWritten

The amount of data actually written.

buffer

A pointer to a buffer to store the result.

blocking

Set to *True* for blocking, *False* for non-blocking.

timeout

The delay before a timeout.

Remarks

By default *Write* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Write* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify a value of *INFINITE*.

CDial::CTypedChannelROEx functions

CTypedChannelROEx class definition

This represents a read-only channel of a specific type

```
class CTypedChannelROEx : public CGenChannel
{
public:
    virtual DIAL_EC Reserve(    DIAL_ID,
                                DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
    virtual DIAL_EC Release(    DIAL_ID ) = 0;
    virtual DIAL_EC Validate(    DIAL_ID ) = 0;
    virtual DIAL_EC DataReady(    DIAL_ID,
                                BOOL&,
                                DWORD& ) = 0;
    virtual DIAL_EC Read(        DIAL_ID,
                                DWORD,
                                DWORD&,
                                void*,
                                BOOL = FALSE,
                                DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```


Reserve

Reserves the channel specified by *type* and must be used before any reading or writing is performed.

```
DIAL_EC Reserve(          DIAL_ID device,  
                        DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

timeout

The length of time to wait for the channel if it is in use by another process.

Remarks

When the channel is no longer required, for example when a program exits, a corresponding Release call must be made.

If the channel is currently in use, the length of time *Reserve* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

Release

Releases the channel specified by `type` on the device specified by *device* for use by other processes.

```
DIAL_EC Release(          DIAL_ID device);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

Validate

Tests for the presence of the channel specified by *type* on the device specified by *device* and is reserved for use by this process.

```
DIAL_EC Validate(          DIAL_ID device);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

None.

DataReady

Determines whether there is any data to read on the specified channel.

```
DIAL_EC DataReady(    DIAL_ID device,  
                      BOOL& isData,  
                      DWORD& status);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

isData

Set to *True* if data is available in the buffer, *False* otherwise.

status

Holds additional status information on return.

Remarks

None.

Read

Reads a specified amount from the specified channel and returns the actual amount read.

```
DIAL_EC Read(          DIAL_ID device,
                      DWORD size,
                      DWORD& sizeRead,
                      const void* buffer,
                      BOOL blocking = FALSE,
                      DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

size

The amount of data to read.

sizeRead

The amount of data actually read.

buffer

A pointer to a buffer to store the result.

blocking

Set to *True* for blocking, *False* for non-blocking.

timeout

The delay before a timeout.

Remarks

By default *Read* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Read* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

CDial::CTypedChannelEx functions

CTypedChannelEx class definition

This represents a read-write channel of a specific type.

CTypedChannelEx inherits from *CTypedChannelROEx* and adds the *Write* function.

```
class CTypedChannelEx : public CTypedChannelROEx
{
public:
    virtual DIAL_EC Write(        DIAL_ID,
                                DWORD,
                                DWORD&,
                                const void*,
                                BOOL = FALSE,
                                DWORD = DIAL_DEFAULT_TIMEOUT ) = 0;
};
```

Write

Writes from the specified channel to the specified address and returns the actual number of bytes read.

```
DIAL_EC Write(    DIAL_ID device,
                  DWORD size,
                  DWORD& sizeWritten,
                  const void* buffer,
                  BOOL blocking = FALSE,
                  DWORD timeout = DIAL_DEFAULT_TIMEOUT);
```

Return value

Returns *DIAL_EC_NOERROR* on success or an error code otherwise.

Parameters

device

A valid device identifier.

size

The amount of data to write.

sizeWritten

The amount of data actually written.

buffer

A pointer to a buffer to store the result.

blocking

Set to *True* for blocking, *False* for non-blocking.

timeout

The delay before a timeout.

Remarks

By default *Write* is non-blocking. To make a blocking call use the extra parameters *blocking* and *timeout*. If blocking, the length of time *Write* waits for the channel request to succeed is specified by *DIAL_DEFAULT_TIMEOUT* which is set to 5000 milliseconds by default. To wait indefinitely, specify *INFINITE*.

CDial::Console functions

CDial::Console class definition

These functions are used to communicate with the target attached to the DASH and are defined in `DialConsole.h`. The `CDial::Console` commands (with the exception of `Inquiry`) halt the target when issued, it is your responsibility to determine whether the target was running before the command was issued and to resume after execution of the command, see *“Inquiry” on page 568*.

```
class CDialConsole : public CGenConsole
{
public:
    virtual DIAL_EC Inquiry(    DIAL_ID,
                               INQUIRY& ) = 0;
    virtual DIAL_EC ProcessInquiryError( const INQUIRY&,
                                         BOOL = TRUE,
                                         BOOL = FALSE ) = 0;

    virtual DIAL_EC Execute( DIAL_ID, DWORD ) = 0;
    virtual DIAL_EC Suspend( DIAL_ID ) = 0;
    virtual DIAL_EC Resume( DIAL_ID ) = 0;
    virtual DIAL_EC ReadMemory(    DIAL_ID,
                                   DWORD,
                                   ELEMENTSIZE,
                                   DWORD,
                                   void* ) = 0;
    virtual DIAL_EC WriteMemory(   DIAL_ID,
                                   DWORD,
                                   ELEMENTSIZE,
                                   DWORD,
                                   const void* ) = 0;
    virtual DIAL_EC ReadContext(   DIAL_ID,
                                   CONTEXTMODE,
                                   WORD,
                                   void* ) = 0;
    virtual DIAL_EC WriteContext(  DIAL_ID,
                                   CONTEXTMODE,
                                   WORD,
                                   const void* ) = 0;
    virtual DIAL_EC ReadConfig(    DIAL_ID,
                                   READCONFIG& ) = 0;
    virtual DIAL_EC ResetNoDebug(  DIAL_ID ) = 0;
    virtual DIAL_EC ResetAndDebug( DIAL_ID ) = 0;
    virtual DIAL_EC MakeSafe(      DIAL_ID ) = 0;
    virtual DIAL_EC GetValidConsoleStatus( DIAL_ID,
                                           CDialConsoleStatus *& ) = 0;
};
```



```
#ifndef CPL_INTERNAL
class CDialConsoleEx : public CDialConsole {};
#endif
```

Inquiry

Returns the current state of the target.

```
DIAL_EC Inquiry(          DIAL_ID device,
                        INQUIRY& inquiry);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

inquiry

A `CConsole::INQUIRY` structure to store the result.

Remarks

The status information returned is state dependent. Within the `CConsole::INQUIRY` class returned, three fields are used to identify the current execution state of the target, the signal category and a signal identifier. The fields relate to enumerations within the `CConsole` class such as `CConsole::SIGNAL`, `CConsole::SIGCAT`.

Signal state	Execution state
<code>SIGNAL_RUNNING</code>	Executing application code.
<code>SIGNAL_STOPPED</code>	In debug stub as result of a <code>Suspend ()</code> target command.
<code>SIGNAL_HALTED</code>	In debug stub as result of CPU exception <i>or</i> target command issued. Check <code>bySigCat</code> and <code>wSigID</code> to differentiate.
<code>SIGNAL_RESET</code>	Initial condition after <code>CConsole::ResetAndDebug ()</code> or manual reset or power cycle causes the stub to (re)load when the DASH is in CPU debug mode.
<code>SIGNAL_SAFE</code>	Indicates a <code>CConsole::MakeSafe ()</code> command was issued.
<code>SIGNAL_FAIL</code>	Failed to load/reset the debug stub.
<code>SIGNAL_NODEBUG</code>	Running without target debug facilities.

When a signal value of either `SIGNAL_STOPPED`, `SIGNAL_HALTED`, `SIGNAL_RESET` or `SIGNAL_SAFE` is flagged, the current PC field is also available and valid.

When a signal value of `SIGNAL_HALTED` is flagged, the signal category and signal ID fields provide processor specific information for the exception or other reason for being in the debug stub.

Possible values of the signal category where `Inquiry` returns `SIGNAL_HALTED`.

Signal category	Means
<code>SIGCAT_UNDEFINED</code>	Reserved for future use.
<code>SIGCAT_ASE</code>	Currently defined for use on the 7091-EVA.
<code>SIGCAT_GENERAL</code>	General exception.
<code>SIGCAT_INTERRUPT</code>	Interrupt.
<code>SIGCAT_CHNLBLOCK</code>	Blocked channel (such as Fserver) operation in progress.
<code>SIGCAT_REPORTEVENT</code>	Check <code>SIGID</code> for event being reported.
<code>SIGCAT_PROFILEBUFF</code>	Statistical sampling and trace profiling buffer signal.

The `SIGCAT_REPORTEVENT` is currently only used to report that a CPU manual reset has been detected. The signal ID will be `SIGID_MANUALRESET`.

The `SIGCAT_CHNLBLOCK` category can be treated in the same way as `SIGNAL_RUNNING`, and target commands may be issued as normal. However, should the `CConsole::Resume` command be issued, the blocked serial channel operation will return `False` if it has not yet completed. The response to this signal category is to wait for the `CHNLBLOCK` signal category to be cleared. For example, `CConsole::Inquiry` will continue to be issued until the signal changes and a signal value such as `SIGNAL_RUNNING` is detected.

The signal ID value is specific to the CPU and matches the manufacturer's exception code for the given category. In general, the 'vector base' and 'offset' are encoded into 'category', then 'exception code' is the category specific ID.

To determine the reason for halting the target execution and entering the debug stub, the `SIGNAL_HALTED` state and signal category `SIGCAT_ASE` are both specified. The two causes of this are:

- A `CConsole` command was issued by the host.
- A software breakpoint (using ASE BRK) or hardware breakpoint (using ASE HBC) was encountered.

The signal identifier allows you to differentiate between ASE exceptions:

```
if (InquirySignal == CConsole::SIGNAL_HALTED)
{
    switch (InquirySignalCategory)
    {
        case CConsole::SIGCAT_ASE:
```

```
switch (InquirySignalID)
{
case 0x04: // ASE-HBC BREAK
    // HBC breakpoint encountered break;
case 0x08: // ASE-SW BREAK
    // BRK software breakpoint encountered break;
case 0x10: // ASE-PIN BREAK
    // ASE Pin break encountered as a result of CConsolecommand being issued
    // (Inquiry() command does not cause ASE-PIN BREAK entry into the stub).
    // This means target was executing game code and would have reported
    // a SIGNAL_RUNNING if we hadn't issued a command to it. After all debug
    // commands have been issued, a CConsole::Resume() command should be issued
    // to exit the debug stub and continue normal execution
    break;

default:
    break;
}

break;
...
...
default:
    ...
    break;
}
}
```

NOTE: *The CConsole::Inquiry() command does not in itself cause a SIGNAL_HALTED and ASE-PIN BREAK exception to be reported, even though it is described as a CConsole command; it is able to non-intrusively report the current status.*

The following example code checks to see if the target is running before executing a Console command and subsequently restarts the target:

```
CGenConsole::INQUIRYinquiry;

//          Get current execution status

if( DIAL_EC_NOERROR == pDial->Console.Inquiry( dialID, inquiry ) )
{
    // Now perform required console commands

    pDial->Console.ReadMemory( dialID, ... );
    pDial->Console.WriteMemory( dialID, ... );

    // When complete restart if target was previously running
```

```
        if( CGenConsole::SIGNAL_RUNNING == inquiry.bySigNum )
        {
            pDial->Console.Resume( dialID );
        }
    }
```

ProcessInquiryError

Queries the `inquiry` data bit-fields retrieved from the `Inquiry` command.

```
CDail::ProcessInquiryError(      const INQUIRY& inquiry,  
                                BOOL needDebug = TRUE,  
                                BOOL reserved = FALSE);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

inquiry

A `CConsole::INQUIRY` structure obtained using the `Inquiry` function.

needDebug

Set to `True` if debug support is required, `False` otherwise.

reserved

Reserved. Always set to `False`.

Remarks

Use this function to validate whether the target is in a state to support debugging using the `CConsole` commands for example, all stubs have been successfully loaded with debug support available. Specify the required level of functionality using the `NeedDebug` boolean parameter.

Execute

Causes execution to start on the target from the given address.

```
DIAL_EC Execute(          DIAL_ID device,  
                    DWORD address);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address

The address to start executing from.

Remarks

This function is equivalent to modifying the PC register using read and write context and then issuing a resume command.

After a `CConsole::Execute()` command has been successfully issued, the `CConsole::Inquiry()` command reports the signal status as `SIGNAL_RUNNING` unless an exception is encountered prior to completion. In this case the relevant signal, category and code is reported.

Suspend

This causes the target specified by `device` to suspend execution and enter the debug stub.

```
DIAL_EC Suspend(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_STOPPED` after a `CConsole::Suspend()` command has been successfully issued.

Resume

Causes execution on the target to resume from the point where it stopped.

```
DIAL_EC Resume(DIAL_ID deviceID);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

Use this function to resume execution when the target has been stopped due to a suspend command, exception or general host command interaction such as the current PC. After a `CConsole::Resume()` command has been successfully issued, the `CConsole::Inquiry()` command reports the signal status as `SIGNAL_RUNNING`, unless an exception is encountered prior to completion. In this case the relevant signal, category and code will be reported.

ReadMemory

Reads the target's memory.

```
DIAL_EC ReadMemory(          DIAL_ID device,
                             DWORD address,
                             ELEMENTSIZE elementSize,
                             DWORD elementCount,
                             void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address

The address to read from.

elementSize

A member of the `CConsole::ELEMENTSIZE` enumeration (`ELEMENTSIZE_BYTE`, `ELEMENTSIZE_WORD`, or `ELEMENTSIZE_LONG`) specifying the size of each element.

elementCount

The number of elements to read.

buffer

A pointer to a buffer large enough to hold the resulting data.

Remarks

NOTE: *The memory is retrieved into the specified destination buffer as a byte stream in the same order as in the target's memory, regardless of the `elementSize` specified.*

WriteMemory

Writes to the target's memory.

```
DIAL_EC WriteMemory(    DIAL_ID device,
                        DWORD address,
                        ELEMENTSIZE elementSize,
                        DWORD elementCount,
                        const void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address

The address to write to.

elementSize

A member of the `CConsole::ELEMENTSIZE` enumeration (`ELEMENTSIZE_BYTE`, `ELEMENTSIZE_WORD`, or `ELEMENTSIZE_LONG`) specifying the size of each element.

elementCount

The number of elements to write.

buffer

A pointer to a buffer holding the data to be written.

NOTE: *The memory is written from the specified source buffer as a byte stream as ordered in the target's memory, regardless of the `ElementSize` specified.*

ReadContext

Reads a context from the target.

```
DIAL_EC ReadContext(    DIAL_ID device,
                        CONTEXTMODE mode,
                        WORD length,
                        void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

mode

A member of the `CConsole::CONTEXTMODE` enumeration that specifies which set of internal registers to read.

length

The size of the buffer to store the result.

buffer

A pointer to a buffer to store the result.

Remarks

The `CConsole::CONTEXTMODE` is defined as an enumeration as follows:

Context	Means
<code>CONTEXTMODE_GENERAL</code>	General registers.
<code>CONTEXTMODE_FPU</code>	Floating point registers.
<code>CONTEXTMODE_UBC</code>	User Break Controller.
<code>CONTEXTMODE_HBC</code>	Reserved - EVA chip's Hardware Break Controller.
<code>CONTEXTMODE_ASE</code>	Reserved - ASE mode specific registers.
<code>CONTEXTMODE_PERF</code>	Reserved for Performance counters.
<code>CONTEXTMODE_TRACE</code>	Reserved for Execution Trace information.
<code>CONTEXTMODE_MMU</code>	Reserved for the Memory management unit.

The structure definition for the general and floating point registers is defined in `GenConsole.h` as the structures `SDIALRegsSH4EVA` and `SDIALRegsSH4EVA_FPU`.

A pointer to one of these structures is passed as the destination buffer to the function and the structure size, `length`, is specified using `sizeof(structure)`.

NOTE: *The context is declared to be represented in the native endian of the processor. The endianness can be checked using the `CConsole::ReadConfig` command.*

WriteContext

Writes a context to the target.

```
DIAL_EC WriteContext(    DIAL_ID device,
                        CConsole::CONTEXTMODE mode,
                        WORD length,
                        const void* buffer);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

mode

A member of the `CConsole::CONTEXTMODE` enumeration that specifies which set of internal registers to read.

length

The size of the buffer storing the data to write.

buffer

A pointer to a buffer storing the data to write.

Remarks

The `CConsole::CONTEXTMODE` is defined as an enumeration as follows:

Mode	Context
<code>CONTEXTMODE_GENERAL</code>	General registers.
<code>CONTEXTMODE_FPU</code>	Floating point registers.
<code>CONTEXTMODE_UBC</code>	User Break Controller.
<code>CONTEXTMODE_HBC</code>	Reserved. Hardware Break Controller.
<code>CONTEXTMODE_ASE</code>	Reserved. ASE mode specific registers.
<code>CONTEXTMODE_PERF</code>	Reserved. Performance counters
<code>CONTEXTMODE_TRACE</code>	Reserved. Execution Trace information
<code>CONTEXTMODE_MMU</code>	Reserved. Memory management unit

The structure definition for the general and floating point registers is defined in `GenConsole.h` as the structures `SDIALRegsSH4EVA` and `SDIALRegsSH4EVA_FPU`.

A pointer to one of these structures is passed as the source buffer to the function. The structure size is specified as the `length` using `sizeof(structure)`.

NOTE: *The context is declared to be represented in the native endian of the processor. The endianness can be checked using the `CConsole::ReadConfig` command.*

ReadConfig

Reads the current configuration of the debug stub on the target.

```
DIAL_EC ReadConfig(          DIAL_ID device,
                             READCONFIG& config);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

config

A `CConsole::READCONFIG` structure to hold the result.

Remarks

The configuration details retrieved in `CConsole::READCONFIG` include the endian of the target CPU, the CPU family and type. The Hitachi SuperH family is identified as an enumeration `UPROCFAMILY_HI_SH` and the processor type within that family is defined to be one of the following:

- `UPROCTYPE_UNDEF` (Reserved)
- `UPROCTYPE_SH1`
- `UPROCTYPE_SH2`
- `UPROCTYPE_SH3`
- `UPROCTYPE_SH3E`
- `UPROCTYPE_SH4`
- `UPROCTYPE_SH4EVA`

ResetNoDebug

Resets the main board without loading a debug stub and limits the available CDial::Console commands to Inquiry and ResetAndDebug. All CDial::DA commands are still available.

```
DIAL_EC ResetNoDebug(DIAL_ID device);
```

Return Value

Returns DIAL_EC_NOERROR on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

Use this function to test applications without interference from the debugging system as if the application was running on a production unit.

ResetAndDebug

Causes a full reset of the system and the 1K ASERAM stub to be loaded onto the target and run. Full debugging functionality is available on the target.

```
DIAL_EC ResetAndDebug(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

The reset behavior is dependent on a DASH configuration status bit, `OSorCPUflag`, defined in `CDA::CONFIGDATA`. A value of 0 means OS debug mode and value of 1 means CPU debug mode.

When the DASH is configured for OS debug mode, the target will automatically exit the debug stub and resume execution after the reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RUNNING` after `CConsole::ResetAndDebug()` has been successfully issued.

When the DASH is configured for CPU debug mode, the target will remain in the debug stub after the reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RESET` after `CConsole::ResetAndDebug()` has been successfully issued and execution of the target will not occur until either a `CConsole::Resume()` or a `CConsole::Execute()` command is issued.

MakeSafe

```
DIAL_EC MakeSafe(DIAL_ID device);
```

Allows the DASH to set the target microprocessor to a safe state by setting various registers.

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

The following operations are carried out as a result of a `MakeSafe` command being issued:

- The DBR register is loaded with the default debug stub exception handler (SH4).
- The VBR register is loaded with the default debug stub exception handler.
- The stack pointer is loaded with the default SP (or 0x0d000000 on SH4).
- The status register block bit is cleared to 0 (value 0x0600000F0 on SH4).

The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_SAFE` after a `CConsole::MakeSafe()` command has been successfully issued.

CDial::DA functions

CDial::DA Class definition

These functions control the DASH and are defined in DialDA.h.

```
class CDialDA : public CGenDA
{
public:
    virtual DIAL_EC ReadProductID(    DIAL_ID,
                                      PRODUCTID& ) = 0;
    virtual DIAL_EC ReadStubLoadAddr( DIAL_ID,
                                      DWORD& ) = 0;
    virtual DIAL_EC ReadVersion(    DIAL_ID,
                                    VERSION& ) = 0;
    virtual DIAL_EC Reset( DIAL_ID ) = 0;
    virtual DIAL_EC ReadConfig(    DIAL_ID,
                                    CONFIGDATA& ) = 0;
    virtual DIAL_EC WriteConfig(    DIAL_ID,
                                    const CONFIGDATA& ) = 0;
    virtual DIAL_EC RequestLoadFullStub( DIAL_ID,
                                          DWORD ) = 0;
    virtual DIAL_EC ClearPendingResponse( DIAL_ID ) = 0;
};

#ifdef CPL_INTERNAL
class CDialDAEx : public CDialDA {};
#endif
```

ReadProductID

Returns the product identifier, model identifier, serial number and test date of the DASH.

```
DIAL_EC ReadProductID(    DIAL_ID device,  
                          PRODUCTID& ProductID);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

ProductID

A `PRODUCTID` structure to store the result.

Remarks

The information returned by `ProductID` is stored in the `CDA::PRODUCTID` structure as zero terminated strings.

ReadStubLoadAddr

Returns the address where the debug stub was loaded.

```
DIAL_EC ReadStubLoadAddr(    DIAL_ID device,  
                             DWORD& address);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address

A `DWORD` to hold the resultant address.

Remarks

If the debug stub was not loaded the address returned will be zero.

NOTE: *It is also possible to determine whether or not the stub was loaded (or failed to load) by calling the `CConsole::ProcessInquiryError()` function and specifying debug support required as a parameter.*

ReadVersion

Returns the version information for the DASH.

```
DIAL_EC ReadVersion(      DIAL_ID device,
                          VERSION& version);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

version

A `VERSION` structure to store the result.

Remarks

The version details are the DASH firmware's major and minor version numbers, a zero terminated revision character string, and two zero terminated character strings for the Date and Time of the build.

Reset

Resets the DASH.

```
DIAL_EC Reset( DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

The DASH is reset by enabling its watchdog timer with a short timeout. The watchdog is not subsequently updated by the DASH and so the DASH is internally reset. This command is reserved for use in hardware debug purposes only and should not generally be used, as there are preferred functions to specifically reset the target, see “*CDial::Console functions*” on page 566.

ReadConfig

Read the DASH's configuration.

```
DIAL_EC ReadConfig(          DIAL_ID device,  
                           CONFIGDATA& config);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

config

A `CONFIGDATA` structure to store the result.

Remarks

None.

WriteConfig

Set the configuration of the DASH.

```
DIAL_EC WriteConfig(          DIAL_ID device,
                             const CONFIGDATA& config);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier

config

A `CONFIGDATA` structure to write to the DASH.

Remarks

DebugMode

The target's reset behavior is dependent on a DASH configuration status bit, `DebugMode`, defined in `CDA::CONFIGDATA`. A value of `CDA::DACONFDATA::DEBUGMODE_OS` means OS debug mode and `CDA::DACONFDATA::DEBUGMODE_CPU` means CPU debug mode. This bit is retrieved and modified using the `CDA::ReadConfig()` and `CDA::WriteConfig()` commands.

When the DASH is configured for OS debug mode, the target will automatically exit the debug stub and resume execution after a `CConsole::ResetAndDebug()` or manual reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RUNNING` after a reset has been successfully processed.

When the DASH is configured for CPU debug mode, the target will remain in the debug stub after the reset has occurred. The `CConsole::Inquiry()` command will report the signal status as `SIGNAL_RESET` after `CConsole::ResetAndDebug()` has been successfully issued and execution of the target will not occur until either a `CConsole::Resume()` or a `CConsole::Execute()` command is issued.

StubCache

When the DASH is configured for CPU debug mode, the debug stubs can be configured to run with the cache disabled or enabled. A value of `CDA::DACONFDATA::STUBCACHE_ON` means the instruction cache and the operand cache are enabled and the operand cache is configured for 'write through'. A Value of `CDA::DACONFDATA::STUBCACHE_OFF` means the debug stub is not cached.

RequestLoadFullStub

Loads the extended debug stub.

```
DIAL_ID RequestLoadFullStub( DIAL_ID device,  
                             DWORD address);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

address

The address to load the stub.

Remarks

None.

ClearPendingResponse

Re-synchronize operations between the host and the DASH.

```
DIAL_EC ClearPendingResponse(DIAL_ID device);
```

Return Value

Returns `DIAL_EC_NOERROR` on success or an error code otherwise.

Parameters

device

A valid device identifier.

Remarks

Due to the nature of the protocol between the host and the DASH, a command request is issued to the DASH, and a response is expected to be retrieved from the DASH. If this reply is not retrieved by the host due to a timeout or other error, the DASH may be left in the state where it will not respond to further commands until such time as the reply packet is retrieved by the host. This function performs a retrieve and discard reply packet operation, which is normally able to free up the DASH to accept further commands.

CPDIAL error codes

All error codes are defined as enumerations in *error.h*. The higher order 16 bits contain the error category and the lower order 16 bits contain the actual error code.

Error categories

Error Code	Means
ERRCAT_NOERROR	No error.
ERRCAT_SALSA	General errors.
ERRCAT_SALSADACON	Additional errors relating to DASH and target.
ERRCAT_DA	Errors from DASH specific commands.
ERRCAT_CON	Errors from target specific commands.
ERRCAT_WINSOCK	Errors Winsock specific commands.
ERRCAT_UNKNOWN	Errors of unknown origin.

SALSA error codes

Error Code	Means
ERRSALSA_NOERROR	No error.
ERRSALSA_NOTINITED	Call made to uninitialized layer.
ERRSALSA_DEVICENOTMATCH	Failed to validate the device type.
ERRSALSA_DEVICENOTFOUND	Didn't find requested device.
ERRSALSA_CHANLNOTALLOCD	Logical channel (LUN) not allocated.
ERRSALSA_NOCHNLAVAIL	No more channels (LUNs) available.
ERRSALSA_MEMALLOCFAIL	Memory allocation failed.
ERRSALSA_BADADAPTER	Failed to validate the DASH.
ERRSALSA_BADDEVICEID	Failed to validate the device identifier.
ERRSALSA_TIMEOUT	Timed out on command.
ERRSALSA_FAILCREATESYNCOBJ	Failed to create the device's synchronization object.
ERRSALSA_FAILCREATELOCKOBJ	Failed to create the device's synchronization control object.
ERRSALSA_FAILLOCK	Timed out or bad attempt to lock a synchronization object.
ERRSALSA_FAILUNLOCK	Failed attempting to unlock a synchronization object.
ERRSALSA_BADVERSION	The version passed to <i>SALSA.Initialize</i> does not match the version used to build the library.
ERRSALSA_DEVICEINUSE	The specified device is already in use.
ERRSALSA_NOTIMPLEMENTED	The specified function has not been implemented.

SALSADACON error codes

Error Code	Means
ERRSALSADACON_CMDPENDING	Command still pending completion.
ERRSALSADACON_CMDNOMATCH	Returned command header did not match in command field.
ERRSALSADACON_SEQNOMATCH	Returned command header did not match in sequence field.
ERRSALSADACON_FLSHOPENFAIL	Failed to open flash image file.
ERRSALSADACON_FLSHSTATFAIL	Failed to <i>fstat()</i> flash image file.
ERRSALSADACON_FLSHSHORTREAD	Short read from flash image file.
ERRSALSADACON_BADELEMENTSIZE	Bad <i>ELEMENTSIZE</i> specified in <i>ReadMemory()</i> or <i>WriteMemory()</i> .
ERRSALSADACON_NOCONPOWER	The target is currently powered off.
ERRSALSADACON_RESETEINGCON	The target is currently being reset (Reset pin low).
ERRSALSADACON_FAILLOADSTUB (SIGNAL_FAIL)	One of the debug stubs failed to load.
ERRSALSADACON_FAIL1KSTUB	The ASE RAM debug stub failed to load.
ERRSALSADACON_FAILFULLSTUB	The Extended debug stub failed to load.
ERRSALSADACON_FAILNODEBUG (SIGNAL_NODEBUG)	No debug support available.

DA error codes

Error Code	Means
DAERRDA_DAAOK	Command completed without an error.
DAERRDA_DATMO DA	Command timed out during action.
DAERRDA_DAERA DA	Command error while erasing firmware.
DAERRDA_DAPRG	DA command error during reflashing.
DAERRDA_DAFIC	DA Command firmware image corrupt.
DAERRDA_DAERR	DA Command general error.

Console error codes

Error Code	Means
DAERRCON_CONAOK	Console command completed without an error.
DAERRCON_CONERR	Console command fatal error.
DAERRCON_CONBAD	Console command unknown.
DAERRCON_CONPRM	Console command parameter error.
DAERRCON_CONADR	Console command bad address.
DAERRCON_CONCNT	Console command bad count.
DAERRCON_CONCBF	Console command channel buffer full.
DAERRCON_CONCBE	Console command channel buffer empty.
DAERRCON_CONBSY	Console command not processed as BUSY.
DAERRCON_CONCNA	Console command not available (no debug stub).

WINSOCK error codes

Error Code	Means
ERRWINSOCK_INVALIDVERSION	The version of the Winsock library is invalid.
ERRWINSOCK_CANTCONNECT	Unable to connect to the specified device.

UNKNOWN error codes

Error Code	Means
ERRUNKNOWN_UNKNOWNERR	Error of unknown origin.

