

SourcePoint 7.7.1

Table of Contents

Using Help	1
SourcePoint Help Overview	1
Welcome to Help	1
About SourcePoint Help User's Guide	2
How To - HTML Help	3
How to Use SourcePoint Help.....	3
How to Find a Help Topic	4
How to Change the Window Size.....	5
How to Copy a Help Topic.....	6
How to Print a Help Topic.....	7
How to Make Relative Moves.....	8
How to Change Fonts.....	9
How to Change Colors	10
How to Hide or Show Help Contents.....	11
Arium	13
What's New in SourcePoint 7.7.1	13
Contacting Arium	15
Contacting Technical Support.....	16
Installation.....	17
Installation Overview	17
Host System Requirements.....	17
Installing and Configuring the ECM-HDT	18
Installing and Configuring the ECM-700	21
Installing and Configuring the ECM-XDP	24

Installing and Configuring the ECM-50	28
Macrovision FLEXIm® Licensing	31
Command Line Key Words	32
How To - Installation	33
How to Update Flash Code	33
SourcePoint Environment.....	35
SourcePoint Overview	35
SourcePoint Parent Window Introduction	35
SourcePoint Icon Toolbar.....	38
File Menu.....	40
File Menu - Project Menu Item	41
File Menu - Layout Menu Item.....	42
File Menu - Program Menu Item.....	43
File Menu - Macro Menu Item	48
File Menu - Print Menu Items	51
File Menu - Update Emulator Flash Menu Item	53
Program Target Devices Menu Item	54
File Menu - Other Menu Items.....	55
Edit Menu	57
View Menu.....	59
Processor Menu	62
Options Menu	63
Options Menu - Preferences Menu Item	64
Options Menu - Target Configuration Menu Item.....	75
Options Menu - Load Target Configuration File Menu Item.....	83

Option Menu - Save Target Configuration File Menu Item	84
Options Menu - Emulator Configuration Menu Item.....	85
Options Menu - Emulator Connection Menu Item	94
Options Menu - Emulator Reset Menu Item.....	96
Options Menu - Confidence Tests Menu Item.....	97
Window Menu.....	98
Help Menu	99
How To -- SourcePoint Environment	100
How to Add Emulator Connections	100
How to Configure Custom Macro Icons	105
How to Configure Autoloading Macros.....	107
How to Display Text on the Icon Toolbar	108
How to Edit Icon Groups to Customize Your Toolbars	109
How to Hot Plug an Arium Emulator	110
How to Modify a Defined Memory Region.....	111
How to Refresh SourcePoint Windows	113
How to Save a Program	114
How to Use the EFI Address Finder.....	115
How to Use the New Project Wizard	117
How to Verify Emulator Network Connections	121
Breakpoints Window	123
Breakpoints Window Overview	123
Breakpoints Window Introduction.....	123
Breakpoints Window Icon Definitions.....	126
Breakpoints Menu	127

Breakpoint Types	129
Edit Breakpoint and Add Breakpoint Dialog Boxes	133
Breakpoints Window Preferences	137
How To - Breakpoints	138
How to Disable and Enable Breakpoints in the Breakpoints Window	138
How to Remove Breakpoints.....	139
How to Set a Breakpoint on an Interrupt	140
How to Set a Bus Breakpoint	142
How to Set an Emulator Breakpoint	143
How to Set a Processor (Debug Register) Breakpoint.....	144
How to Set a Software Breakpoint	145
Code Window	147
Code Window Overview.....	147
Code Window Introduction	147
Code Window Icon Definitions	149
Code Window Menu	150
Code Window Preferences.....	154
How To - Code Window	155
How to Open a Code Window	155
How to Disassemble Code at a Specific Location.....	156
How to Save Code Window Settings	157
How to Save Code Window Contents	158
Command Window	159
Command Window Overview.....	159
Command Window Introduction	159

Command Syntax and Conventions.....	160
Command Window Menu	161
How To - Command Window	162
How to Abort a Command.....	162
How to Display and Delete Procs.....	163
How to Edit a Macro From a Command Window	164
How to Load Macros and Procs From a Command Window	165
How to Use REXX to Automate and Control SourcePoint	166
How to Use the Wait Command.....	169
Confidence Tests Window	171
Confidence Tests Window Overview	171
Confidence Tests Window Introduction.....	171
Confidence Tests Tabs	174
Table of Confidence Test Failures and Symptoms	177
Descriptors Tables Window.....	179
Descriptors Tables Window Overview	179
Descriptors Window Introduction.....	179
Descriptors Window Menu	181
How To - Descriptors	183
How to Replace a Descriptor Entry	183
Devices Window	185
Devices Window Overview	185
Devices Window Introduction.....	185
Devices Window Menu.....	194
Accessing Devices Window Cells in the Command Window	196

How To - Devices Window	198
How to Create Simple Devices Windows	198
Log Window	201
Log Window Overview	201
Log Window Introduction	201
Log Window Icon Definitions	203
Log Window Menu	204
Memory Window	207
Memory Window Overview	207
Memory Window Introduction	207
Memory Window Menu	209
Memory Window Preferences	211
How To - Memory Window	212
How to Open a Memory Window	212
How to View Memory at an Address	213
How To Change Memory Values	214
Operating System Window	215
Operating System Windows Overview	215
Linux OS-Aware Debugging	215
How To - Operating System Windows	223
How to Set Breakpoints in Linux From the Breakpoints Window	223
Page Translation Window	225
Page Translation Windows Overview	225
Page Translation Window Introduction	225
PCI Devices Window	227

PCI Devices Window Overview	227
PCI Devices Window Introduction	227
PCI Devices Window Menu	231
How To - PCI Devices Window	232
How to Open the PCI Registers View From the PCI Devices Window	232
How to Refresh a PCI Devices Dialog Box	233
Registers Window	235
Registers Window Overview	235
Registers Window Introduction	235
Register Window Menu	239
How To - Registers	242
How to Change Binary Values	242
How to Change Hexadecimal Values	243
Symbols Windows	245
Symbols Window Overview	245
Symbols Window Introduction	245
Symbols Window Icon Definitions	248
Symbols Window Menus	249
Classes Tab	251
Globals Tab	252
Locals Tab	254
Stack Tab	255
How To - Symbols Window	256
How to Change Values in the Symbols Window	256
Trace Window	257

Trace Window Overview	257
Trace Window Introduction.....	257
Trace Configuration.....	261
Trace Display Settings	263
How To - Trace Window	265
How to Print Trace.....	265
How to Save Trace.....	266
Viewpoint Window	267
Viewpoint Window Overview.....	267
Viewpoint Window Introduction	267
Linux Task Debugging.....	268
Viewpoint Window Menu	269
Disabling Processors.....	271
Watch Window.....	273
Watch Window Overview	273
Watch Window Introduction.....	273
Watch Window Menu	276
How To - Watch Window	278
How to Add and Expand Registers in a Watch View	278
How to Add Symbols to a Watch or Quick Watch View	280
Technical Notes	281
Descriptor Cache: Revealing Hidden Registers	281
UEFI Framework Debugging	283
Overview.....	283
UEFI Macros.....	283

PEI Debugging	283
DXE Debugging.....	285
HOBs.....	286
System Configuration Table	287
UEFI System Memory Map	287
Notes	287
Memory Casting.....	289
Defining Debug Variables of a Symbol Type as Defined in a Loaded Program	289
Casting Blocks of Target Memory as a Symbol Type as Defined in a Loaded Program	289
Microsoft® PE Format Support in SourcePoint	290
Overview.....	290
FAQs	291
Known restrictions of PE/PDB support in SourcePoint	292
Multi-Clustering	293
Hardware Setup	293
Software Setup.....	294
Running in Multi-Cluster Mode	294
Timing.....	295
Registers Keyword Table.....	296
Stepping	299
Strategies for Source Level Stepping.....	299
Symbolic Text Format (Textsym)	302
File Format	302
Example.....	303
Using Bookmarks	304

Adding/Removing Bookmarks	304
Navigating Bookmarks	304
Clearing Bookmarks	304
Bookmark Indications	304
Which Processor Is Which	306
Introduction	306
What Does "Last on the Chain, First on the Chain" Mean?	306
How Is This Related to the PROCESSORCONTROL Variable in SourcePoint?	307
What Does It Mean to Control More Than One Processor?	307
SourcePoint Commands	309
Command-Related Topics	309
Array Data Type	309
Character Strings	311
Control Register Bit Maps and Names	312
Control Constructs	314
Control Variables	315
Data Types	317
Allowable Operation by Type	319
Debug Procedures	320
Debug Pointer Types	324
Debug Variables	325
Expressions	326
Functions	329
Macros and Procs	331
Macro and Proc Limitations	332

Memory Access	333
Operand Character Delimiters.....	338
Operators.....	339
Real Numbers.....	341
Register Group Commands.....	342
Sub-Expressions	343
Symbolic References	345
Type Conversions	350
Types and Type Classes.....	351
Commands.....	352
#define Command	352
#undef Command.....	353
csr.....	354
Aadump command	356
Abort Command	357
Abs Command.....	358
Acos Command	359
Asin Command	360
Asm Command.....	361
Asmmode Control Variable	372
Atan Command	373
Atan2 Command	374
Base Control Variable	375
Bell (Beep) Command	377
bits	378

Break Command	381
Breakall Control Variable.....	382
Busbreak, Busremove, Busdisable, Busenable Commands.....	383
Cachememory Control Variable	386
Cause Control Variable	388
Character Functions	389
clock	395
Continue Command	396
Cos Command	397
Cpubreak, Cpuremove, Cpudisable, Cpunable Commands	398
cpuid_eax	400
cpuid_ebx	402
cpuid_ecx	404
cpuid_edx	406
cscfg and local_cscfg	408
csr.....	419
Ctime Command	421
cwd	422
Dataqualmode, Dataqualstart Commands	423
Dbgbreak, Dbgremove, Dbgdisable, Dbgenable Commands	425
defaultpath.....	427
Define Command	429
DefineMacro Command	432
devicelist.....	434
Displayflag Control Variable	438

Dos Command	439
Do While Command	440
Dport Command	442
Edit Command.....	443
Editor Control Variable	444
Eflags Command.....	445
Emubreak, Emuremove, Emudisable, Emuenable Commands.....	447
Encrypt Command.....	449
error	450
idcode	451
Eval Command.....	453
Execution Point Command.....	454
Exit Command	455
Exp Command.....	456
Fc Command.....	457
Fclose Command	458
Feof Command.....	459
Fgetc Command.....	460
Fgets Command.....	461
first_jtag_device	462
Flags Command	463
flist	465
Flush Command	467
Fopen Command.....	468
For Command.....	470

fprintf.....	472
Fputc Command.....	473
Fputs Command.....	474
Fread Command	475
fseek	476
ftell	477
Fwrite Command	478
Getc Command	479
getfile	480
getNearestProgramSymbol.....	482
GetProgramSymbolAddress Command.....	484
Gets Command	486
go.....	487
Halt Command	489
Help Command.....	490
homepath.....	491
Hotplug Command	492
idcode	493
If Command.....	495
include	497
IsDebugSymbol Command	499
isem64t.....	500
IsProgramSymbol Command	501
isrunning	502
issmm	504

itpcompatible	506
JtagChain Command.....	507
JtagDeviceAdd Command.....	509
JtagDeviceClear Command	510
JtagDevices Command	511
JtagScan Command.....	512
Keys Command.....	513
Last Command	515
last_jtag_device.....	517
Left Command	518
libcall.....	519
License Command	522
Linear Command.....	523
List, Nolist Commands	524
Load Command.....	526
LoadProject Command.....	527
LoadTarget Command	528
Log, Nolog Commands.....	529
Log10 Command.....	531
Loge Command.....	532
MacroPath Control Variable	533
messagebox	534
Mid Command	536
Msgclose Command.....	537
Msgdata Command	538

Msgdelete Command	540
Msgdr Command.....	541
Msgir Command	543
Msgopen Command	545
Msgreturndatasize Command	546
Msgscan Command	547
Msr Command.....	548
num_devices	549
num_jtag_devices	550
num_processors.....	551
pause.....	552
Physical Command	553
Port Command	554
Pow Command.....	556
Print Command.....	557
Print Cycles Command.....	558
printf.....	559
Proc Command.....	562
ProcessorControl Control Variable.....	563
ProcessorFamily Function.....	565
Processors Control Variable.....	566
ProcessorMode Command.....	567
ProcessorType Function	568
projectpath.....	569
Putchar Command	570

Puts Command	571
Rand Command	572
ReadSetting Command	573
reg	574
Reload Command	576
ReloadProject Command	577
Remove Command	578
Reset Command	580
Right Command	582
restart	583
Safemode Control Variable	584
SequenceType and SequenceCount Commands.....	585
Shell Command	587
Show Command	588
Sin Command.....	590
sleep	591
Softbreak, Softremove, Softdisable, Softenable Commands.....	592
sprintf.....	594
Sqrt Command	595
Srand Command	596
step.....	597
Stop Command	599
Strcat Command	600
strchr.....	601
Strcmp Command	602

Strcpy Command.....	604
String [] (Index Into String) Functions.....	605
Strlen Command	606
Strncat Command	607
Strncmp Command	608
Strncpy Command.....	610
_strdate.....	611
_strlwr	612
strpos.....	613
Strstr Command	614
_strtime.....	615
Strtod Command	616
Strtol Command	617
Strtoul Command	619
_strupr	621
Swbreak Command.....	622
Switch Control Construct.....	624
Swremove Command.....	626
Tabs Control Variable.....	627
Tan Command.....	628
TargPower Control Variable	629
TargStatus Control Variable	630
Taskattach Command	631
Taskbreak, Taskremove, Taskdisable, Taskenable Commands	632
Taskend Command	634

Taskgetpid Command	635
Taskstart Command	636
Tck Control Variable	638
Time Command	639
Triggerposition Command	640
Unload Command	641
UnloadProject Command	642
Use Control Variable	643
Ver Command	644
Verify Control Variable	645
Viewpoint Control Variable	646
vpalias	647
Wait Command	649
While Command	650
Wport Command	652
WriteSetting Command	653
Yield Command	654
Yieldflag Control Variable	655
Putfile Command	657
Osaware Control Variable	658
Index	659

Using Help

SourcePoint Help Overview



Address: 14811 Myford Road, Tustin, CA 92780
Phone: 877-508-3970 toll free in the US
714-731-1661 outside the US
Fax: 714-731-6344
E-Mail: info@arium.com
URL: <http://www.arium.com>

Welcome to Help

SourcePoint™ Version 7.7.1

[Using HTML Help](#)
[Installing SourcePoint](#)
[SourcePoint Environment](#)
[Breakpoints Window](#)
[Code Window](#)
[Command Window](#)
[Confidence Tests Window](#)
[Descriptors Tables Window](#)
[Devices Window](#)
[Log Window](#)
[Memory Window](#)
[Operating Systems Window](#)
[Page Translation Window](#)
[PCI Devices Window](#)
[Registers Window](#)
[Symbols Window](#)
[Trace Window](#)
[Viewpoint Window](#)
[Watch Window](#)

For more information on SourcePoint, SourcePoint commands, and SourcePoint technical notes, see the table of contents.

About SourcePoint Help User's Guide

Copyright © 2007 Arium, Tustin, CA. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without the prior permission of Arium.

While every precaution has been taken in the preparation of this document, Arium assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

Trademarks

SourcePoint is a trademark of Arium.

Intel and the names of all Intel processors are trademarks or registered trademarks of Intel Corporation.

Microsoft, Windows, and MS DOS are registered trademarks of Microsoft Corporation.

The zlib compression library © 1995-1998, Jean-loup Gailly and Mark Adler, is used in this product.

Note:

SourcePoint 7.3 supports Intel IA-32, Intel 64-bit extensions to IA-32, and AMD64 processors. This version of SourcePoint does not support Intel IA-64 processors.

How To - HTML Help

How to Use SourcePoint Help

SourcePoint incorporates HTML Help. The following topics cover basic Help operations:

[Find a topic](#)

[Change window size](#)

[Copy a topic](#)

[Print a topic](#)

[Move through help topics](#)

[Change fonts](#)

[Change colors](#)

[Hide/show help contents](#)

How to Find a Help Topic

In Help, click one of the following tabs:

- To browse through topics by category, click the **Contents** tab.
- To see a list of index entries, click the **Index** tab, and then either type a word or scroll through the list.
- To search for words or phrases that may be contained in a Help topic, click the **Search** tab.

In the left pane of the Help window, double-click the topic, to display the corresponding topic in the right frame.

- To find the definition of a word or to find out what word(s) an acronym stands for, click on the **Glossary** tab.

In the upper left pane of the Help window, click on the word or acronym to display the corresponding description in the lower left pane.

How to Change the Window Size

To make the left or right window pane wider or narrower, point to the divider between the two frames. When the pointer changes to a double-headed arrow, hold down the left mouse button as you drag the divider right or left.

To proportionally shrink or enlarge the entire Help window, point to any corner of the Help window. When the pointer changes to a double-headed arrow, hold down the left mouse button as you drag the corner.

To change the whole Help window height or width, point to the top, bottom, left, or right edge of the Help window. When the pointer changes to a double-headed arrow, hold down the left mouse button as you drag the side.

How to Copy a Help Topic

1. In the right pane of the **Help** window, right-click inside the topic you want to copy, and then click **Select All**.
2. Inside the topic, right-click again, and then click **Copy**.

This copies the topic to the Clipboard.

3. Open the document where you want to place the copy of the topic.
4. Click the place in your document where you want the information to appear.
5. On the **Edit** menu, click **Paste**.

Note: If you want to copy only part of a topic, select the part you want to copy, right-click your selection, and then click **Copy**.

How to Print a Help Topic

1. In the right pane of the **Help** window, click the topic you want to print.
2. On the Help toolbar, click **Options**.
3. Click **Print**.

Note: If you print from the **Contents** tab, you will see options to print the current topic, all topics under the selected book, or all topics in the table of contents. To print a pop-up topic, right-click inside the pop-up window, and then click **Print Topic**.

How to Make Relative Moves

On the Help toolbar, click **Back** to display the last Help topic you saw.

On the Help toolbar, click **Forward** to display the next Help topic in a previously displayed sequence of topics.

How to Change Fonts

1. In Internet Explorer 4.0 or later, on the **View** menu click **Internet Options**.
2. On the **General** tab, click **Fonts**.
3. In the **Fonts** dialog box, select the options you want, and then click **OK**.
4. On the **General** tab, click **Accessibility**.
5. Under **Formatting**, select the check boxes for the options you want, and then click **OK**.
6. To apply the new font settings, in the **Internet Options** dialog box, click **OK**.

Note: This changes the font only in the right frame of the Help window.

How to Change Colors

1. In Internet Explorer 4.0 or later, on the **View** menu click **Internet Options**.
2. On the **General** tab, click **Colors**.
3. In the **Colors** dialog box, select the options you want, and then click **OK**.
4. On the **General** tab, click **Accessibility**.
5. Under **Formatting**, select the check boxes for the options you want, and then click **OK**.
6. To apply the new color settings, in the **Internet Options** dialog box, click **OK**.

Note: This changes the color only in the right frame of the Help window.

How to Hide or Show Help Contents

On the Help toolbar, click **Hide** to hide the table of contents, index, glossary, or search results list.

On the Help toolbar, click **Show** to display the table of contents, index, glossary or search results list.

Arium

What's New in SourcePoint 7.7.1

A version of Sourcepoint 7.7.1 is now available that runs on Linux. Supported versions of Linux are Fedora (up to and including Core 7), Suse 10.2, and Red Hat Enterprise (up to and including v5).

The PCI Devices window now displays capabilities for each device and also displays bit fields within registers.

The Devices window has a new MSR cell type that allows any MSR number to be displayed. Bit fields within these registers are also supported.

The following commands were added to be more compatible with the Intel ITP:

- libcall, byref
- idcode
- cpuid
- devicelist
- pause
- csr
- bits
- isrunning
- num_processors
- error
- vpalias

The following commands were added to the command language:

- fseek
- ftell
- itpcompatible control variable
- _strlwr
- _strupr
- _strdate
- _strtime
- clock
- strchr
- strops
- getNearestProgramSymbol
- messageBox
- restart

Miscellaneous

- The ord12, ord16 and real 10 data types were added to the command language.
- The default size of integers was changed from 16 bits to 32 bits in the command language.
- The Memory view now supports Unicode display formats.
- The Devices view has a new MSR cell type for reading and writing any MSR.
- Emulator Configuration dialog
 - o No Test-Logic-Reset control added to JTAG tab.
 - o TCK0 and TCK1 Edge rate controls added to JTAG tab (for ECM-XDP3).
 - o JTAG Voltage control added to JTAG tab (for ECM-XDP3).

SourcePoint 7.7.1

- o Changes made to ECM-XDP3 tab.
- Symbol Finder button added to Memory view, Code view, and Add Breakpoint dialog.
- SourcePath dialogs overhauled.
- Find Symbol added to Symbol view context menu.

Contacting Arium

Headquarters:

14811 Myford Road, Tustin, CA 92780

Browse:

<http://www.arium.com>

Phone:

Headquarters: (877) 508-3970 (toll free in the US) or (714) 731-1661 (outside the US)

East Coast Office: (866) 700-9144 (toll free in the US) or (919) 323-8194 (outside the US)

Fax:

Headquarters: (714) 731-6344

E-Mail:

Sales: info@arium.com

Support: support@arium.com

Contacting Technical Support

Before contacting technical support, please run the **aadump()** command. Either include the results with your e-mail or have them in front of you when calling Arium.

To contact us:

Web

<http://www.arium.com/support.html>

Here you will find file downloads, application notes, and sample command macros.

Email

support@arium.com

Please attach the results of '[aadump\(\)](#)' command along with a description of your problem.

Phone

877-508-3970 - Western US, Mon-Fri, 8:00 am - 5:00 PM PT

866-700-9144 - Eastern US, Mon-Fri, 8:30 am - 5:30 PM ET

International support

For a list of distributors and their contact information, go to:

<http://www.arium.com/contact/distributors.html>

Installation

Installation Overview

Host System Requirements

The following system features are required at a minimum to run SourcePoint:

- Intel® Pentium® processor or better computer or equivalent
- Microsoft® Windows® XP (SP2) or Vista*
- 512 MB RAM (1024 MB recommended)
- 60 MB disk
- CD-ROM drive
- 10/100Base-T LAN port or USB port

* Vista users must install SourcePoint for each user.

The following system features are required at a minimum to run SourcePoint on a Linux host:

- 700 MHz processor (1.5 GHz recommended)
- Linux
- 512 MB RAM (1024 MB recommended)
- 8 GB free disk space (20 GB recommended)
- CD-ROM drive
- 10/100Base-T LAN port or USB port

Information on installing [hardware](#) and [software](#) for single units is found under "Installation Overview," part of *Installation*. (Specifics on hardware installation pack with the unit.) For details on hardware and software installation with a multi-cluster target, see the topic entitled, "[Multi-Clustering](#)" found under *Technical Notes*.

Installing and Configuring the ECM-HDT

1. **Unpack the equipment.** If anything is missing, contact Arium immediately.

Note: Power down your target if it is on.

2. **Install the Arium software.** Install SourcePoint by inserting the SourcePoint CD into the CD drive of your host computer. The **SourcePointShield Wizard** displays. Complete the wizard.

Note: You may need to contact your systems administrator to gain administrator privileges.

Note: If the setup program does not run automatically, you can start it manually. Choose **Run** from the **Start** menu on your computer, type the following command, and click OK.

```
<CD-ROM drive>:\disk1\setup.exe
```

The setup program installs the SourcePoint software onto your host computer. If necessary, you can rerun the setup program to install additional features.

During setup, you will be asked for the SourcePoint license file. This file resides on a separate CD that shipped with your unit. Insert the file CD when prompted. Install the license file to your root SourcePoint directory.

3. **Connect the emulator to the host computer.** Begin by determining the type of connection you want. A direct TCP/IP or USB connection is the simplest way to connect the host computer and the base unit. Only a single host PC can use the Arium debugger with this type of connection. A network TCP/IP connection may be your best choice if you already have a network in place. It will allow users from different locations to make use of the debugger if your network security allows such access.

For a **USB connection**, use the beige USB cable. Connect one end to the base unit at the connector port labeled USB. Connect the other end to the host computer's USB port. For a **direct TCP/IP connection**, use the orange direct crossover cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to the host computer at the 10/100Base-T network connector (also known as the RJ-45 connector). For a **network TCP/IP connection**, use the blue Ethernet patch cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to your network hub.

4. **Connect the emulator to the target.** There are two choices for the HDT cable (the standard 24-inch cable and a 6-inch cable for improved signal quality) for JTAG connection to the target and run control for one processor. The MPHDT cable **MUST** be connected to provide run control for multiprocessor-capable targets, even if only one processor is being debugged. Determine which cable(s) to use. Choose either the long or short HDT cable and, if the target has multiple-processor capability, the MPHDT cable. Connect the cable(s) to the connector(s) located on the front panel of the emulator and to the debug port(s) on your target.
5. **Power up the emulator.** Connect the power supply to the coaxial power connector on the back of the emulator. Then connect the other end to your power source. Finally, flip the On/Off switch on the front of the unit to the On position.

6. **Power up your target.**
7. **Install the USB driver (if you are using a USB connection).** If you have set up the hardware for a USB connection, you are first asked to load the USB driver via a standard Microsoft Windows dialog box. The driver file you need to load is "AriumUsb.inf", located in the root directory of the CD-ROM or the "SB_Drivers" subdirectory of your SourcePoint installation.

Note: If you have not set up a USB connection, skip Step 7.

8. **Launch and configure SourcePoint.** To launch SourcePoint, double-click on the "sp.exe" file in the SourcePoint directory.

You do not have to configure a USB connection from the SourcePoint interface. It is a "plug and play" connection. SourcePoint opens with the **New Project Wizard** displayed. Complete the wizard to begin working in SourcePoint. (If the wizard does not display, click on **File|Project|New Project** in the SourcePoint menu bar.)

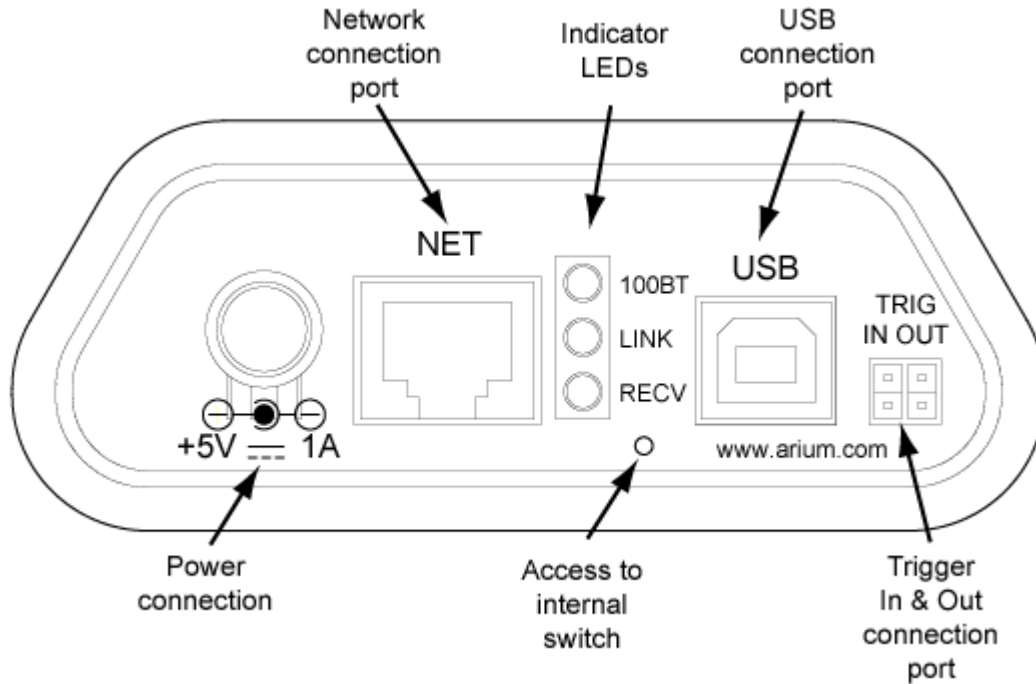
Note: If you want to determine the IP address that was used in configuring the connection or if you want to change that address, select **Options|Emulator Configuration** from the menu bar and click on the **Network** tab. Changes made in this tab do not affect the entries in the **Emulator Connections** dialog box.

If you are configuring a TCP/IP connection, you may need to know the serial number of your emulator. It can be found on the bottom of the unit. You may need some or all of the values listed below in order to configure the emulator with the debugger connection, depending on your connection type. In some instances you can use the default values. In others, you may need to contact your network administrator for this information.

	Direct TCP/IP Connection	Network TCP/IP Connection
Host PC TCP/IP Address*	Use 192.168.000.002	N/A
Emulator Base Unit TCP/IP Address	Use 192.168.000.001	Determined automatically by the wizard
Emulator Base Unit Network Mask	Use 255.255.255.000	Determined automatically by the wizard
Emulator Base Unit Network Gateway	Use 192.168.000.002	Determined automatically by the wizard

*This value is set on the network card of your host computer.

At this point, it is a good idea to locate the internal switch access hole on the emulator's back panel. You are asked to access it when you are in the TCP/IP setup wizard.



Typical back panel with internal switch

In SourcePoint, select **Options|Emulator Connection** from the menu bar if this dialog box is not open already. Click on the **TCP/IP Setup** button. Follow the on-screen instructions to properly configure the emulator TCP/IP settings.

Caution: Arium recommends you do not configure a dynamic IP address at initial configuration. Once you have made an initial connection, you may be able to establish a dynamic IP address, depending on your server. Details are provided in the manual/online help files.

9. **Configure the emulator TCK current level setting.** The JTAG current level is set in the SourcePoint debugger. Select **Options|Emulator Configuration** from the menu bar. After the dialog box opens, go to the **JTAG** tab. Set the JTAG current to the highest value if the TCK termination resistor on the target is a matched-end termination. Otherwise, set it to less.

In the same drop down box, set the TCK edge rate. There are two choices of TCK edge rate, "slow" and fast." Use a "slow" setting if the target TCK signal quality is questionable (due to poor routing or poor termination). Use a "fast" rate for greatest speed.

Installing and Configuring the ECM-700

1. **Unpack the equipment.** If anything is missing, contact Arium immediately.

Note: Power down your target if it is on.

2. **Install the Arium software.** Install SourcePoint by inserting the SourcePoint CD into the CD drive of your host computer. The **SourcePointShield Wizard** displays. Complete the wizard.

Note: You may need to contact your systems administrator to gain administrator privileges.

Note: If the setup program does not run automatically, you can start it manually. Choose **Run** from the **Start** menu on your computer, type the following command, and click OK.

```
<CD-ROM drive>:\disk1\setup.exe
```

The setup program installs the SourcePoint software onto your host computer. If necessary, you can rerun the setup program to install additional features.

During setup, you will be asked for the SourcePoint license file. This file resides on a separate CD that shipped with your unit. Insert the file CD when prompted. Install the license file to your root SourcePoint directory.

3. **Connect the emulator to the host computer.** Begin by determining the type of connection you want. A direct TCP/IP or USB connection is the simplest way to connect the host computer and the base unit. Only a single host PC can use the Arium debugger with this type of connection. A network TCP/IP connection may be your best choice if you already have a network in place. It will allow users from different locations to make use of the debugger if your network security allows such access.

For a **USB connection**, use the beige USB cable. Connect one end to the base unit at the connector port labeled USB. Connect the other end to the host computer's USB port. For a **direct TCP/IP connection**, use the orange direct crossover cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to the host computer at the 10/100Base-T network connector (also known as the RJ-45 connector). For a **network TCP/IP connection**, use the blue Ethernet patch cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to your network hub.

4. **Connect the emulator to the target.** The purpose of the PBD-700E is to act as a buffer and provide voltage level shifting between the target debug/ITP port and the emulator. Connect the cable to the connector located on the front panel of the emulator. Connect the PBD-700E board to your target debug port.

Note: For information on how to set the jumpers on the PBD-700E, see the PBD-700E *Getting Started* guide that shipped with your unit.

5. **Power up the emulator.** Connect the power supply to the coaxial power connector on the back of the emulator. Then connect the other end to your power source. Finally, flip the On/Off switch on the front of the unit to the On position.

6. **Power up your target.**
7. **Install the USB driver (if you are using a USB connection).** If you have set up the hardware for a USB connection, you are first asked to load the USB driver via a standard Microsoft Windows dialog box. The driver file you need to load is "AriumUsb.inf", located in the root directory of the CD-ROM or the "SB_Drivers" subdirectory of your SourcePoint installation.

Note: If you have not set up a USB connection, skip Step 7.

8. **Launch and configure SourcePoint.** To launch SourcePoint, double-click on the "sp.exe" file in the SourcePoint directory.

You do not have to configure a USB connection from the SourcePoint interface. It is a "plug and play" connection. SourcePoint opens with the **New Project Wizard** displayed. Complete the wizard to begin working in SourcePoint. (If the wizard does not display, click on **File|Project|New Project** in the SourcePoint menu bar.)

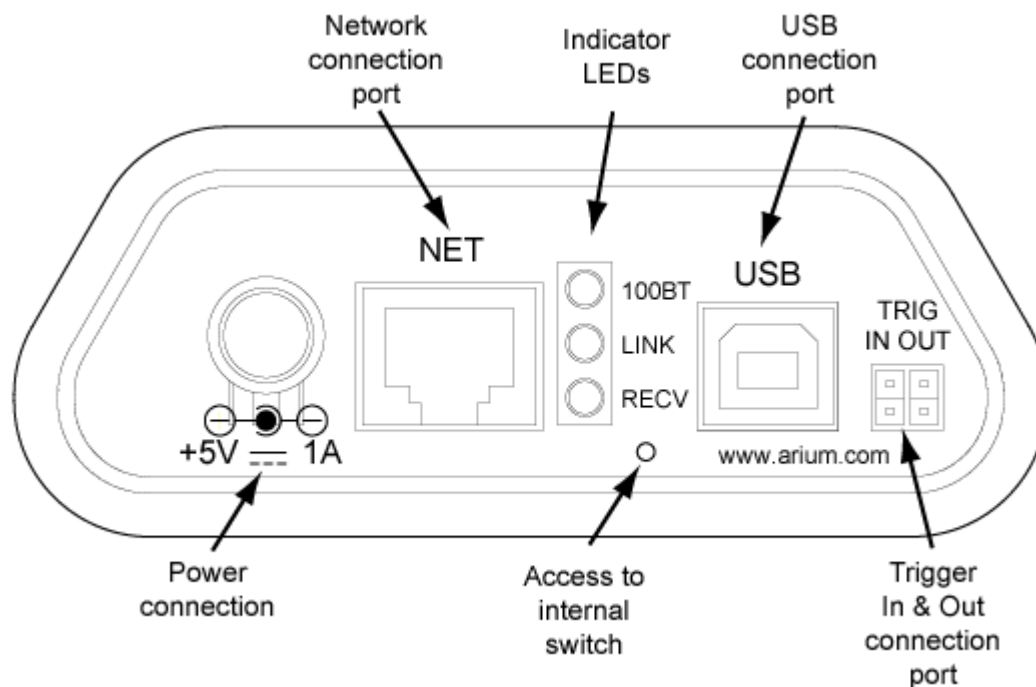
Note: If you want to determine the IP address that was used in configuring the connection or if you want to change that address, select **Options|Emulator Configuration** from the menu bar and click on the **Network** tab. Changes made in this tab do not affect the entries in the **Emulator Connections** dialog box.

If you are configuring a TCP/IP connection, you may need to know the serial number of your emulator. It can be found on the bottom of the unit. You may need some or all of the values listed below in order to configure the emulator with the debugger connection, depending on your connection type. In some instances you can use the default values. In others, you may need to contact your network administrator for this information.

	Direct TCP/IP Connection	Network TCP/IP Connection
Host PC TCP/IP Address*	Use 192.168.000.002	N/A
Emulator Base Unit TCP/IP Address	Use 192.168.000.001	Determined automatically by the wizard
Emulator Base Unit Network Mask	Use 255.255.255.000	Determined automatically by the wizard
Emulator Base Unit Network Gateway	Use 192.168.000.002	Determined automatically by the wizard

*This value is set on the network card of your host computer.

At this point, it is a good idea to locate the internal switch access hole on the emulator's back panel. You are asked to access it when you are in the TCP/IP setup wizard.



Typical back panel with internal switch

In SourcePoint, select **Options|Emulator Connection** from the menu bar if this dialog box is not open already. Click on the **TCP/IP Setup** button. Follow the on-screen instructions to properly configure the emulator TCP/IP settings.

Caution: Arium recommends you do not configure a dynamic IP address at initial configuration. Once you have made an initial connection, you may be able to establish a dynamic IP address, depending on your server. Details are provided in the manual/online help files.

Installing and Configuring the ECM-XDP

1. **Unpack the equipment.** If anything is missing, contact Arium immediately.

Note: Power down your target if it is on.

2. **Install the Arium software.** Install SourcePoint by inserting the SourcePoint CD into the CD drive of your host computer. The **SourcePointShield Wizard** displays. Complete the wizard.

Note: You may need to contact your systems administrator to gain administrator privileges.

Note: If the setup program does not run automatically, you can start it manually. Choose **Run** from the **Start** menu on your computer, type the following command, and click OK.

```
<CD-ROM drive>:\disk1\setup.exe
```

The setup program installs the SourcePoint software onto your host computer. If necessary, you can rerun the setup program to install additional features.

During setup, you will be asked for the SourcePoint license file. This file resides on a separate CD that shipped with your unit. Insert the file CD when prompted. Install the license file to your root SourcePoint directory.

3. **Connect the emulator to the host computer.** Begin by determining the type of connection you want. A direct TCP/IP or USB connection is the simplest way to connect the host computer and the base unit. Only a single host PC can use the Arium debugger with this type of connection. A network TCP/IP connection may be your best choice if you already have a network in place. It will allow users from different locations to make use of the debugger if your network security allows such access.

For a **USB connection**, use the beige USB cable. Connect one end to the base unit at the connector port labeled USB. Connect the other end to the host computer's USB port. For a **direct TCP/IP connection**, use the orange direct crossover cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to the host computer at the 10/100Base-T network connector (also known as the RJ-45 connector). For a **network TCP/IP connection**, use the blue Ethernet patch cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to your network hub.

4. **Connect the emulator to the target.** The purpose of the JTAG cable is to act as a buffer and provide voltage level shifting between the target debug/ITP port and the emulator. Carefully connect the cable to your target's XDP connector.

Note: Minimum bend radius on the cable is one-fourth inch (6.35 mm).

In the rare case that the target requires a supplemental reset signal in addition to DBR# present on the XDP, connect the yellow and black twisted reset cable from the RST OUT connector on the front panel to the target RESET push button post.

5. **Power up the emulator.** Connect the power supply to the coaxial power connector on the back of the emulator. Then connect the other end to your power source. Finally, flip the On/Off switch on the front of the unit to the On position.

6. **Power up your target.**
7. **Install the USB driver (if you are using a USB connection).** If you have set up the hardware for a USB connection, you are first asked to load the USB driver via a standard Microsoft Windows dialog box. The driver file you need to load is "AriumUsb.inf", located in the root directory of the CD-ROM or the "SB_Drivers" subdirectory of your SourcePoint installation.

Note: If you have not set up a USB connection, skip Step 7.

8. **Launch and configure SourcePoint.** To launch SourcePoint, double-click on the "sp.exe" file in the SourcePoint directory.

You do not have to configure a USB connection from the SourcePoint interface. It is a "plug and play" connection. SourcePoint opens with the **New Project Wizard** displayed. Complete the wizard to begin working in SourcePoint. (If the wizard does not display, click on **File|Project|New Project** in the SourcePoint menu bar.)

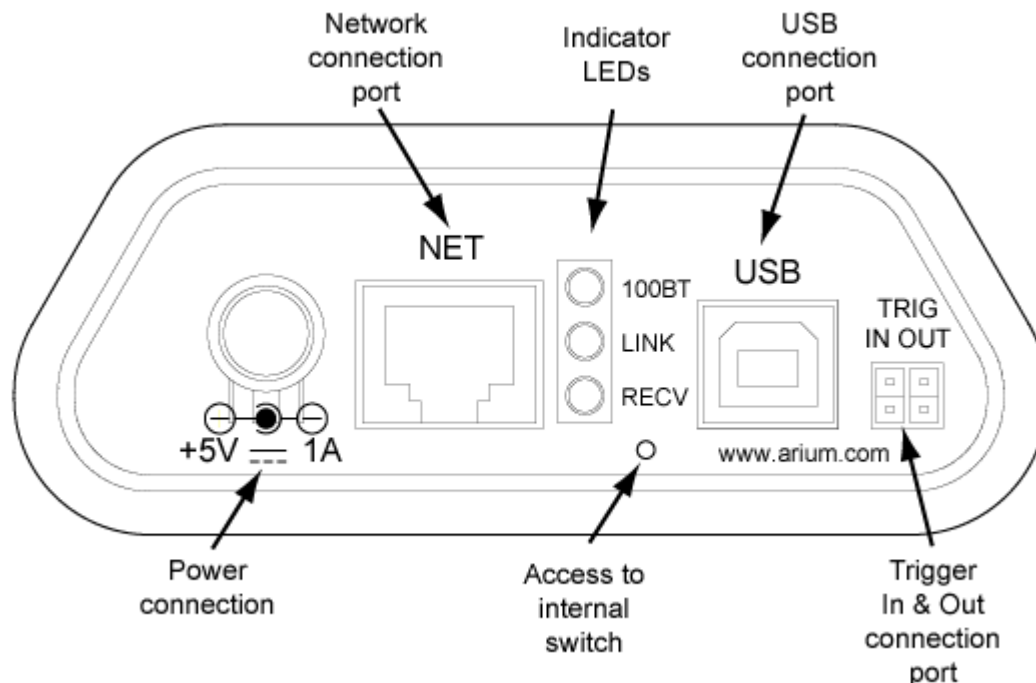
Note: If you want to determine the IP address that was used in configuring the connection or if you want to change that address, select **Options|Emulator Configuration** from the menu bar and click on the **Network** tab. Changes made in this tab do not affect the entries in the **Emulator Connections** dialog box.

If you are configuring a TCP/IP connection, you may need to know the serial number of your emulator. It can be found on the bottom of the unit. You may need some or all of the values listed below in order to configure the emulator with the debugger connection, depending on your connection type. In some instances you can use the default values. In others, you may need to contact your network administrator for this information.

	Direct TCP/IP Connection	Network TCP/IP Connection
Host PC TCP/IP Address*	Use 192.168.000.002	N/A
Emulator Base Unit TCP/IP Address	Use 192.168.000.001	Determined automatically by the wizard
Emulator Base Unit Network Mask	Use 255.255.255.000	Determined automatically by the wizard
Emulator Base Unit Network Gateway	Use 192.168.000.002	Determined automatically by the wizard

*This value is set on the network card of your host computer.

At this point, it is a good idea to locate the internal switch access hole on the emulator's back panel. You are asked to access it when you are in the TCP/IP setup wizard.



Typical back panel with internal switch

In SourcePoint, select **Options|Emulator Connection** from the menu bar if this dialog box is not open already. Click on the **TCP/IP Setup** button. Follow the on-screen instructions to properly configure the emulator TCP/IP settings.

Caution: Arium recommends you do not configure a dynamic IP address at initial configuration. Once you have made an initial connection, you may be able to establish a dynamic IP address, depending on your server. Details are provided in the manual/online help files.

9. **Configure the emulator connector pins.** The JTAG current level is set in the SourcePoint debugger. Select **Options|Emulator Configuration** from the menu bar. After the dialog box opens, click on the **XDP Pins** tab. Set the pins to configure the emulator for compatibility with your specific target hardware.

Faster JTAG access. The default state of some internal jumpering on the ECM-XDP allows it to work with almost any JTAG system, regardless of how well the TMS, TCK0 and TCK1 signals are terminated. However, this jumpering may reduce the maximum JTAG rate. To operate at higher JTAG rates on well designed targets, move the shunts on the E1, E2, and E3 jumpers from their default pin 2 to pin 3 position (away from the reference designation on the jumpers) to the pin 1 to pin 2 position. For poorly terminated targets (e.g., an adapter to a 25-pin ITP-700 style debug port TCK termination resistor near the debug port instead of at the far end of the trace), return E1, E2, and E3 to their pin-2-to-pin-3 position and lower the JTAG TCK frequency.

BCLK inversion. If needed for proper operation of bus-analyzer triggering (available only on certain processors), BCLK may be inverted by changing jumpers inside the ECM-XDP. By default shunts are installed on JP1 and JP2, and BCLK is not inverted. To invert BCLK,

rotate the pair of shunts 90 degrees to cross-connect the two jumpers, then short each pin on JP1 to its neighbor on JP2.

Enable only those pins/groups that are connected to processor “BPM” signals on the target. (BPM 5# and BPM 4# may have alternate names PREQ# and PRDY#.).

Installing and Configuring the ECM-50

1. **Unpack the equipment.** If anything is missing, contact Arium immediately.

Note: Power down your target if it is on.

2. **Install the Arium software.** Install SourcePoint by inserting the SourcePoint CD into the CD drive of your host computer. The **SourcePointShield Wizard** displays. Complete the wizard.

Note: You may need to contact your systems administrator to gain administrator privileges.

Note: If the setup program does not run automatically, you can start it manually. Choose **Run** from the **Start** menu on your computer, type the following command, and click OK.

```
<CD-ROM drive>:\disk1\setup.exe
```

The setup program installs the SourcePoint software onto your host computer. If necessary, you can rerun the setup program to install additional features.

During setup, you will be asked for the SourcePoint license file. This file resides on a separate CD that shipped with your unit. Insert the file CD when prompted. Install the license file to your root SourcePoint directory.

3. **Connect the emulator to the host computer.** Begin by determining the type of connection you want. A direct TCP/IP or USB connection is the simplest way to connect the host computer and the base unit. Only a single host PC can use the Arium debugger with this type of connection. A network TCP/IP connection may be your best choice if you already have a network in place. It will allow users from different locations to make use of the debugger if your network security allows such access.

For a **USB connection**, use the beige USB cable. Connect one end to the base unit at the connector port labeled USB. Connect the other end to the host computer's USB port. For a **direct TCP/IP connection**, use the orange direct crossover cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to the host computer at the 10/100Base-T network connector (also known as the RJ-45 connector). For a **network TCP/IP connection**, use the blue Ethernet patch cable. Connect one end to the emulator at the connector port labeled NETWORK. Connect the other end to your network hub.

4. **Connect the emulator to the target.** The purpose of the JTAG cable is to act as a buffer and provide voltage level shifting between the target debug/ITP port and the emulator. Connect the cable that shipped with your unit to the connector located on the front panel of the emulator and the other end to the debug port.

Note: For information on the personality module, see the appropriate *Getting Started* guide that shipped in the box with your unit.

5. **Power up the emulator.** Connect the power supply to the coaxial power connector on the back of the emulator. Then connect the other end to your power source. Finally, flip the On/Off switch on the front of the unit to the On position.

6. **Power up your target.**
7. **Install the USB driver (if you are using a USB connection).** If you have set up the hardware for a USB connection, you are first asked to load the USB driver via a standard Microsoft Windows dialog box. The driver file you need to load is "AriumUsb.inf", located in the root directory of the CD-ROM or the "SB_Drivers" subdirectory of your SourcePoint installation.

Note: If you have not set up a USB connection, skip Step 7.

8. **Launch and configure SourcePoint.** To launch SourcePoint, double-click on the "sp.exe" file in the SourcePoint directory.

You do not have to configure a USB connection from the SourcePoint interface. It is a "plug and play" connection. SourcePoint opens with the **New Project Wizard** displayed. Complete the wizard to begin working in SourcePoint. (If the wizard does not display, click on **File|Project|New Project** in the SourcePoint menu bar.)

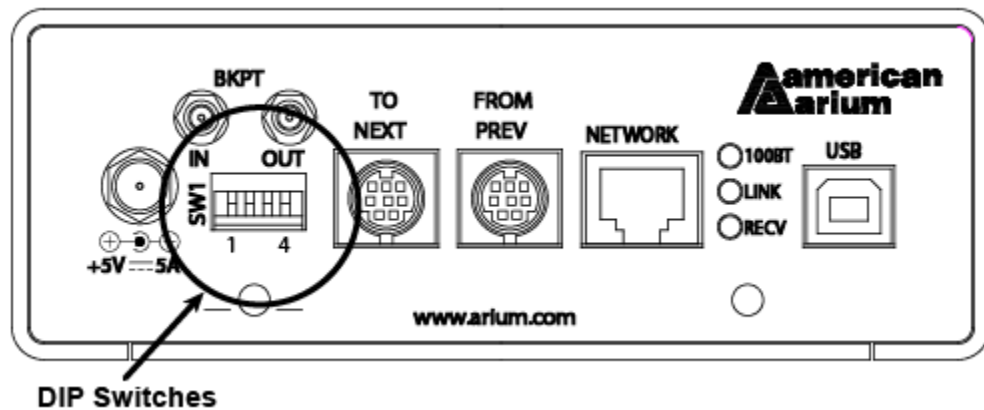
Note: If you want to determine the IP address that was used in configuring the connection or if you want to change that address, select **Options|Emulator Configuration** from the menu bar and click on the **Network** tab. Changes made in this tab do not affect the entries in the **Emulator Connections** dialog box.

If you are configuring a TCP/IP connection, you may need to know the serial number of your emulator. It can be found on the bottom of the unit. You may need some or all of the values listed below in order to configure the emulator with the debugger connection, depending on your connection type. In some instances you can use the default values. In others, you may need to contact your network administrator for this information.

	Direct TCP/IP Connection	Network TCP/IP Connection
Host PC TCP/IP Address*	Use 192.168.000.002	N/A
Emulator Base Unit TCP/IP Address	Use 192.168.000.001	Determined automatically by the wizard
Emulator Base Unit Network Mask	Use 255.255.255.000	Determined automatically by the wizard
Emulator Base Unit Network Gateway	Use 192.168.000.002	Determined automatically by the wizard

*This value is set on the network card of your host computer.

At this point, it is a good idea to locate the DIP switches on the emulator's back panel. You are asked to flip one of the switches when you are in the TCP/IP setup wizard.



ECM-50 back panel

In SourcePoint, select **Options|Emulator Connection** from the menu bar if this dialog box is not open already. Click on the **TCP/IP Setup** button. Follow the on-screen instructions to properly configure the emulator TCP/IP settings.

Caution: Arium recommends you do not configure a dynamic IP address at initial configuration. Once you have made an initial connection, you may be able to establish a dynamic IP address, depending on your server. Details are provided in the manual/online help files.

BKPT In/Out

BKPT In

A transition from the “running” to the “stopped” level on this signal causes the target/processors to stop. The opposite transition does NOT restart the target, however.

- Input logic levels (TTL): $V_{il}=0.8V$, $V_{ih}=2.0V$
- Recommended input voltage limits: Low: -0.5V, High: 3.6V
- Active level is selectable (High=trigger, or Low=trigger)
- If High=trigger, then internal termination is 64 ohms to 0.64 V
- If Low=trigger, then internal termination is 64 ohms to 2.4 V

BKPT Out

This signal provides real time status of the “running” or “stopped” state of the target/processors. The signal will be in the “running” state when running and in the “stopped” state when the target is stopped.

- Active level is selectable (High=running, or Low=running)
- Output High characteristics: 60 ohms (max) to 3.3V
- Output Low characteristics: 26 ohms (max) to GND.

Macrovision FLEXIm® Licensing

FLEXIm® licensing files act as a key, enabling use with specific emulator model/serial numbers and activating optional features within SourcePoint. SourcePoint requires a FLEXIm license file for successful installation. These files use a digital signature to ensure integrity. Each emulator is shipped with a separate disk that contains a license file corresponding to that emulator's serial number. If optional features or software upgrades are ordered at a later time, a new license file is generated by Arium and made available to you.

FLEXIm licensing files serve two purposes:

1. Indicate which SourcePoint optional features are activated.
2. Indicate the licensed SourcePoint version for the specified emulator serial number(s).

If the emulator serial number is not found in any FLEXIm license file, an error message is generated. Press the **OK** button to exit SourcePoint. If the file is lost or corrupt, contact Arium technical support for a replacement.

Note: SourcePoint does not function without a FLEXIm file.

Note: You need to copy the sp.lic file to the SourcePoint root directory; it is not recognized from a sub-directory.

When SourcePoint is executed, the emulator serial number is read, the corresponding license file is located, and activated feature information is verified. FLEXIm licensing files are generally stored in the same directory as the SourcePoint executable (default path = \Program Files\American Arium\SourcePoint) utilizing a ".lic" extension.

Command Line Key Words

After you have installed SourcePoint and started it once, you can create shortcuts to a specific SourcePoint program file. There are several command line key words recognized by SourcePoint. They can be placed either in the Properties|Short Cut|Target line of a SourcePoint icon/shortcut, or SourcePoint can be invoked from the command line.

The formats of the arguments are as follows:

-m or -i = begin running a macro or an include file immediately after startup

-p or -c = define the project file or configuration file that will be used

For example, if you wanted SourcePoint to run a macro file named "looper.mac", from the command line, type: "\Program Files\American Arium\SourcePoint-i\macros\looper mac".

How To - Installation

How to Update Flash Code

1. Open SourcePoint.

The main SourcePoint window displays.

2. Go to **File|Update Flash** on the menu bar.
3. A **File Open** dialog box displays.
4. Select the flash file appropriate to your emulator hardware. For additional information, refer to "ReadMe.txt" file.

When the flash update is successfully updated, a **Flash update complete** box appears.

5. Click the **OK** button.
6. Go to **File|Exit** on the menu bar.
7. Cycle power to the target and emulator, then restart SourcePoint.

Note: These instructions assume you have already run SourcePoint with your current hardware configuration. If not, please refer to the appropriate hardware installation instructions in your *Getting Started* guide that shipped with your unit to ensure the emulator switch and LAN and COM ports settings have been set properly.

SourcePoint Environment

SourcePoint Overview

SourcePoint Parent Window Introduction

SourcePoint is the software interface to all Arium emulator systems. The program is dedicated to providing non-intrusive, hardware-assisted debug support for Intel 32- and 64-bit processors and AMD64 processors. Applications include debugging hardware, BIOS, kernel, drivers, and embedded software.

When SourcePoint opens, in many ways it looks like any standard Microsoft® Windows® screen. Menu and icon bars at the top and a single screen fills much of your display. Various menu/icon options let you open other windows (or views), as needed, to debug your code.

This topic includes information on:

[Docking/floating menu items](#)

[Menu toolbar](#)

[Icon toolbar](#)

[Status bar](#)

Docking/Floating Menu Items

SourcePoint windows can float (the default behavior) or be docked. To dock a window, right-click on its title bar to display a context menu.

Docked/floating context menu

Docked/Floating menu items. Use these menu items to toggle between docking and floating windows. A view that is set for **Floating** can be moved outside of SourcePoint (onto another display, for example).

MDI Child menu item. This menu item causes the windows to be neither floating nor docked.

Docked to menu item. Options include **Top**, **Left**, **Bottom**, and **Right**. Use these options to tuck a view into a corner of the SourcePoint window.

MDI Child as menu item. Options include **Minimized**, **Maximized**, and **Restored**. These let you minimize a window, maximize it, and restore it to its previous size.

Toolbar Menu

There are two kinds of toolbar menus— the menu toolbar and the icon toolbar. The menu items associated with the text menu/icons are described in separate topics.

Menu Toolbar

[File Menu](#)

[Edit Menu](#)

[View Menu](#)

[Processor Menu](#)

[Options Menu](#)

[Window Menu](#)

[Help Menu](#)

Icon Toolbar

[SourcePoint Icon Toolbar](#)

Status Bar

The status bar contains information about the focus processor and the communication to the emulator.

Function Keys and Field Information

As SourcePoint is running, this text changes to describe what is happening. As you move the mouse over a detectable area, the text gives helpful information about that area. When errors occur, the text gives information about the error. When the application has no other information to give, the active function key combinations display.

Current Focus Processor Name

In a single-processor target system, this field does not display. In multi-processor target systems, one of the processors is selected as the current focus of display by SourcePoint, and that processor number is output in this status field.

Focus Processor Run State

This field gives the state of the current focus processor. The following are valid processor states:

Stopped. The processor is not executing instructions.

Running. The processor is currently executing instructions.

Stepping. SourcePoint is currently stepping the processor through instructions.

Sleeping. The processor is not in one of the above states.

Emulator display status indicator. The status number on the LED on the emulator displays on the taskbar here, too. This is designed for those of you working with a remote emulator.

Focus Processor Mode

This field displays the current focus processor mode.

Focus Processor Run State	
Processor Mode	Description
Real	The processor is emulating the addressing required for programs written for the 8086, 8088, 80186 or 80188 processor. This is the mode used after reset.
BigReal	The processor is emulating the addressing as though it were in Real mode, but addresses aren't limited to 20 bits.
Virtual86	The processor is emulating the programming environment of an 8086 processor.
Protected	The processor is enabled for addressing protection.
Special	The processor is in a special addressing mode such as that entered when in SMM (System Management Mode).
Switching	This represents the time between Real and Protected mode when the code is setting up for Protected mode.

Communications Status Indicator Lights

Connectivity. This field is solid green when there is an active connection to the emulator. Otherwise it is gray. A double-click on this field displays more information in the status field.

Send in progress. This field is solid purple when there is information going to the emulator. Otherwise it is gray. A double-click on this field displays more information in the status field.

Receive in progress. This field is solid blue when there is information coming from the emulator. Otherwise it is gray. A double-click on this field displays more information in the status field.

Error detected. This field is solid red when information has been lost or corrupted going to or coming from the emulator. Otherwise it is gray. A double-click on this field displays more information in the status field.

SourcePoint Icon Toolbar

The SourcePoint toolbar displayed on the SourcePoint main window directly links toolbar buttons to menu items. Clicking on a desired toolbar button executes a procedure in the same manner as selecting that same menu item from its corresponding menu.

Icons are organized into several groups. They are listed below along with information on the attendant context menu.

Icon Groups

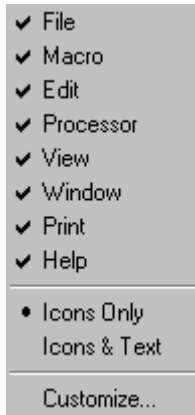
	File	Load Project, Reload Project, Save Project, Save Project As, Load Program, Reload Program, Load Macro, Update Emulator Flash, Program Target Flash
	Macro	Execute Macro 0, Execute Macro 1, Execute Macro 2, Macro 3*
	Edit	Cut, Copy, Paste, Search, Replace
	Processor	Go, Stop, Step Into, Step Over, Step Out Of, Reset, Connect**, Disconnect**, Snapshot
	View 1	Breakpoints window, Code window, Command window, Log window, Symbols window, Trace window, Watch window, Devices window, Memory window, Registers window, Viewpoint window
	View 2	Descriptors Table window, Devices window, Page Translation window, PCI Devices window
	Window	Close, Cascade, Tile Windows Horizontally, Tile Windows Vertically, Arrange Icons, Close All
	Print	Print, Print Preview
	Help	Help!

* You can customize the toolbar to include as many as 10 macros.

** Connects or disconnects emulator from target

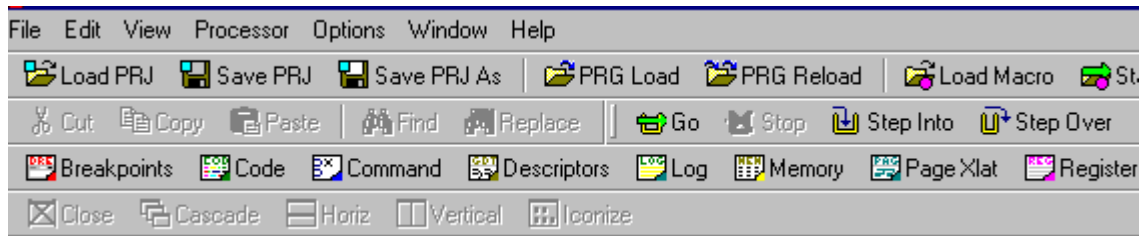
Context Menus

Right clicking on any icon brings up a context menu displaying the icon groups, icon displays with or without text options, and a toolbar customization option. You can choose to display text next to all icons or next to a single icon group.

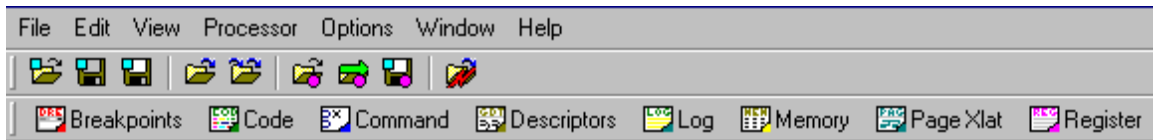


File, Macro, etc. menu items. The menu items in the top section of the menu let you choose which groups of icons you want to display. All icons are displayed by default.

Icons Only and Icons & Text menu items. You can choose to display text next to all icons or next to a single icon group. You can also choose to display the icons without text via the context menu.



Portion of toolbar showing all icons with text



Portion of toolbar showing a single icon group with text

Customize Menu Item

If you click on the **Customize** menu item, you are taken to a **Customize Toolbar** dialog box. From there you can customize your toolbar to best meet your needs. For more information, go to the topic, "[Edit Icon Groups to Customize Your Toolbars](#)" found under "How To - SourcePoint Environment," part of the *SourcePoint Overview*.

File Menu

For easier navigation, we have broken the subjects covered in this menu into several topics. Please click on the hyperlinked text below for documentation on a specific **File** menu item.

[Project Menu Item](#) - Options are New Project, Load Project, Reload Project, Save Project, Save Project As, and Unload Project.

[Layout Menu Item](#) - Options are **Load Layout** and **Save Layout**.

[Program Menu Item](#) - Options are **Load Program**, **Reload Program**, **Remove All Programs**, and **Save Program**.

[Macro Menu Item](#) - Options are **Load Macro** and **Configure Macros**.

[Print Menu Items](#) - Options are **Print**, **Print Preview**, and **Print Setup**.

[Update Emulator Flash Menu Item](#) - There are no other options with this menu item.

[Program Target Devices Menu Item](#) - Options are **Program Flash** and **Program PLD**.

[Other Menu Items](#) - **Save As**, **Recent Projects**, **Recent Layouts**, **Recent Programs**, **Recent Macros**, **Exit** menu items.

File Menu - Project Menu Item

Select **File|Project** on the menu bar to access the following options: **New Project**, **Load Project**, **Reload Project**, **Save Project**, **Save Project As**, and **Unload Project**.

New Project Option

Select **New Project** to create a new project file. The wizard allows you to select a name for the project file, load a target configuration from the Target Configuration Database, edit emulator configuration parameters, and edit target configuration parameters.

For additional information on the **New Project Wizard**, see, "[How to Use the New Project Wizard](#)," part of "How To - SourcePoint Environment," found under *SourcePoint Environment*.

Reload Project Option

Select **Reload Project** to reload the current project.

Save Project Option

Select **Save Project** to save the activated project settings file under its current file name.

Save Project As Option

Select **Save Project As** to open a **Save As** dialog box. The **Save As** dialog box is used to save information relevant to a window or group of windows in a project ("prj") file. The information saved includes the position, size, and parameter settings of each window. (Displayed data are not saved as they are governed by processor activity.)

Enter a file name with a "prj" extension, type in a name in the **File Name** text box, and click the **Save** button to save a window or group of windows in a project file.

Note: When SourcePoint is reopened for subsequent debugging sessions, the last window or group of windows saved as a project file is loaded automatically.

Unload Project Option

Select **Unload Project** to unload the current project. All windows are closed. If you have not saved the project, or if you have not saved a particular portion, you lose it when you use this option.

Caution: If you have disabled **Save settings on exit** under the **General** tab of the **Preferences** dialog box and you wish to retain the data and settings in a currently active window or window group, you must save the project ("prj") file before exiting SourcePoint. Select **File|Save Project** on the menu bar to save the file using the current name and location, or **File|Save Project As** to save it as another file name or location.

File Menu - Layout Menu Item

A layout is the set of open SourcePoint windows along with their locations, sizes, docking type, etc. The default file extension is .LYT. You can develop a set of layout files, each with a specific debugging purpose in mind, and can quickly access one when needed. Although you can just use multiple project files to accomplish this same functionality, loading a layout is less disruptive because it only affects the windows in the **View** menu that are open, including their locations, and sizes. Whereas loading a project file completely resets SourcePoint's entire state. Select the **Layout** menu item to load or save a layout file that you have generated.

Load Layout Menu Item

To load a user-generated SourcePoint layout that has been saved, click on **Load Layout** menu item in the **File** menu. Select the file you want to load.

Save Layout Menu Item

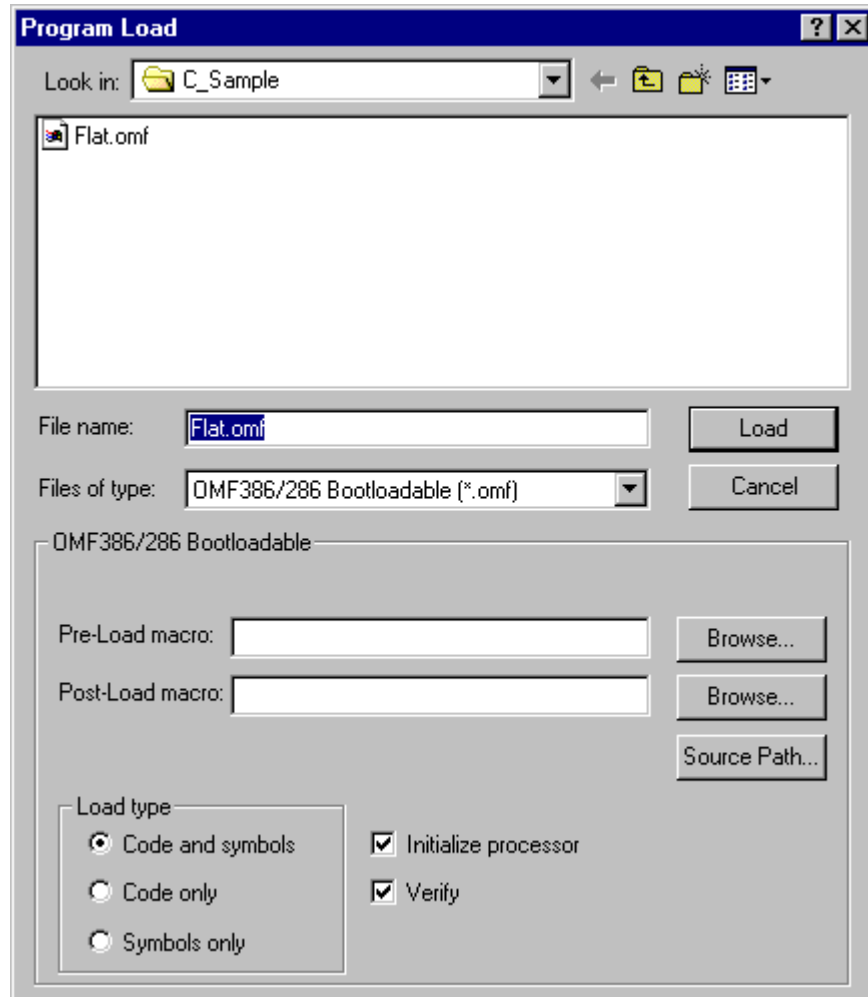
To save a user-generated SourcePoint layout, click on **Save Layout** option in the **File** menu. Select the file you want to save.

File Menu - Program Menu Item

Select **File|Program** on the menu bar to access the following options: **Load Program**, **Reload Program**, **Remove All Programs**, and **Save Program**. The **Program Load** and **Reload Program** options are also available as icons on the icon toolbar.

Load Program Option

The **Load Program** option allows you to load programs into target memory and/or load debug (symbol) information for symbolic or source-level debugging.



Program Load dialog box

The option supports a number of file formats, as listed in the table below. The format of the file affects the options available in the **Program Load** dialog box.

File Format Type				
File format	Initialize processor(1)	Macros(2)	Offset(3)	Address(4)

type				
OMF386/286 Bootable (*.omf)	X	X		
ELF executable (*.elf)	initializes EIP only	X	X	
Intel OMF86 files (*.omf)		X		
AOUT Executable (*.out)		X		
PE32/PE32+ (*.exe)	initializes EIP only	X		X
PE32/PE32+ (*.dll)	initializes EIP only	X		X
EFI(PE) format (*.efi)	initializes EIP only	X		X
MS-DOS EXE (*.exe)		X	X	
PDB format (*.pdb)		X		X
Intel HEX files (*.hex)		X	X	
Intel TEXTSYM symbol file (*.sym)		X	X	
Binary files (*.bin)		X		X

- (1) Provides initialization of processor registers
- (2) Pre- and/or post-load macro
- (3) Numeric offset added to load address of code/debug information to form new load address
- (4) Allows placement of file in user-selected memory location via address

Lower Half of Dialog Box

Depending on the format of the file you chose in the **List of files of type**, you have various options available.

Offset: This option lets you place the file in memory some place other than at the default setting.

- For the **Binary** format; enter the load address into the **Address** box.
- For relocatable formats, enter the signed relocation value into the **Offset** box (typically 0).

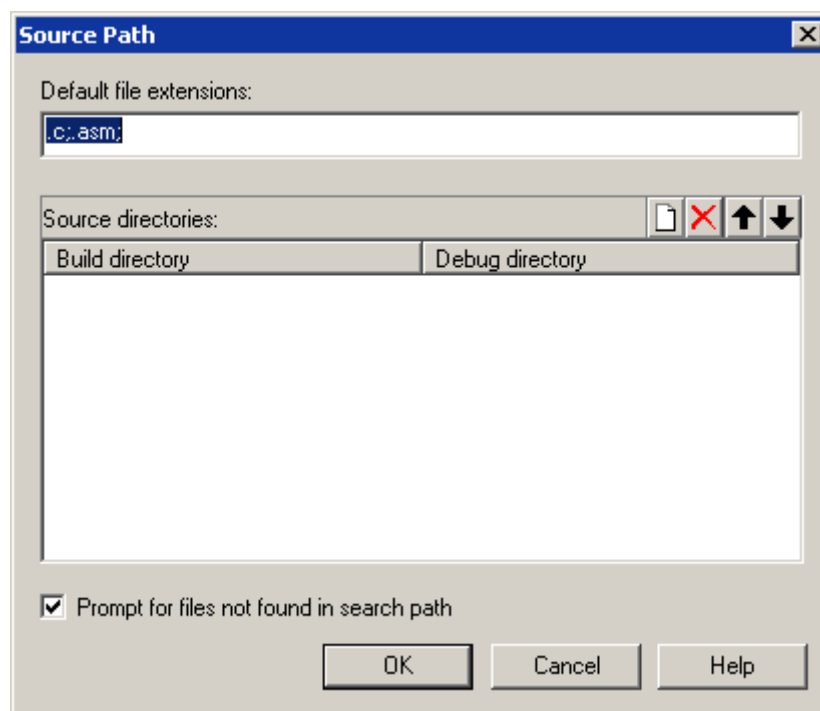
Note: Any file whose signature is not recognized by the loader is treated as binary format.

Pre-Load Macro. The **Browse** button allows you to select a macro file that runs prior to the loading of code or symbols. The primary use of this macro is to automate initialization of your target to a known state prior to the actual program load. If this feature is not desired, then leave the field blank. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly.

Post-Load Macro. The **Browse** button allows you to select a macro file that will run after the loading of code and/or symbols. Some file formats contain information for initializing the processor state after the program is loaded. A file of this format does not require a post-load macro. Other file formats do not contain this initialization information and require further initialization of the processor state after the code has been loaded. The post-load macro is useful for automating this processor state initialization process. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly.

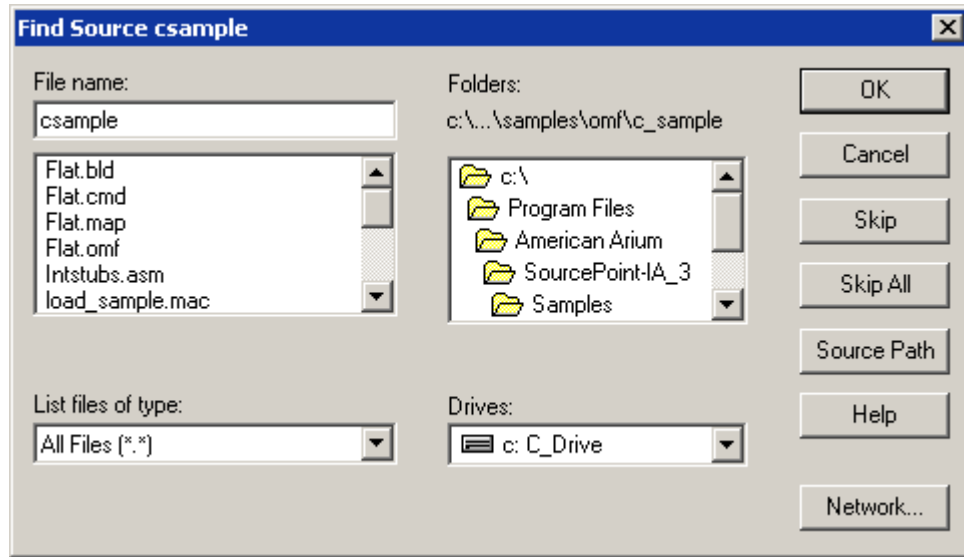
Source path button. Click the **Source path** button to open the **Source Path** dialog box. In it you can specify where the loader should locate source files and what file extensions it should look for. The **Source Path** dialog box now supports both path maps and search paths (for OMF). A path map requires entries under both the **Build directory** and **Debug directory**, while a search path requires only a **Debug directory** entry.

Source Path Dialog



Source Path dialog box

If you enable the **Prompt for files not found in search path**. The **Find Source** dialog box displays, giving you access to your source files via this easy-to-use GUI.



Find Source dialog box

Detect button. When you have an Intel 64-bit processor on your motherboard and you are loading a program file with an EFI(PE) format (.efi extension), a **Detect** button displays. If you click on the button, an EFI base address finder dialog box appears with the appropriate address range within which lies the EFI base address table. Clicking on the **Find** button in the address range dialog box causes the EFI address finder to find the address in the table. The finder then places the address in the **Address** text box in the **Load Program** dialog box.

Load type section

You have three options: Load **Code and symbols**, **Code only**, or **Symbols only**.

- **Code and symbols option:** Use this option to write the program (code) into target memory to load symbolic and source file information into SourcePoint. This allows program addresses to be referenced symbolically, and disassembly to show source code and symbol names.
- **Code only option:** Use this option to write the program into target memory.
- **Symbols only option:** Use this option to load source and symbol information into SourcePoint. Select this option when the program is already in the target (in ROM or Flash). This results in shorter load times.

Initialize processor. Enable this option to set the PC to the entry point location specified in the file you are loading.

Verify. When this option is enabled, SourcePoint verifies that the program you selected to load is the one being loaded.

Reload Program Option

Select **Program Reload** to initiate a load operation using the parameters from the prior program load in the current project without any further intervention. If no program has yet been loaded, the **Program Load** dialog box is displayed.

Remove All Programs Option

This option removes all source or symbol information from the **Symbols** window. It is the same as the **Remove All Programs** option in the **Symbols** window.

Save Program Option

The **Save Program** option lets you save your program. For details on how to save a program, see "[How to Save a Program](#)," part of "How To - SourcePoint Environment," found under *SourcePoint Environment*.

File Menu - Macro Menu Item

Select **File|Macro** to access the following menu items: **Load Macro** and **Configure Macros**.

Load Macro Option

Select the **Load Macro** menu item to load an existing macro. **Load Macro** allows you to load specific macros at your discretion.

1. Select the **Load Macro** menu item.

A **Load Macro** dialog box displays.

2. Click the **Browse** button.

A standard **Open** file dialog box displays.

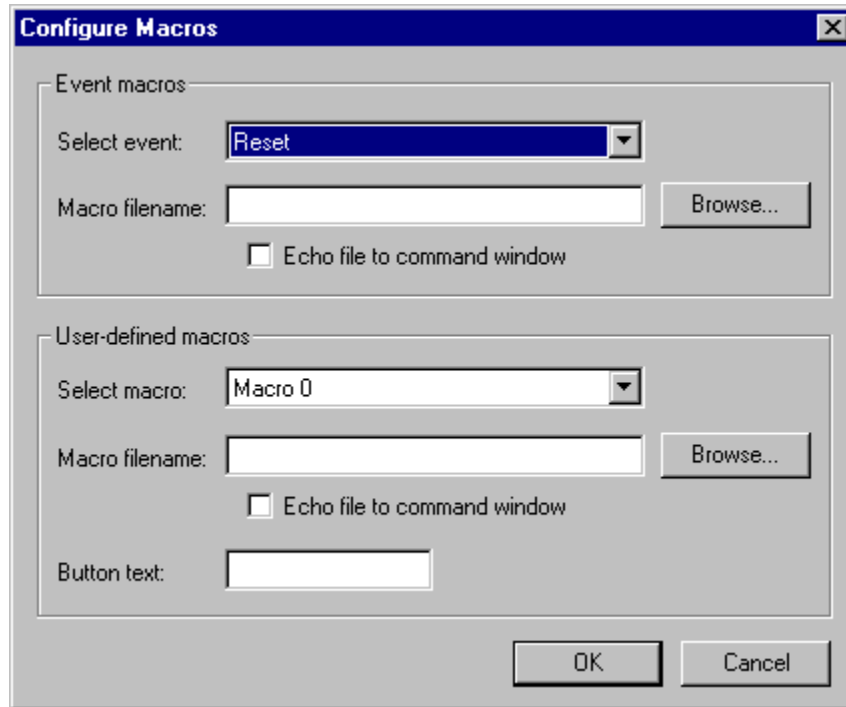
3. Select the desired macro by clicking on it to highlight it.
4. Click the **Open** button.

The **Open** file dialog box closes. The **Load Macro** dialog box redisplay with the selected file listed in the **Filename** text box.

5. To enable the **Echo File to Command Window**, click on the check box. By enabling this option, all commands in the macro file are copied to the SourcePoint **Command** window.
6. Click the **OK** button.

Configure Macros Option

Selecting **Configure Macros** opens a dialog box of the same name. There are two types of macros – event macros and user-defined macros. Each time one of the specified events occurs, the event macro you create and attach to a predetermined name is executed automatically. In addition, up to 10 user-defined macros can be assigned to a macro icon button on the icon toolbar.



Configure Macros dialog box

Event Macros

The **Event macros** section of the dialog box allows you to associate macros with predefined events.

- **Select event.** Choose one of the following events from the text drop down list: **Startup**, **Project load**, **Go**, **Stop**, **Reset**, and **Breakpoint**. The macro associated with this event, if any, is displayed in the **Macro Filename** text box.
- **Macro Filename.** This edit control text box displays a macro associated with an event. A blank filename indicates no macro is associated with the event. The file name can be added to the box via the **Browse** button. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly.
- **Echo file to command window.** Enabling this option causes the selected event macro to be copied to the SourcePoint window.

User-Defined Macros

Select macros. Use this drop down text list to associate a user-defined macro with a macro icon on the toolbar. You can have up to 10 macro icon buttons on the toolbar.

Note: You must add the macro icons to the icon toolbar prior to adding user-defined Macros 4-9. If you do not, the **Select macros** drop down text box reveals only Macros 0-3. To add macros to the toolbar, see "[How to Edit Icon Groups to Customize Your Toolbars](#)," found under "How To - SourcePoint Environment," part of *SourcePoint Environment*.

- **Macro filename.** Use this box to specify the macro to execute. Select the **Browse** button at the bottom of the dialog box to find the file. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly.

- **Macro button text.** You may choose to give your macros names that are more descriptive than “Macro 0” or “Macro 1”, etc. by changing the icon text. The text is visible on the toolbar when you choose to display both the icon and associated text.

Note: See "[How to Configure Custom Macro Icons](#)" and "[How to Configure Autoloading Macros](#)," both found under "How To - SourcePoint Environment," part of *SourcePoint Environment*.

File Menu - Print Menu Items

There are three print menu items in the **File** menu: **Print**, **Print Preview**, and **Print Setup**.

To go directly to the description of one of these menu items, click below:

[Print Menu Item](#)

[Print Preview Menu Item](#)

[Print Setup Menu Item](#)

Print Menu Item

1. Select **File|Print** on the menu bar to print.

A **Range** dialog box displays.

2. Choose to print all or a portion of a selection.

The range may be of one of the following types:

All	All of the data, both visible and not visible, from the currently selected window.
Current Display	The data from the lines currently displayed in the selected window. This is the default unless text has been selected.
Selection	The data that has already been highlighted in a window. Text selection can be accomplished via the keyboard (e.g., shifting the right arrow) or via the mouse (e.g., dragging the mouse over the region while holding down the left mouse button). This is the default when text has been selected. Otherwise, the current display is the default.
"Special"	Specific data can be identified via beginning and ending range information. The unit of measure will vary, depending on the window.

Note: Many of the windows in SourcePoint (but not all) have the capability to print their contents. You are prompted for a range to print. You may want to limit the range of print because the total potentially could be huge.

3. Click on the **OK** button.

The **Range** dialog box closes. A standard Windows® **Print** dialog box displays.

4. Determine print options.
5. Click the **OK** button.

Print Preview Menu Item

Select **File|Print Preview** on the menu bar to view the selected window as it will appear when printed. Once displayed, print setup options are available from the preview screen.

Print Preview initially shows data from the beginning. By using the range selection dialog box at print time, you can start at another location. In the **Print Preview** window, use the **Next Page** and **Prev Page** buttons or the scroll bars to see other potential pages of print.

Print Setup Menu Item

Select **File|Print Setup** on the menu bar to access printer options. Printer selection and other default printer options are specified from this screen.

The print setup varies depending on the print capabilities at your site. Each printer has a number of capabilities that may be used to configure a print environment. The two most common items to change in **Print Setup** dialog box are the printer and the orientation of the image (profile or landscape). While SourcePoint supports many printing devices, the colored window displays and windows that have contents that are wide may not print well on your printer. High resolution PS-capable printers have been found to provide the best output. For wide displays, consider using a landscape orientation.

By changing the print setup parameters, you can change those parameters for all applications that print on that device (not just SourcePoint). If this is a problem, change the parameters each time you print something from within SourcePoint rather than changing the print setup. Modifications made during a particular print job aren't persistent like those made during the print setup.

File Menu - Update Emulator Flash Menu Item

Select **File|Update Emulator Flash** on the menu bar to update the firmware stored in the flash memory of the emulator. A standard file **Open** dialog box displays. The file (with an ".fls" extension) resides in the SourcePoint root directory. This menu item is also available via the icon toolbar.

This step usually is required when upgrading to a newer version of SourcePoint.

For specifics on how to update flash, see, "[How to Update Flash Code](#)," part of "How To - Installation," found under *Installation*.

Program Target Devices Menu Item

Program Flash Option

This option takes you to the **Target Configuration** dialog box. However, programming the target flash is currently not available.

Program PLD Option

Not functional.

File Menu - Other Menu Items

The file menu also contains these menu items: **Save As**, **Recent Projects**, **Recent Programs**, **Recent Macros**, and **Exit**. They are described in more detail below.

[Save As Menu Item](#)

[Recent Layouts Menu Item](#)

[Recent Projects Menu Item](#)

[Recent Programs Menu Item](#)

[Recent Macros Menu Item](#)

[Exit Menu Item](#)

Save As Menu Item

Many of the windows in SourcePoint have the capability to save their content display as text to a file. This capability is invoked via **File|Save As** on the menu bar. The **Save As** dialog box displays.

The dialog box, which works much like any Microsoft® Windows® Save As screen, also has the ability to save specific ranges of contents. In addition, you can save specific ranges of contents. It is necessary to limit the range of output in many cases because the time it takes to save the data can be well into the minutes.

The range may be of one of the following types:

All	All of the data from the currently selected window.
Current Display	The data from the lines currently displayed in the selected window. This is the default setting unless text has been selected.
Selection	If you highlight certain data from a window, the Selection field is enabled. Text selection can be accomplished via the keyboard (e.g., shifting the Right Arrow key) or via the mouse (e.g., dragging the mouse over the region while holding down the left mouse button). This is the default when text has been selected. Otherwise, the current display is the default.
"Special"	Specific data can be identified via beginning and ending range information. The unit of measure will vary, depending on the window.

To help ensure the output text file does not overwrite a file already present, a confirmation dialog box displays.

As the output may take some time, a progress window is shown. The **Cancel** button is available at any time to stop the output at the shown percentage of the range. The output data up to the time of the cancellation can be saved, thus enabling you to start the output of a large range and then change your mind and stop it at any time.

Recent Projects Menu Item

Select **File|Recent Projects** on the menu toolbar. The last nine files ("prj") you most recently loaded display. This list is persistent and cumulative between invocations of SourcePoint. The full path displays if the current directory is not the same as that of the file. If the file path is the same as the current directory, only the name and extension of the file display.

Recent Layouts Menu Item

Select **File|Recent Layouts** for a list of SourcePoint layouts you have developed. The last nine files ("lyt") you most recently loaded display. This list is persistent and cumulative between invocations of SourcePoint. The full path displays if the current directory is not the same as that of the file. If the file path is the same as the current directory, only the name and extension of the file display.

Recent Programs Menu Item

Select **File|Recent Programs** on the menu toolbar. The last nine files you most recently loaded display. This list is persistent and cumulative between invocations of SourcePoint. The full path displays if the current directory is not the same as that of the file. If the file path is the same as the current directory, only the name and extension of the file display.

Recent Macros Menu Item

Select **File|Recent Macros** on the menu toolbar. The last nine files you most recently loaded display. This list is persistent and cumulative between invocations of SourcePoint. The full path displays if the current directory is not the same as that of the file. If the file path is the same as the current directory, only the name and extension of the file display.

Exit Menu Item

Select **File|Exit** on the menu bar to exit SourcePoint.

CAUTION: *If you have disabled **Save Settings on Exit** under the **General** tab of the **Preferences** dialog box and you wish to retain the data and settings in a currently active window or window group, you must save the Project (".prj") file before exiting SourcePoint. Select **File|Save Project** on the menu bar to save the file using the current name and location, or **File|Save Project As** to save it as another file name or location.*

Edit Menu

The **Edit** menu contains **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, **Find**, **Replace**, and **Find Symbol** menu items.

[Undo](#)

[Redo](#)

[Cut, Copy, Paste](#)

[Find](#)

[Replace](#)

[Find Symbol](#)

Undo Menu Item

The **Undo** menu item "undoes" anything you have done immediately before and has numerous uses. For example, if you have added something and you wish you hadn't, you may want to use the **Undo** menu item. If you want to bring back something you have just deleted, use this item. You can "undo" something only once.

Redo Menu Item

The **Redo** menu item lets you "undo" what you have just "undone." For example, if you have deleted something with the **Undo** menu item, you can bring it back in with the **Redo** menu item.

Cut, Copy, Paste Menu Items

The ability to access the **Cut**, **Copy**, and/or **Paste** menu items is conditional on many parameters: you have selected an editable area, the active window can accept the edit, the data selected to cut or paste is compatible with what is being solicited, etc. When the ability to cut, copy, and/or paste is inhibited because it violates one of the above conditions, the corresponding menu item is grayed out, indicating that it is currently not available.

To select a single word of text, place the blinking cursor in the word and double-click the left mouse button. This highlights it. To select a region of text, hold down the left mouse button and drag the mouse across the desired area. When the desired region has been highlighted, release the mouse button.

The selected area appears with the colors inverted (white goes to black, blue goes to yellow, etc.). Additionally, most standard Microsoft Windows selection modes are available.

The SourcePoint **Cut**, **Copy**, and **Paste** operations use the standard Microsoft Windows clipboard so that text can be transferred between SourcePoint windows and dialog boxes as well as between SourcePoint and other applications and editors. Once the selected area has been cut or copied, it can then be pasted in the desired location.

Find Menu Item

Use the **Find** menu item to enter the text to be found and then clicking the **Find Next** button. Options may be selected prior to the search to set the direction of search and set case sensitive constraints.

Replace Menu Item

Use the **Replace** menu item to enter both the text to be found and the replacement text. Case-sensitive constraints are optional and can be selected by clicking the **Match case** check box. The **Replace** and **Replace All** buttons may be used to replace the first occurrence or all occurrences of the find text, respectively.

Find Symbol Menu Item

The **Find Symbol** menu item opens a dialog box that allows you to quickly maneuver and find any program symbol and its memory address. This dialog can be summoned in two additional ways: by selecting a program from within a **Symbols** window **Global** tab and pressing CTRL-F, or by pressing CTRL-S from anywhere within SourcePoint. When it is invoked from the **Symbols** window, the dialog also serves as a finder for symbol tree items in the view.

Find Symbol dialog box

The Dialog Box

The dialog box displays three tabs: **Code**, **Data**, and **Modules**.

Right-clicking on a symbol in the **Code** or **Modules** view brings up a context menu with the following menu items: **Open Code Window**, **Open Memory Window**, **Set Breakpoint**, **Go Until**, and **Add Performance Analysis Range**.

- **Open Code Window/Open Memory Window menu item.** Clicking on these menu items opens a **Code** or **Memory** window at the address of the symbol highlighted in the **Find Symbol** dialog box.
- **Set Breakpoint menu item.** This menu item lets you set a breakpoint at the address of a highlighted symbol.
- **Go Until menu item.** This menu item sets a temporary breakpoint at the symbol and lets the target run.

The context menu that opens from the **Data** view includes two menu items: **Open Memory Window** and **QuickWatch**.

- **Open Memory Window menu item.** Clicking on this menu item opens a **Memory** window at the address of the symbol highlighted in the **Find Symbol** dialog box.
- **QuickWatch menu item.** Clicking on this menu item drops the symbol into a **QuickWatch** view, which then displays the value of the symbol, as well. Keep in mind that a value placed in the **QuickWatch** view is lost at the next **Step** or **Go** command; it is just a handy way to get a quick view of that value.

If you are running a single program, the white text box below the tabs displays the current program. When more than one program is running, the text box is replaced by a drop down list box from which you can select the program you want to view.

View Menu

The **View** menu contains **Toolbars**, **Dialog Bar**, **Breakpoints**, **Code**, **Command**, **Descriptors**, **Devices**, **Log**, **Memory**, **Page Translation**, **PCI Devices**, **Registers**, **Symbols**, **Trace**, **Viewpoint**, and **Watch** menu items. Those items that open a window also are available as icons on the icon toolbar.

Toolbars Menu Item

Select **View|Toolbars** on the menu bar to enable/disable the display of the icon toolbars available in SourcePoint's main window. They are: **File**, **Macro**, **Edit**, **Processor**, **View**, **Window**, **Print**, and **Help**. SourcePoint allows you to customize the toolbars. All icons are enabled by default. (To customize the toolbars, see the "[Edit Icon Groups to Customize Your Toolbars](#)" topic in "How To - SourcePoint Environment" under *SourcePoint Environment*.) Each icon directly corresponds to a menu item located within a menu from the SourcePoint menu bar.

Dialog Bar Menu Item

Several windows (such as **Code** and **Memory** windows) contain an optional dialog bar that allows you to control the range and format of the data displayed in that view. To enable the dialog bar, select **View|Dialog Bar** on the menu bar and enable or disable the option by clicking on it. A check mark by it indicates the option is enabled.

Breakpoints Menu Item

Select **View|Breakpoints** on the menu bar to access the **Breakpoints** window. The **Breakpoints** window displays a list of current breakpoints or events, including their location, sequence, and all specified attributes. The **Breakpoints** window is used to add, edit, disable, enable, and remove breakpoints.

For additional information on breakpoints, begin with the topic, "[Breakpoints Window Introduction.](#)"

Code Menu Item

Select **View|Code** on the menu bar to access the **Code** window and display the **Code** menu. The **Code** window menu duplicates the dialog bar and contains additional menu items that aid in the examination and tracking of program code. The **Code** window is used to view code at a specific address, set breakpoints, run the processor, and track program execution.

For additional information on the **Code** window, begin with the topic, "[Code Window Introduction.](#)"

Command Menu Item

Select **View|Command** on the menu bar to open the **Command** window and to display a **Command** menu.

For additional information regarding the **Command** window, begin with the topic, "[Command Window Introduction.](#)"

Descriptors Menu Item

Go to **View|Descriptors** on the menu bar to open a window displaying the processor descriptor tables.

Note: The target must be in Protected mode in order for the **Descriptors** command to display valid information.

For additional information on the Descriptors window, begin with the topic, "[Descriptors Window Introduction.](#)"

Devices Menu Item

Select **View|Devices** on the menu bar to open the **Devices** window. The **Devices** window allows you to define a grid to view memory-mapped I/O devices and related registers.

For additional information on the **Devices** window, begin with the topic, "[Devices Window Introduction.](#)"

Log Menu Item

Select **View|Log** on the menu bar to access the **Log** window and display the **Log** menu. The **Log** window tracks and logs SourcePoint events such as warnings and errors. The **Log** menu allows for the selection of specific information or events to be logged.

For additional information regarding the **Log** window or the **Log** menu, begin with the topic, "[Log Window Introduction.](#)"

Memory Menu Item

Select **View|Memory** on the menu bar to open the **Memory Address** dialog box. This dialog box prompts you to enter an address with a choice of styles. After an address is entered, a **Memory** menu dialog bar is activated on the **Memory** window, and the **Memory** menu appears on the SourcePoint menu bar. The **Memory** window menu contains menu items to aid in the examination and modification of memory; it also duplicates the dialog bar.

For additional information on the **Memory** window, begin with the topic, "[Memory Window Introduction.](#)"

Operating System [Resources] Menu Item

Select the **Operating System** menu item to debug a Linux target. The two primary windows are the **Operating System Resources** window and the **Target Console** window.

For more information, start with the topic, Linux-Aware Debugging.

PCI Devices Menu Item

Select **View|PCI Devices** on the menu bar or click on the **PCI Devices** icon on the icon toolbar to access the **PCI Devices** window. The **PCI Devices** window displays basic information for the PCI devices on the target. It scans the PCI buses you specify using a process called PCI device enumeration and displays a summary of each PCI device found, ordered by its bus, device, and function numbers.

For additional information on the PCI Devices window, begin with the topic, [PCI Devices Window Introduction.](#)"

Page Translation Menu Item

Select **View|Page Translation** on the menu bar to open a window displaying the processor page translation tables. Page translation tables are used to look at the memory paging features.

For additional information on the **Page Translation** window, begin with the topic, "[Page Translation Window Introduction.](#)"

Registers Menu Item

Select **View|Registers** on the menu bar to open a window that displays the hexadecimal values of the general registers.

For more information about the **Registers** window, begin with the topic, "[Registers Window Introduction.](#)"

Symbols Menu Item

Select **View|Symbols** on the menu bar to access the **Symbols** window. The **Symbols** window displays all symbols and their values by default. You can also choose to display their types and addresses.

For additional information on the **Symbols** window, begin with the topic, "[Symbols Window Introduction.](#)"

Trace Menu Item

Select **View|Trace** on the menu bar to open the **Trace** window. The **Trace** window provides a record of processor events that can be used to determine the exact path of the executed software. This menu item is grayed out when the connected base unit does not include trace functionality.

For additional information on the **Trace** window, begin with the topic, "[Trace Window Introduction.](#)"

Viewpoint Menu Item

Select **View|Viewpoint** on the menu bar to open a window showing the state of each processor in the target system. The window opened also allows viewpoint selection among the target processors. The command is available on multi-processing targets.

Watch Menu Item

Select the **Watch** menu item to open a window into which you can put user-selected symbols. Once placed in the window, their values are displayed. Symbol values change in the **Watch** window as the values themselves change.

For more information on the **Watch** window, start with the topic, "[Watch Window Introduction.](#)", part of "Watch Window Overview," found under *Watch Window*.

Processor Menu

Items in the **Processor** menu let you "step through" source or assembly code in various ways. The menu contains **Go**, **Stop**, **Step Into**, **Step Over**, **Step Out Of**, **Reset**, and **Snapshot** menu items. These are described in detail below.

Note: For more information on stepping, see the topic entitled, "[Stepping](#)" found under *Technical Notes*.

Go Menu Item

Select **Processor|Go** on the menu bar to start program execution at the current instruction pointer (IP). The processor stops when a breakpoint is encountered. If no breakpoints are set, the processor is stopped by executing the **Stop** menu item.

Stop Menu Item

Select **Processor|Stop** on the menu bar to halt the processor.

Step Into Menu Item

This single-steps the next instruction in the program and enters each function call that is encountered. This is useful for detailed analysis of all execution paths in a program.

Step Over Menu Item

This single-steps the next instruction in the program and runs through each function call that is encountered without showing the steps in the function. This is useful for analysis of the current routine while skipping the details of called routines.

Step Out Of Menu Item

Step Out Of causes the processor to run until it comes to the end of the current subroutine and returns to the next high level of the call stack. This is useful as a quick way to get back to the parent routine.

Reset Menu Item

Select the **Reset** menu item to reset the processor(s).

Snapshot Menu Item

Select **Snapshot** on the menu bar or icon bar to enable this menu item. When this item is enabled and the target is running, the target is stopped, all windows are refreshed, and the target is restarted. If the target is not running, no action occurs.

Options Menu

For easier navigation, we have broken the subjects covered in this menu into several topics.
Please click on the hyperlinked text below for documentation on a specific **Options** menu item.

[Preferences](#)

[Target Configuration](#)

[Load Target Configuration File](#)

[Save Target Configuration File](#)

[Emulator Configuration](#)

[Emulator Connection](#)

[Emulator Reset](#)

[Confidence Tests](#)

Options Menu - Preferences Menu Item

To set, change, or modify SourcePoint preferences, select **Options|Preferences** on the menu bar. The **Preferences** dialog box displays with the following tabs displayed: **General**, **Emulator**, **Breakpoints**, **Code**, **Memory**, **Program**, and **Colors**.

To go directly to a tab, click on the link below.

[General tab](#)

[Emulator tab](#)

[Breakpoints tab](#)

[Code tab](#)

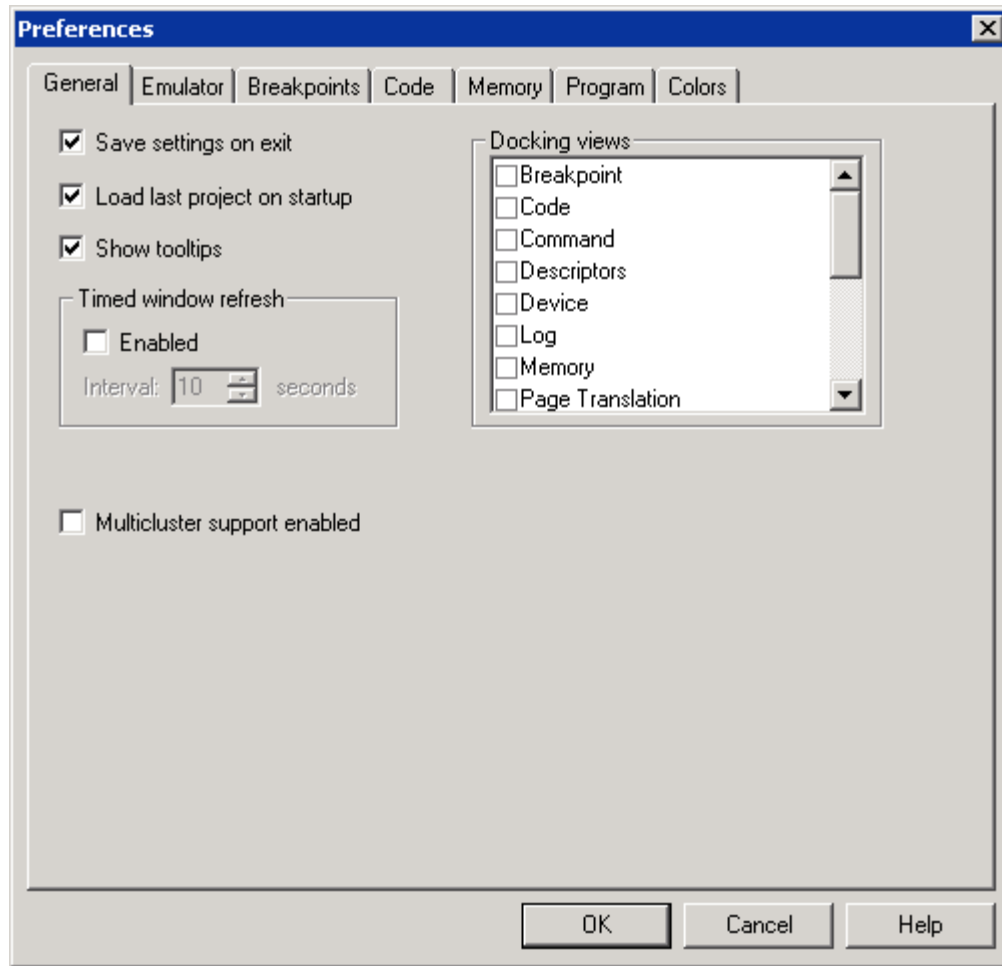
[Memory tab](#)

[Program tab](#)

[Colors tab](#)

General Tab

The section under the **General** tab contains options that apply to all of SourcePoint. These are: **Save settings on exit**, **Load last project on startup**, **Show tooltips**, **Timed window refresh**, **Docking views**, and **Multicluster support enabled**.



General tab under **Options/Preferences**

Save settings on exit. This option determines whether or not to save all settings when SourcePoint terminates. By default, it is enabled.

CAUTION: If you have disabled **Save settings on exit** and you wish to retain the data and settings in a currently active window or window group, you must save the Project ("prj") file before exiting SourcePoint. Select **File|Save Project** on the menu bar or click on the appropriate icon on the icon toolbar to save the file using the current name and location, or **File|Save Project As** (via the menu bar or the icon toolbar) to save it as another file name or location.

Load last project on startup. This option determines whether the project you worked on last is automatically loaded again at startup. By default, the option is enabled.

Change viewpoint to the processor that causes the break automatically. This option is enabled only for multi-processor targets. It causes the **Viewpoint** window to switch automatically to a particular processor if that processor reaches a breakpoint.

Note: This option is available only if you are attached to a multi-processor target.

Show tooltips. This option enables flyover help and is enabled by default.

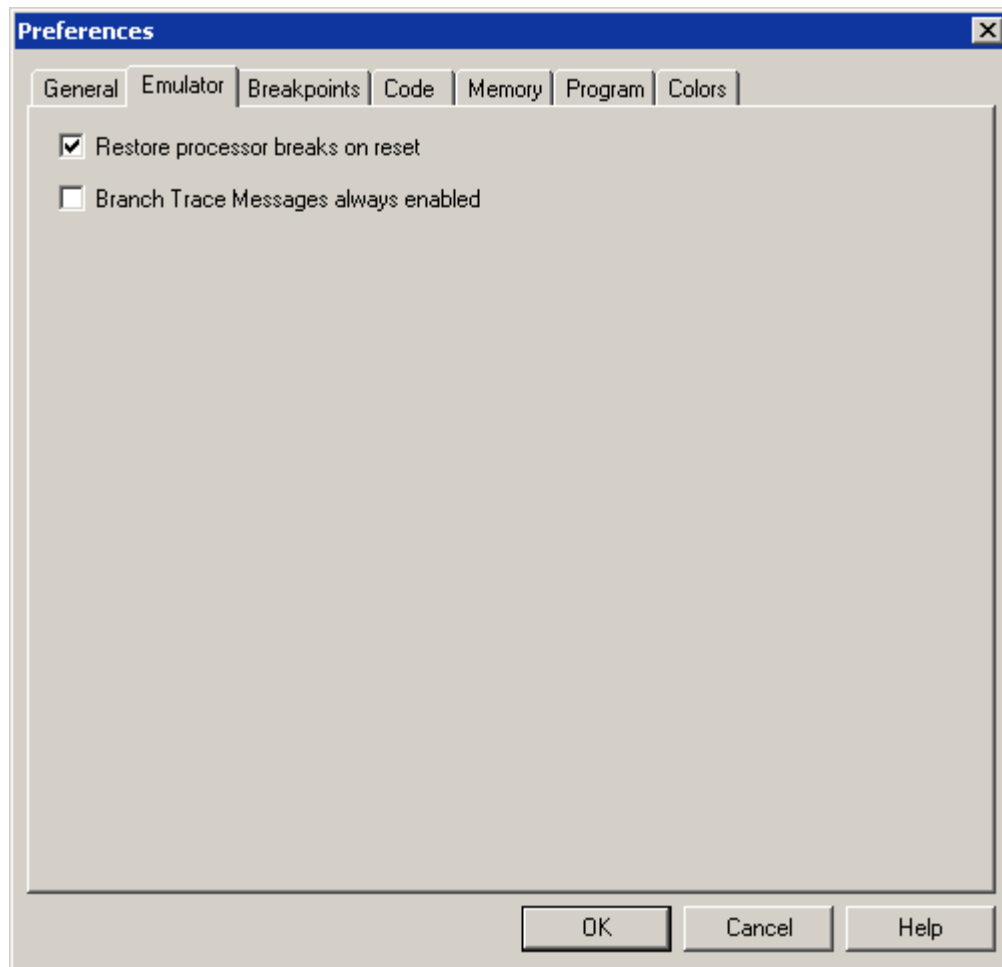
Timed window refresh section. When this option is enabled, all windows are refreshed every n seconds. In the **Interval** text box, you can specify values between 1-999 seconds. The default value is 10 seconds.

Docking views. This option lets you set default docked windows.

Multiclustur support enabled. Put a check mark beside this option if you want to enable multiclustur support.

Emulator Tab

The **Emulator** tab offers three options: **Restore processor breaks on reset** and **Branch Trace Messages always enabled**.



Emulator tab under Options|Preferences

Restore processor breaks on reset. Select this option to restore, upon target reset, the listed processor resource breakpoints and then start the processor. Clearing this option results in a loss of all processor resource breakpoints if a target reset occurs while the processor is running.

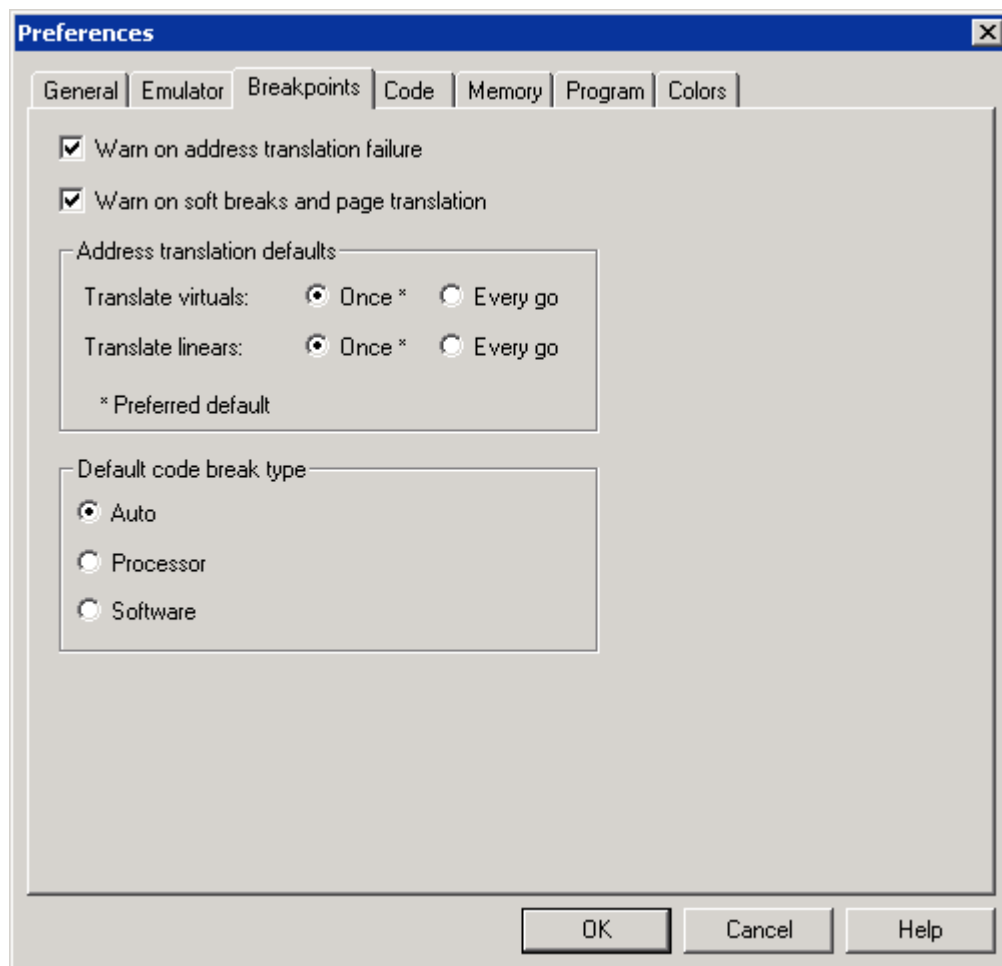
Note: A break on a reset breakpoint performs a similar function, stopping the processor upon reset and restoring its breakpoints when started. However, it does not automatically restart the processor.

Branch Trace Messages always enabled. BTMs allow disassembly of trace. When this option is enabled, each time the processor is reset (while executing code or from a SourcePoint command) the BTM control inside the processor is set and the processor emits BTMs while executing, allowing disassembly of the trace. Disable this option if the emulator has no trace or to disable BTMs.

Note: Additional options may display in this dialog box, depending on whether the target is multi-processor (a processor control list box displays) and whether it has an AMD chip (a cache control groupbox is displayed).

Breakpoints Tab

The **Breakpoints** tab displays warning and address translation options.



Breakpoints tab under Options/Preferences

Warn on address translation failure. When this option is enabled, a warning message is displayed whenever a breakpoint address cannot be translated.

Warn on soft breaks and page translations. The use of soft breaks in a system with **Page Translation** enabled is not guaranteed to work. Prior to starting the processor, SourcePoint writes any soft breaks into target memory. When **Page Translation** is enabled, the page containing the soft break may get swapped out and then back in, thus losing the soft break. When this option is selected, a warning message is displayed whenever the processor is started, at least one soft break has been set, and the processor has **Page Translation** enabled.

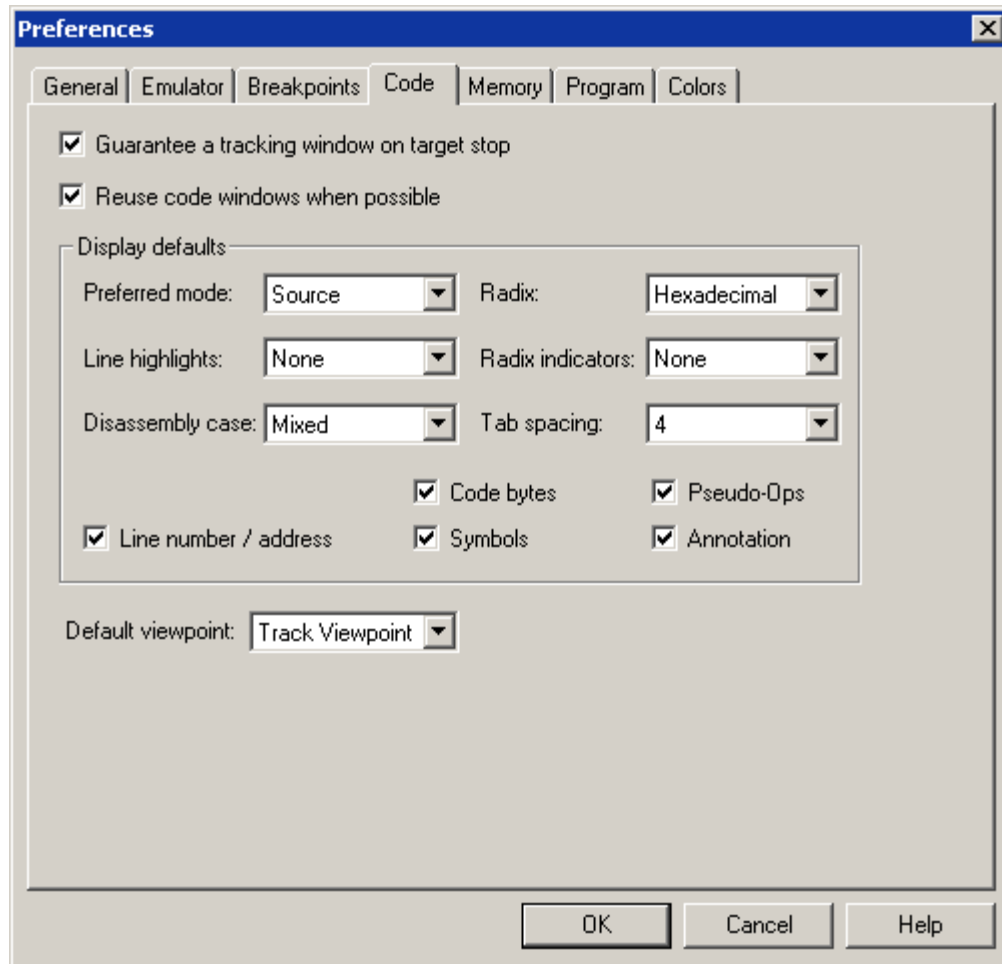
Address translation defaults section. There are two types of breakpoint address translation options: **Translate virtuals** and **Translate linears**. You can choose **Once**, where the breakpoint address is translated immediately using the current processor context, or **Every Go**, where the address is re-translated every time the processor is started. By default, SourcePoint translates virtual and linear addresses once. This **Address translation defaults** section allows these defaults to be overridden. In addition, individual breakpoints can have their translation types changed from within the **Breakpoints** window.

Default code break type section. The default is **Auto**. When **Auto** is selected, for a breakpoint explicitly set in code (such as through the **Code** window), a software breakpoint is used if one is available; otherwise, a processor breakpoint is used if one is available. For temporary breakpoints implicitly set in code (as on a go til address or source level step), a processor breakpoint is used if one is available; otherwise, a software breakpoint is used if one is available. Selecting the **Processor** option specifies the setting of only processor breakpoints. Selecting the **Processor** option specifies the setting of only processor breakpoints. When using a processor breakpoint type, you can only set two breakpoints. Where appropriate, however, you may wish to set the **Software** option as your default since you can set unlimited software breakpoints. Note that when the target is running in Monitor mode, software breakpoints are not available. For all cases, when no resources are available to set a breakpoint, an error message results.

SMM Break Table section. Currently not available.

Code Tab

The section under the **Code** tab contains options that apply to the **Code** window.



Code tab under Options/Preferences (multi-processor target)

Guarantee a tracking window on target stop. If this option is enabled, every time the target system stops, SourcePoint guarantees that a tracking **Code** window opens for the focus processor. SourcePoint may reuse either an existing **Code** window or create a new one. This is the default behavior.

Reuse code windows when possible. If this option is enabled, SourcePoint attempts to reuse existing **Code** windows rather than create new ones. This applies to **Code** windows that may be generated by the following: **Open Code Window** from the context menu or the **Symbols**, **Trace**, or **Breakpoints** windows, or by **Guarantee a tracking window on target stop**, the option described just above. This option is enabled by default.

Display Defaults. There are a number of options you can set in this field. They are described briefly below.

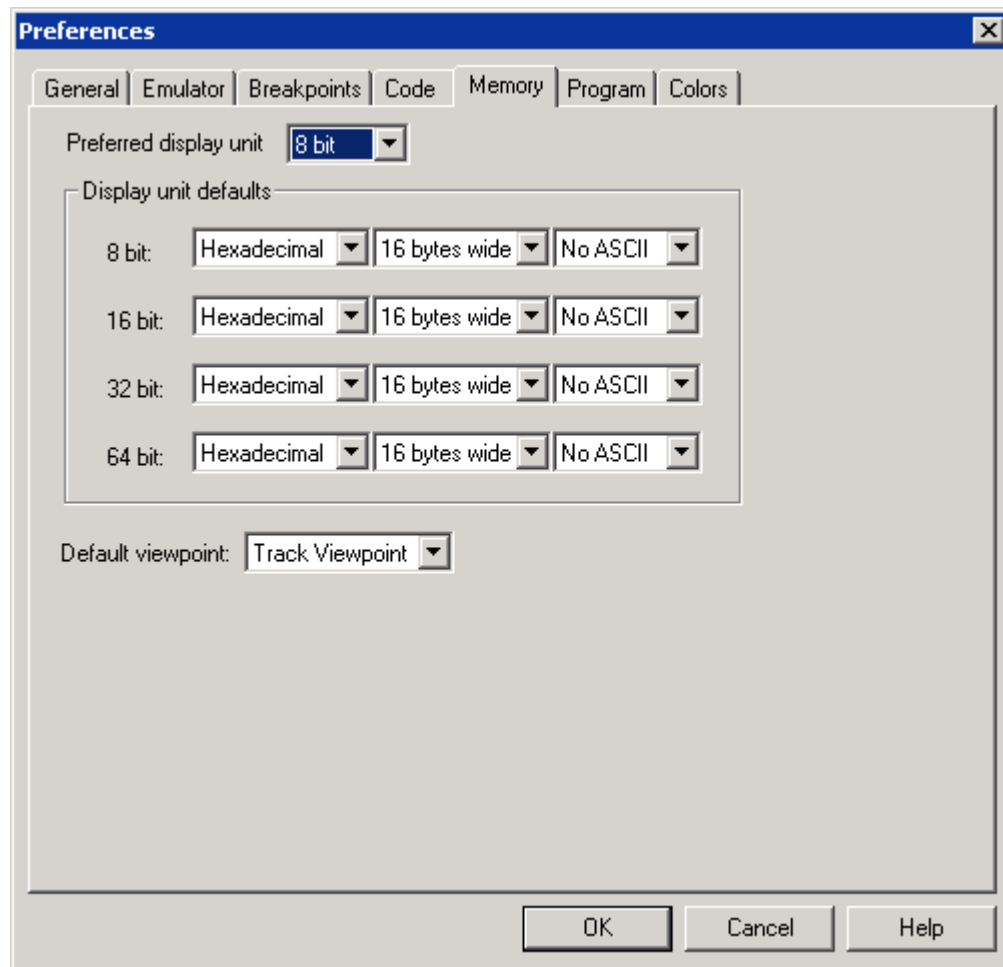
- **Preferred mode.** Choices are **Disassembly**, **Mixed**, and **Source**.
- **Line highlights.** Options are **Group**, **Current IP**, and **None**.
- **Disassembly case.** Options are **Mixed**, **Upper**, and **Lower**.
- **Radix.** Options are **Hexadecimal**, **Octal**, and **Decimal**.
- **Radix indicators.** Options are **Prefix**, **Suffix**, and **None**.
- **Tab spacing.** Allows you to modify tab spacing in the **Code** window.

- **Line number/address.** Displays line number and/or address of code.
- **Code bytes.** Display raw data values of code.
- **Symbols.** Display symbols, also known as labels.
- **Pseudo-Ops.** Pseudo-Ops are mnemonics such as register or instruction names.
- **Annotation.** Enables display of source code comments. All annotated lines have a line of underscores before and after the annotated text.

Default viewpoint. Lets you choose the default processor you want to track. This option displays only when you are connected to a multi-processor system.

Memory Tab

The **Memory** tab provides default display options for the **Memory** window.



Memory tab under Options/Preferences

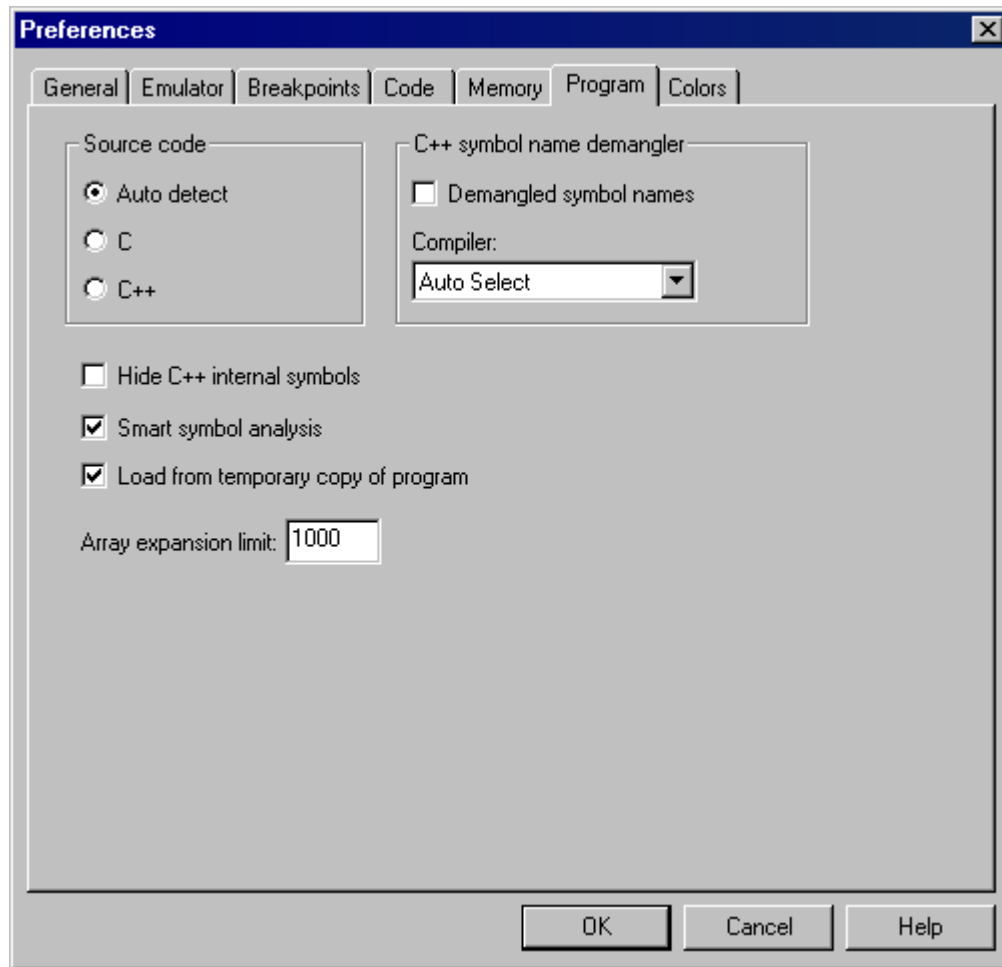
Preferred display unit. Determines the default units for display of memory data (8-, 16-, 32-, or 64-bit).

Display unit defaults section. This section allows you to set the preferences for each Display mode. The options for each are for the **Radix (hex, sign, unsign)**, **Display width (in bytes)** and **ASCII/No ASCII**.

Default viewpoint. Lets you display the default viewpoint you want to track if you are in a multi-processor configuration. In a single processor configuration, the default is **Track Viewpoint**.

Program Tab

The **Program tab** offers options that control the display of code (source and disassembly), including source code type and view, demangling of symbol names, and program caching.



Program tab under Options/Preferences

Note: Many of the options in this dialog box do not take effect until you have reloaded the symbols portion of the file. To do this, select **File|Reload Program**. In the lower right quadrant of the dialog box is the option **Symbols only**. Enable the check box and click on the **Load** button to reload the symbols with the new setting in effect.

Source code section. For correct symbol analysis, SourcePoint needs to know the language in which the source code has been written. This usually can be determined automatically, but you may want to specify the language. This field offers three options: **Autodetect**, **C**, or **C++**. The default setting is **Autodetect**. However, you may specify whether you want to view your symbols as if the source code was C or C++ . This may be useful, say, if your source code was written in C but compiled using a C++ compiler.

C++ symbol name demangler section. As the name implies, this section addresses symbol name demangling.

- **Demangled symbol names.** When enabled, this option demangles symbol names. When working in C++, enable this option.

Note: The **Demangled symbol names** option, when enabled, is available immediately. You do not have to reload the symbols portion of the file before it becomes active.

- **Compiler.** SourcePoint needs to know the compiler used to create your binary. You may choose **Auto Select** or one of a list of compilers from the drop down text box.

Hide C++ internal symbols. This option is self-explanatory. Enabling the box hides C++ internal symbols.

Smart symbol analysis. When this option is enabled, SourcePoint loads program symbols as they are required ("just-in-time" symbol loading). This prevents the long delays that would otherwise occur if SourcePoint attempted to load all symbols at once. With Smart Symbol Analysis enabled, some SourcePoint views (such as the tree view of the **Symbols** window) occasionally may show less symbolic information because SourcePoint has not yet analyzed all symbol data. If you want to ensure that all symbolic information is always available, disable **Smart Symbol Analysis**. This forces SourcePoint to load all symbolic information at program load time. While disabling **Smart Symbol Analysis** may provide more complete symbolic information, this setting can increase significantly program load time.

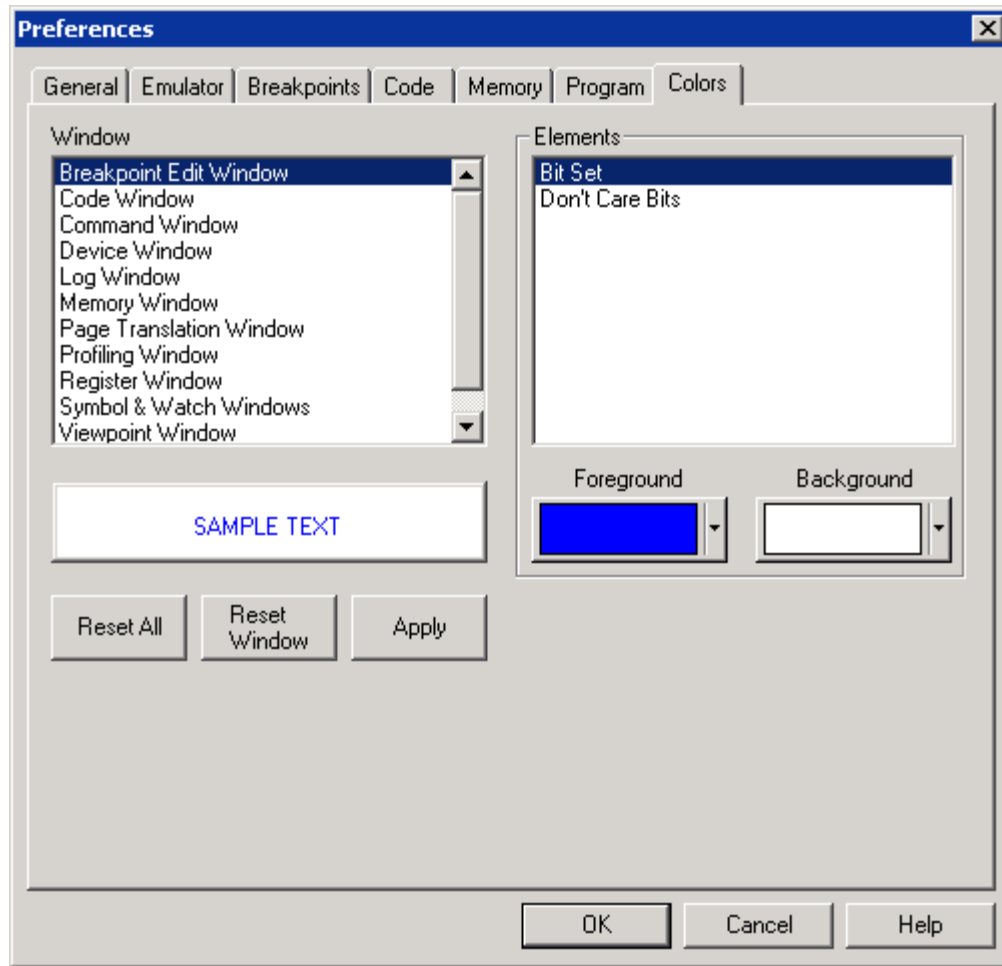
Load from temporary copy of program. Enabling this option causes SourcePoint to make a copy of your program file and load the copy. This frees up your original file for other uses.

Array expansion limit. If your program has very large arrays, you may not want SourcePoint to expand them fully. The **Array expansion limit** option lets you set a threshold.

Colors Tab

The **Colors** tab allows you to change the display colors for various SourcePoint windows.

Note: It is recommended that you be consistent with the choice of background colors.



Colors tab under Options/Preferences

Window text box. Allows you to select the window in which you want to change the colors.

Elements text box. Allows you to select the element in the window whose color you want to change.

Foreground button. Allows you to select a new foreground color for the currently selected element.

Background button. Allows you to select a new background color for the currently selected element.

Reset All button. This button allows you to reset all the windows' colors back to the SourcePoint default colors.

Reset Window button. Allows you to reset all the colors for the currently selected window back to the SourcePoint default colors.

Apply button. Allows you to apply the colors to any window currently displayed

Options Menu - Target Configuration Menu Item

Select **Options|Target Configuration** to open the target configuration dialog.

To move directly to a particular tab, click here:

[Memory Map Tab](#)

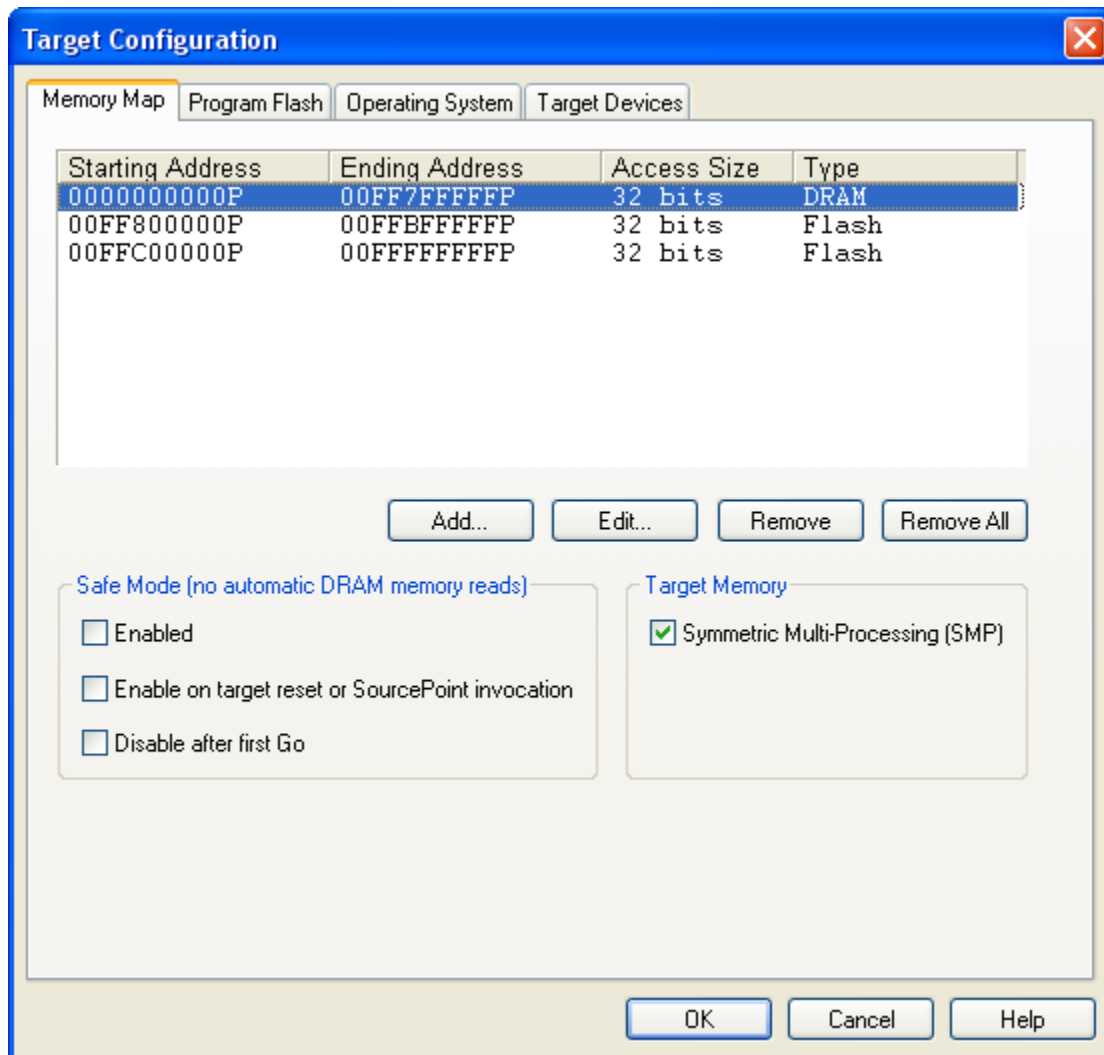
[Program Flash Tab](#)

[Operating System Tab](#)

[Target Devices Tab](#)

Memory Map Tab

The **Memory Map** tab allows you to define regions of memory and control how those regions are accessed by SourcePoint.



Memory Map tab under ***Options/Target Configuration***

Memory Map text box. The upper half of this tab displays already defined memory map ranges, providing four columns of data labeled **Starting Address**, **Ending Address**, **Access Size**, and **Type**. These columns are described briefly below:

- **Starting address column.** This column lists the physical address where the memory range begins.
- **Ending address column.** This column lists the physical address where the memory range ends.
- **Access Size column.** This column lists the physical memory width (8, 16, or 32 bits) that is used when memory within this range is read or written to.
- **Type column.** This column lists the type of memory: **SRAM**, **DRAM**, **ROM**, or **Flash**.

Buttons. The four buttons beneath the text box let you add, modify, or delete the data in the **Memory Map** text box.

- **Add button.** Opens an **Add Memory Map Entry** dialog box for use in adding a memory range.
- **Edit button.** Opens an **Edit Memory Map Entry** dialog box for use in editing a memory range.
- **Remove button.** Removes a highlighted memory range.
- **Remove All button.** Removes all memory ranges.

For more information on how to create or edit these data, see "How to Modify a Defined Memory Region," part of "How To - SourcePoint Environment," found under *SourcePoint Environment*.

Safe Mode (no automatic memory reads) section. The options in this section let you determine the parameters for entering Safe Mode.

Note: Normally, SourcePoint automatically refreshes memory-based windows by re-reading target memory after the target stops, steps, or resets. In some targets, however, reading memory immediately following a reset hangs the target processor. For instance, if a **Memory** window is open and the memory displayed is in an area that is unavailable until the chipset is initialized, then clicking the **Reset** icon hangs the target. This is also a potential problem with the **Code**, **Memory**, **Trace**, **Page Translation**, and **Devices** windows (all windows that can cause target memory reads).

- **Enabled.** If the **Enabled** option is checked, then Safe mode is enabled and automatic refresh of memory-based windows is disabled. When Safe mode is enabled, SourcePoint displays the text "Safe mode)" in the SourcePoint title bar.
- **Enable on target reset or SourcePoint invocation.** If this option is checked, Safe mode is enabled automatically on target reset or SourcePoint invocation, and automatic refresh of memory reads is disabled.
- **Disable after first Go.** This option automatically disables Safe mode following a target run.

If all three check boxes are checked, Safe mode is enabled upon target reset, but it is disabled again when the next **Go** command is issued by the user. This gives you a convenient way to avoid the hazard of windows that cannot be refreshed safely immediately following a target list.

If Safe mode is in effect for a memory range, and that range currently is displayed in a window, the following occurs:

- A **Code** window displays a **No data available** message.
- A **Memory** window displays question marks instead of data.
- Other memory-based windows display old data.

The **Refresh** button of a window always forces memory reads to occur for the data range in that window.

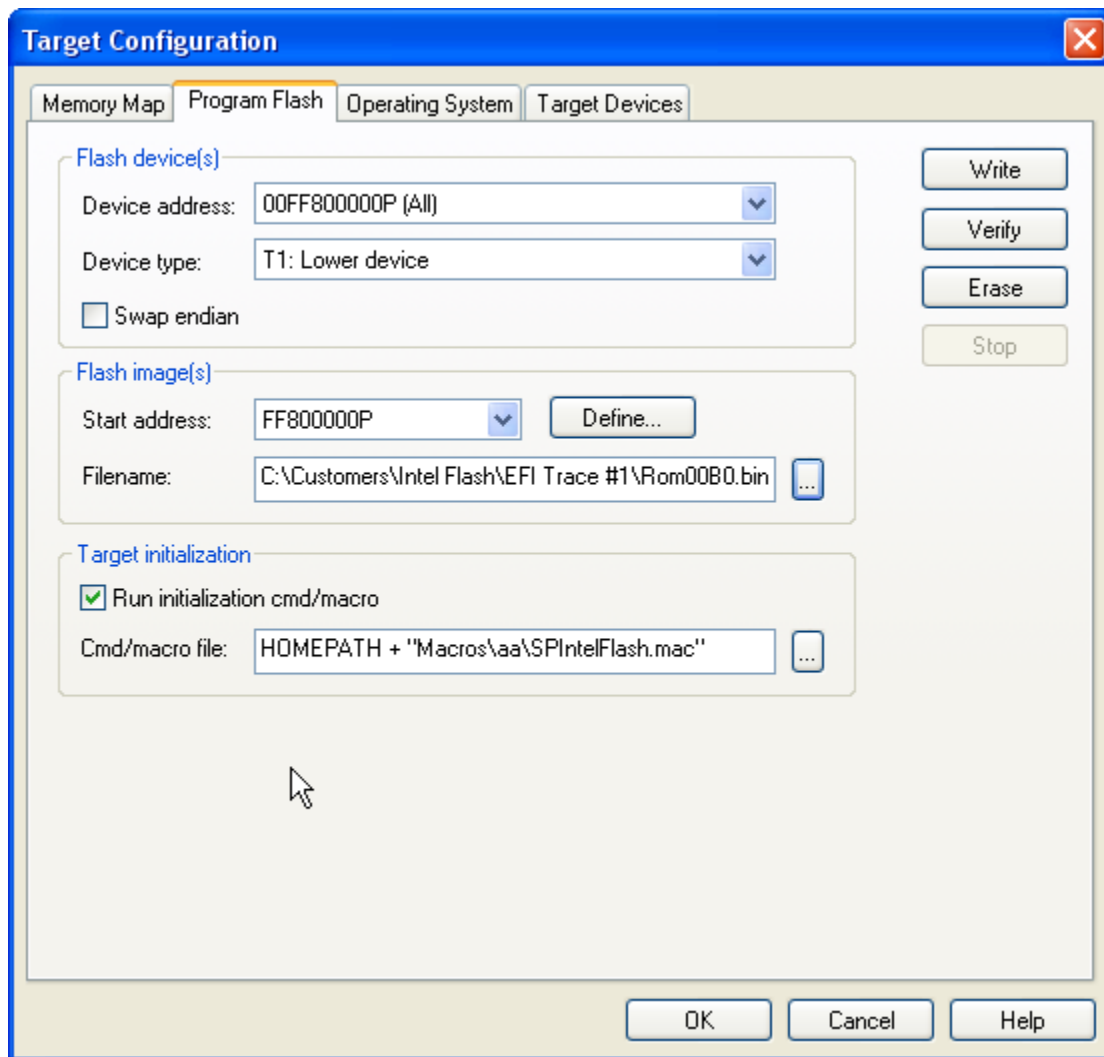
Target Memory section. If you are working in a multi-processing setup, a **Target Memory** section displays to the right of the **Safe Mode** section. The section contains the option **Symmetric Multi-Processing (SMP)**. A Symmetric Multi-Processing (SMP) system is a multi-processor system in which the memory maps of all processors are identical. In other words, all memory is available to all processors at exactly the same address. Check this box if your target is an SMP system.

If the SMP box is not checked, SourcePoint adds a Processor ID column to the **Memory Map** text box so that you can declare memory ranges for each processor independently. A memory range may belong to a single processor or all processors. The concept of a range of memory being shared by some processors, but not all processors, is not supported.

In single processor systems, the SMP check box does not display.

Program Flash Tab

The **Program Flash** feature allows you to program the flash device(s) on a target platform. You must specify a binary file containing the data to be programmed and can also specify a target initialization macro to perform any target initialization that may be required before programming the flash device.



Program Flash tab under **Options|Target Configuration**

Flash Device(s) Section

Device address. Select the correct device address from the drop down list populated from the memory map.

Device type. The **Device type** drop down box contains a list of all supported devices. Use the drop down box to select one.

Swap Endian. The purpose of this check box is to allow you to program an image that is backwards in endianness relative to the target. If you have a big endian target and wish to use a little endian image or visa verse, you can enable the Swap endian option. You may want to swap endianness, depending on how your target processor handles byte storage.

Flash Image(s) Section

Start address. If you want to select a previously defined address, use the drop down box to select one. If you want to define a new start address, click the **Define** button. This opens the **Define Flash Image Start Address** dialog box. Key in the address there.

Note: The start address is NOT a relative offset. This option allows you to program a specific block/sector within the flash device.

Filename. Enter the flash image file name or click on the **Browse** button to select a stored file.

Target Initialization Section

Run initialization/cmd macro. Enable this option to run the initialization macro.

Cmd/Macro File. Enter the name of the macro to be executed before a flash operation occurs or click on the **Browse** button to find it.

Buttons

Write, Verify, Erase, Stop buttons. To execute the macro, click on one of the first three buttons.

The **Write** button programs the selected flash device.

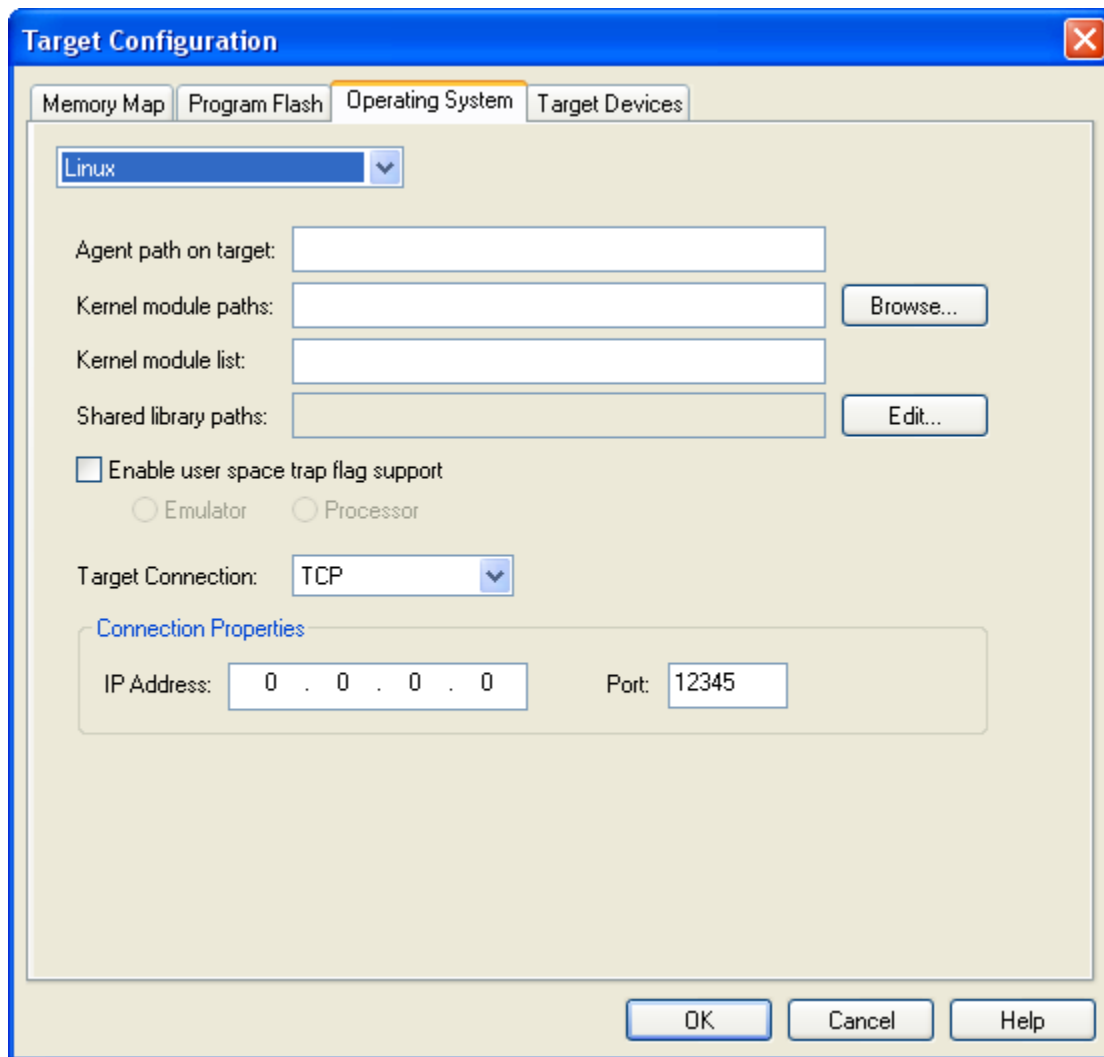
The **Verify** button verifies that the selected flash device is programmed correctly.

The **Erase** button erases the selected flash device.

Use the Stop button to terminate any operation that is currently in process.

Operating System Tab

The **Operating System** tab is designed primarily for targets running a Linux operating system. The options in the tab help to optimize communications between a Linux target and the debugger.



Operating System tab under Options|Target Configuration

Agent path on target. Specifies the full path (in the target file system) to the directory containing the on-target debugging agent (dccwrap and gdbserver). If this path information is included in the normal search path on the target (\$path environment variable), then this field can be left blank.

Kernel module paths. Specifies, for kernel modules, the list of directories on the host system to be searched when symbols are loaded. Multiple paths are delimited with commas. This list is updated automatically whenever you are prompted to browse for symbols during loading of kernel modules.

Kernel module list. A comma delimited list showing which kernel modules you wish to debug. If this list is blank, then all kernel modules are loaded for debugging.

Shared library paths. A list of host paths, delimited by commas, to be searched for shared library symbols files. This list is updated automatically whenever you are prompted to browse for symbols during loading of shared libraries. The **Edit** button lets you edit these paths.

Enable user space trap flag support. The Linux kernel and debug utilities use several IA-32 debug assets, including the trap flag in the EFLAGS register. In order to function correctly when a jtag debugger is attached and breakpoints are set in the kernel, a special trap flag configuration is required.

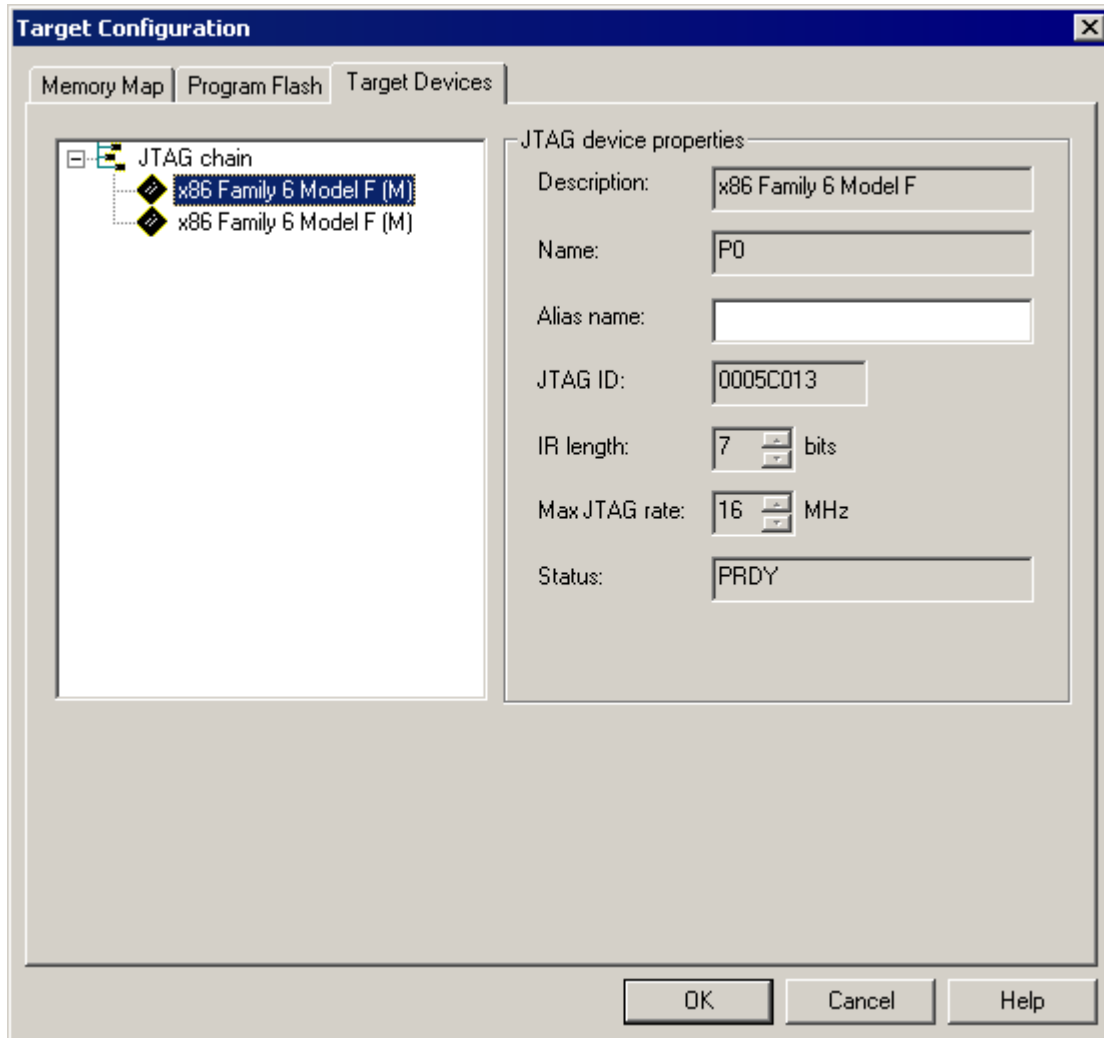
Emulator. Setting this parameter enables the emulator to manage the trap flag such that a dbserver can be used concurrently with jtag debugging. This must be set when performing application debug and any type of breakpoints are set in the kernel.

Target Connection. TCP selects the TCP/IP connection protocol to the target. This is the only protocol currently supported.

Connection Properties. IP Address: The IP address of the target. Port: The base port to run the current instance of the target debug agent (adbserver).

Target Devices Tab

This tab displays information about the target JTAG chain.



Target Devices tab under **Options/Target Configuration**. Note the **JTAG chain properties** section.

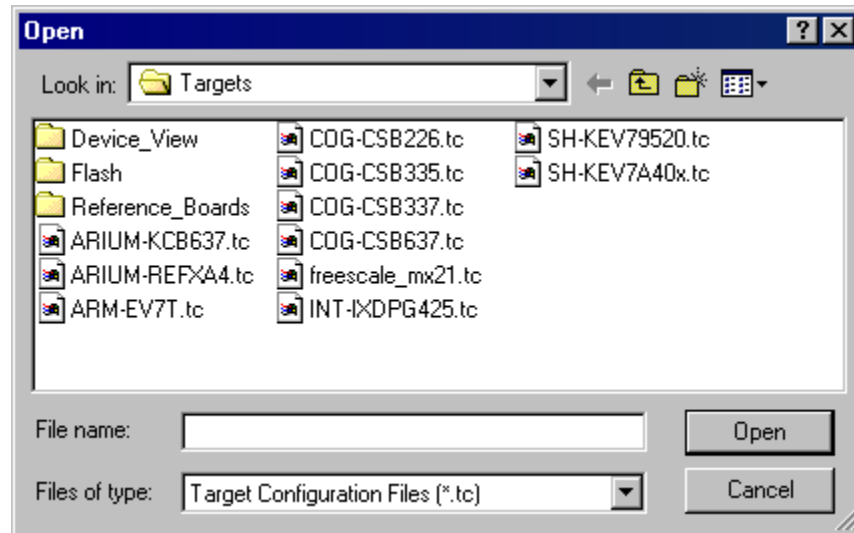
Selecting an item in the tree on the left displays its properties on the right. For the most part, the information is read-only and cannot be modified. However, if SourcePoint does not recognize the JTAG ID of the device, the JTAG properties section includes a **Description** with a drop-down text box from which you can select a processor type. An **IR length** and **Max JTAG rate** spin controls also become editable. Aliases can be added here. If an alias has already been created, it can be edited here.

- **Description.** Specifies the target's core/processor.
- **Name.** This is SourcePoint's "name" for the processor (P0, P1, P2, etc.).
- **Alias name.** Specifies an alias for the device. For instance, P0 could be aliased as BOOT. This alias can then be used throughout SourcePoint where P0 would normally be used.
- **JTAG ID.** Specifies the JTAG ID.
- **IR length.** Specifies the JTAG instruction register in bits.
- **Max JTAG rate.** Indicates the maximum JTAG clock rate.
- **Status.** Specifies the status of the device.

Note: Not all controls in the properties section are displayed, depending on the device.

Options Menu - Load Target Configuration File Menu Item

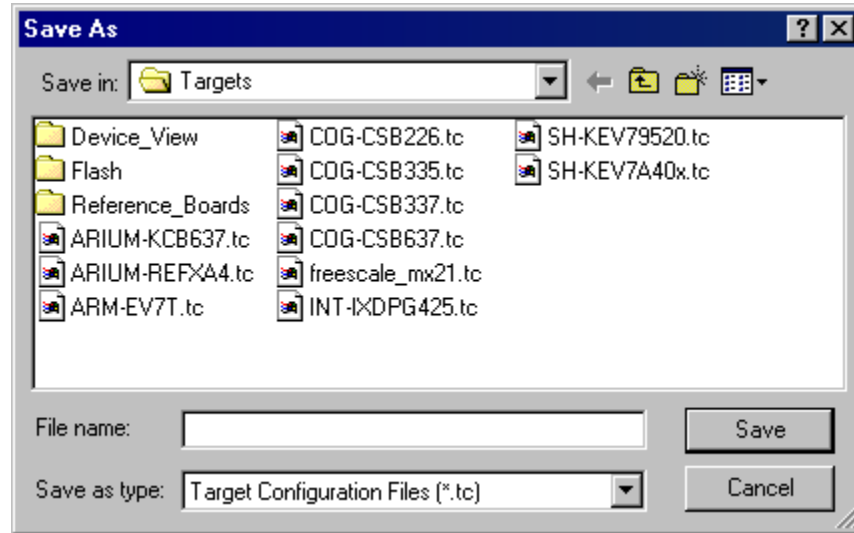
To load a target configuration file, click on the **Load Target Configuration File** menu item in the **Options** menu. Select the file you want to load.



Use this dialog box to load a target configuration file

Option Menu - Save Target Configuration File Menu Item

To save a target configuration file, click on the **Save Target Configuration File** menu item in the **Options** menu. Save the file.



Use this dialog box to save a target configuration file

Options Menu - Emulator Configuration Menu Item

The initial host/emulator/target setup is managed by the emulator. Certain defaults are pre-set for optimum communications with the target. You may find, however, that these settings may not be optimum for your setup. One of the ways to make changes is through the **Emulator Configuration** menu item, which allows you to change certain signaling parameters. Once SourcePoint is running, you can access this menu item and make your changes; after the emulator is reset, it remembers the changes and use them as defaults.

Select **Options|Emulator Configuration** on the menu bar. One of two **Emulator Configuration** dialog boxes displays, depending on the attached emulator.

If you are configuring an emulator that supports an ethernet connection, an **Emulator Configuration** dialog box displaying several tabs appears.

Note: All dialog boxes include a **Description** field at the bottom. If no particular field is selected in a dialog box, the **Description** field gives you a brief description of the dialog box itself. If a particular field is selected, the **Description** field gives you a brief description of that field.

[General Tab](#)

[JTAG Tab](#)

[JTAG Clock Tab](#)

[Target Reset Tab](#)

[ECM-XDP\(3\)Tab](#)

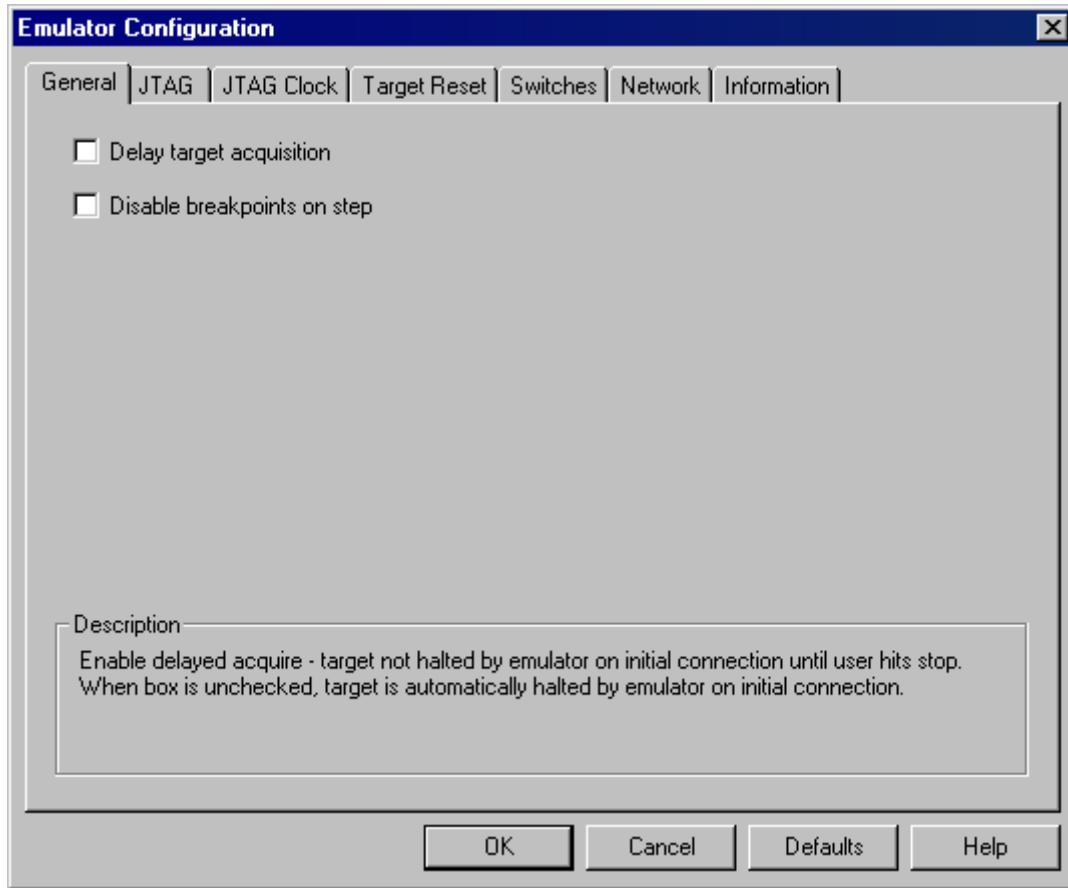
[Switches Tab](#)

[Network Tab](#)

[Information Tab](#)

General Tab

The **General** tab lets you delay target acquisition or disable breakpoints while stepping.



General tab under Options|Emulator Configuration

Delay target acquisition. When this option is enabled, the target is not halted on initial connection. It will not be halted until you click on the stop icon on the icon toolbar or select **Processor|Stop** from the menu toolbar. When this option is disabled, the target is halted automatically by the emulator on initial connection.

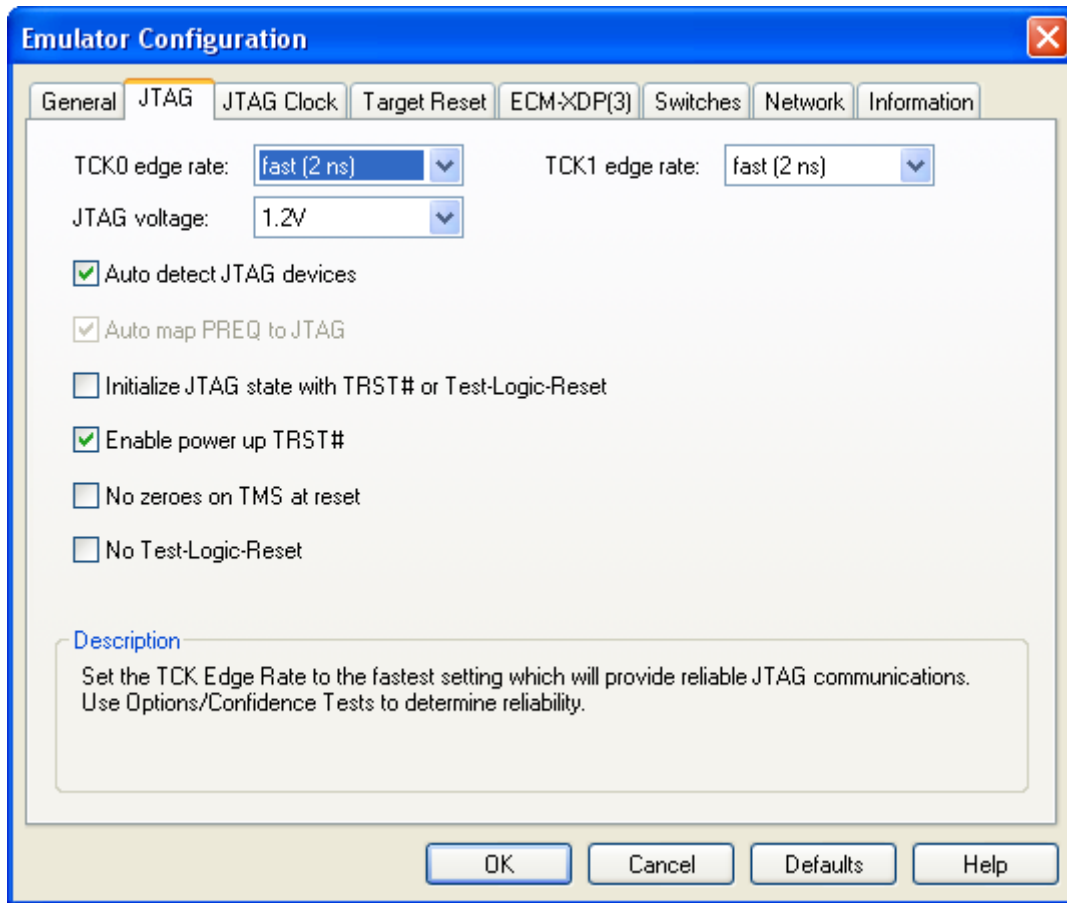
Disable breakpoints on step. When this option is enabled, breakpoints are disabled when low level stepping is performed. The option is designed for those processors that do not allow a refresh of breakpoints on a single step.

Cache control section. When the option **Never touch cache** is enabled, cache is never cleared until the emulator is powered down or a session is restarted. When the option **Clear entire cache when necessary** is enabled, SourcePoint executes write back invalidate (WBINVD) when the emulator stops the target and acquires it.

Note: This section is visible only if your target processor has cache.

JTAG Tab

The **JTAG** tab lets you change preset options associated with the JTAG scan chain.



JTAG Tab dialog box

TCK0 edge rate (ECM-XDP3 only). This option sets the TCK edge rate for the first JTAG chain. Set the TCK edge rate to the fastest setting which will provide reliable JTAG communications. Use Options/Confidence Tests to determine reliability. The choices are: slow (10ns), medium (5 ns), fast (2 ns). The default setting is fast

TCK1 edge rate (ECM-XDP3 only). This option sets the TCK edge rate for the second JTAG chain. Set the TCK edge rate to the fastest setting which will provide reliable JTAG communications. Use Options/Confidence Tests to determine reliability. The choices are: slow (10ns), medium (5 ns), fast (2 ns). The default setting is fast.

JTAG voltage (ECM-XDP3 only). Set to the voltage that the pull-ups on the processor(s) TDI and TDO are connected to on target. If that voltage is also connected to pin 43 of the XDP connector, you may choose 'Track VTT_AB'. If in doubt about a suitable level, use 1.2 V. The choices are: Track VTT_AB, 0.9V, 1.0V, 1.1V, 1.2V, 1.3V, 1.4V and 1.5V. The default is 1.2V.

Auto detect JTAG devices. When enabled, the emulator automatically identifies devices in the JTAG scan chain. Leave the box checked unless otherwise instructed by Arium personnel.

Auto map PREQ to JTAG. This option applies to PBD-S2x personality modules only. When enabled, the emulator automatically determines how PREQ and PRDY pairs are associated with the JTAG order. (JTAG order = Viewpoint.) If not enabled, the emulator assumes PREQ and PRDY Pair 0 is associated with Viewpoint 0, Pair 1 with Viewpoint 1, and so on.

Initialize JTAG state with TRST# or Test-Logic-Reset. This causes the emulator to assert TRST on setup to ensure the target's JTAG chain is initialized. This may cause certain targets to execute a few instructions from reset (including the Intel® Pentium® 4 and Xeon™ processors).

Enable power up TRST#. This option causes the target to be transitioned from TLR state to RTI state as soon as possible after power up. This may be useful in preventing execution of a few instructions from reset.

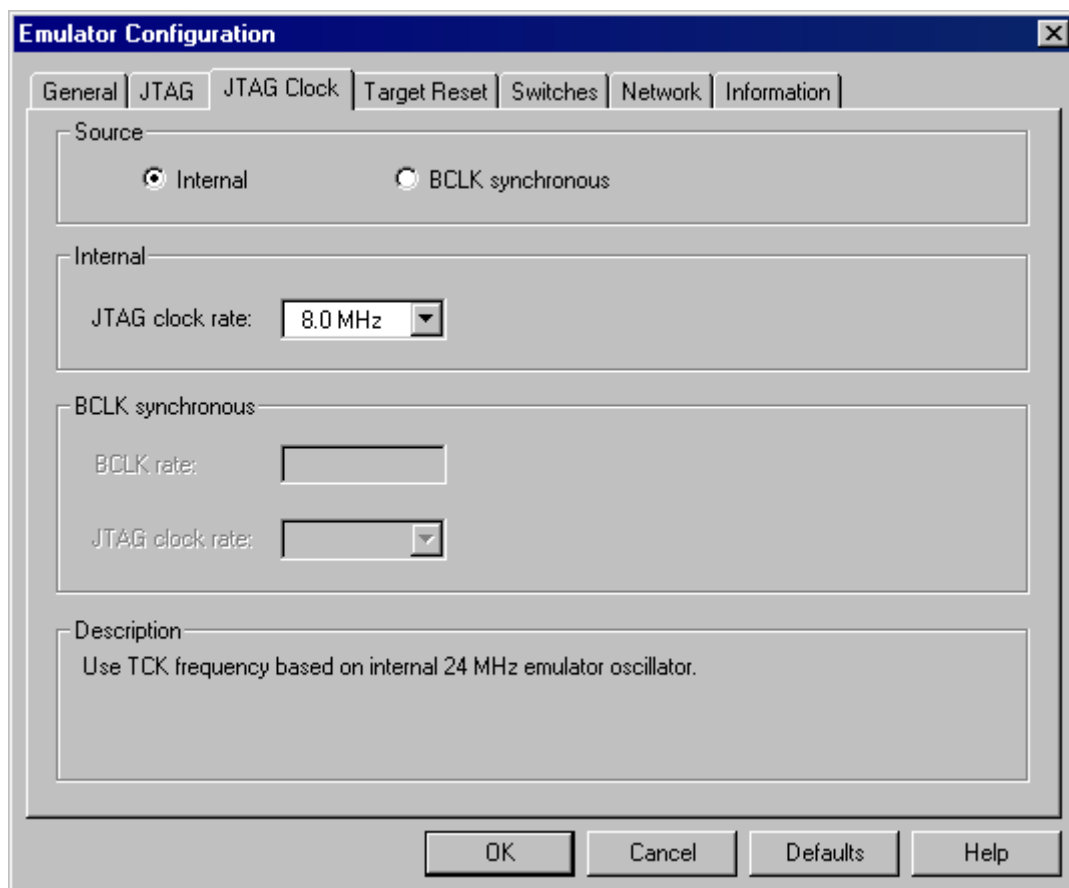
No zeroes on TMS at reset. This option determines whether zeroes are pumped out on TMS at reset.

No Test-Logic-Reset. Do not drive target through Test-Logic-Reset state during operation.

Caution: For S2Vs, set the strength to 4 or greater if the target without 39 Ohm termination resisters on TCK and TMS. Otherwise, then set the strength to 3 or less.

JTAG Clock Tab

The options on the **JTAG Clock** tab let you modify JTAG clock settings.



JTAG Clock Tab dialog box

Source field: Internal source. When enabled, this button tells the emulator to derive the TCK rate from its 24 MHz clock source.

Source field: BCLK synchronous. When enabled, this button tells the emulator to derive the TCK rate from the target BCLK signal (BCLK divided by 4 or 8, depending on the type of processor).

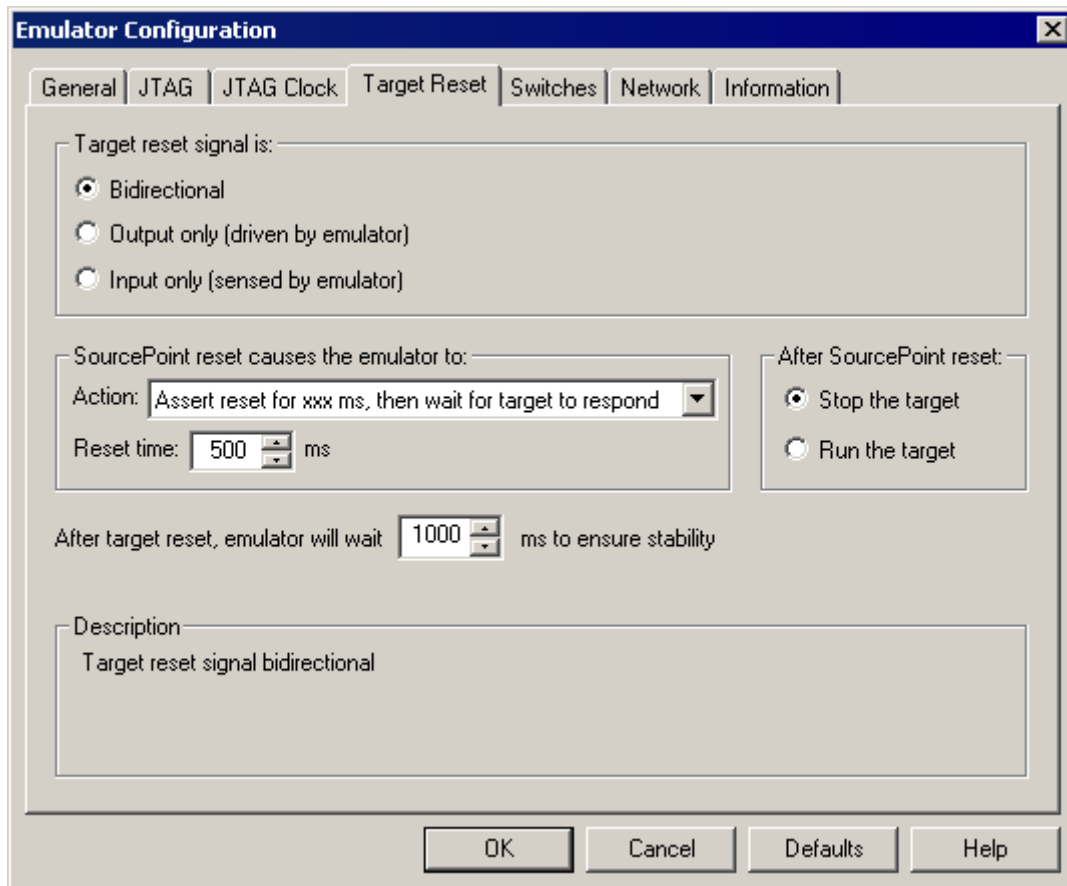
Note: BCLK synchronous TCK should not be used with emulators using PBD-S2x personality modules. Click on the Internal source button and use a jumper on the personality module to select BCLK synchronous TCK if desired.

Internal field: JTAG clock rate. This field allows you to specify a clock rate from among the choices provided in the text drop down box.

BCLK synchronous field: BCLK rate/JTAG clock rate. This field allows you to specify a BCLK rate and JTAG clock rate from among choices provided in attendant text drop down boxes.

Target Reset Tab

As the name indicates, the options on the **Target Reset** tab deal with target reset. The options you choose on this tab affect the way the **Reset** button works on the SourcePoint icon toolbar.



Target Reset Tab dialog box

Note: The options you select below should be determined by the way your target handles reset. For example, some targets may require that you manually toggle a switch to reset it while others

reset when a debugger pulls on them. How this works for your target is based on how it was designed to behave. You need to understand that behavior to make appropriate use of this tab.

Target reset signal is section. The options in this section let you select a target operation mode.

- Target reset is **Bidirectional**, where the emulator can assert and sense reset
- Target reset is **Output only (driven by emulator)**, where the emulator can assert reset, but does not sense reset
- Target reset is **Input only (sensed by emulator)**, where the emulator can sense reset, but not assert reset.

SourcePoint reset causes the emulator to section.

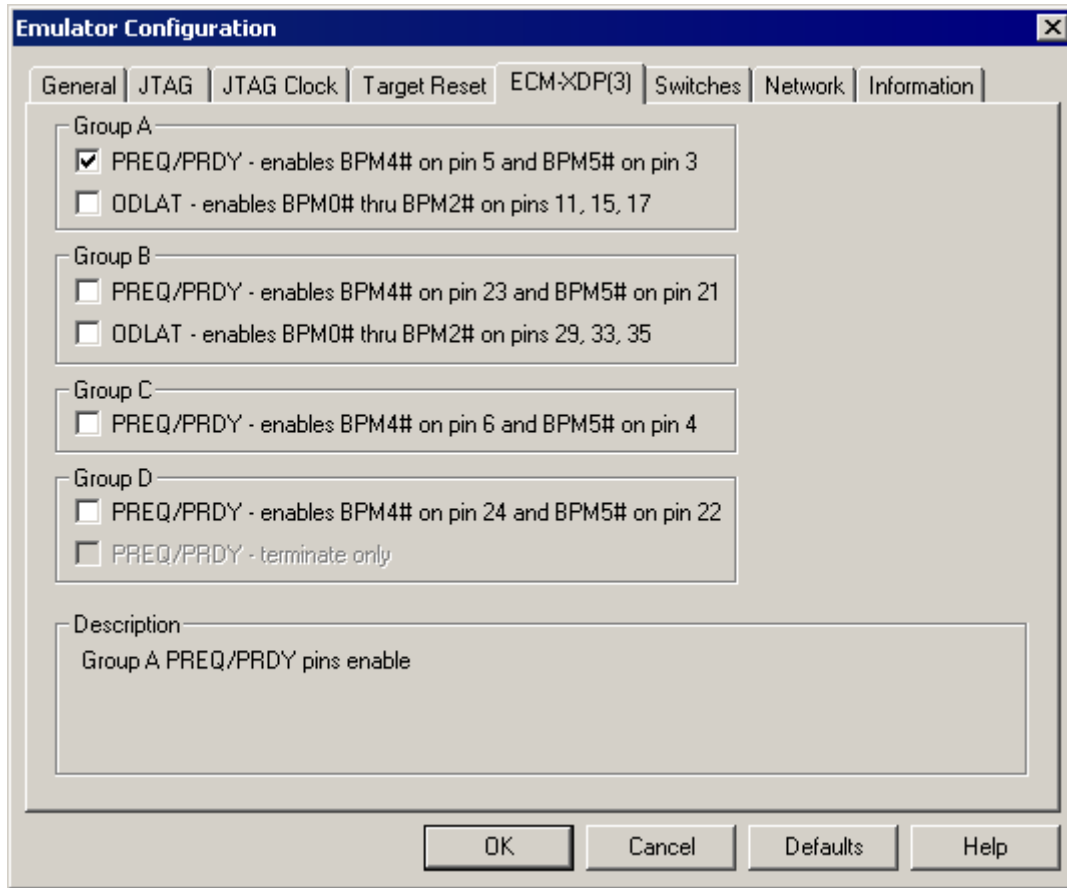
- **Action.** There are four options: **Wait on manual (external) target reset, Assert reset until target responds, Assert reset for xx ms, then wait for target to respond, or Assert reset for xx ms, don't wait for target to respond.**
- **Reset time.** Allows you to choose the length of time you want the emulator to wait. (Some targets need longer pulses than others.)

After SourcePoint reset section. There are two options in this section: **Stop the target** and **Run the target.**

After target reset, emulator will wait...ms to ensure stability. Allows you to set a time, in milliseconds, for the emulator to wait after the deassertion of target reset before JTAG communication is attempted.

ECM-XDP(3) Tab

This tab displays when you are connected to an ECM-XDP or an ECM-XDP3.



ECM-XDP(3) tab

The PREQ#/PRDY# signals (also known as BPM5# and BPM4#) provide the ability to start and stop the microprocessor(s) that is(are) connected to the XDP connector. The ODLAT signal group (BPM0# to BPM2#) provide the ability to set breakpoints on frontside bus accesses on a FEW types of processors. On a target with a single processor socket, that socket is connected to the group A pins. On a target with two processor sockets, the processors may be connected in parallel and may connect just to the group A pins. Alternatively, the two sockets may be connected to pin in separate groups; likely choices are groups A and C together or groups A and B together. Target systems with four processor sockets will likely have the processors bused in pairs (two on a bus), and again likely choices for proper operation are groups A and C or groups A and B.

Note: ODLAT (bus trigger) support is not available on groups C and D.

For proper run control of the target, if the target schematic shows a connection of PREQ# and PRDY# (or BPM5# and BPM4#) from a processor package to the XDP connector on the pins listed to the right of a check box, the box **MUST** be enabled. If the corresponding pins on the XDP connector are open (unconnected), the box may be checked or unchecked. If signals other than PREQ#/PRDY# (or BPM5#/BPM4#) are connected to the listed pins of the XDP connector, the box **MUST** be disabled.

For proper bus trigger operation, if the target schematic shows a connection of BPM[0-2]# from a processor package to the XDP connector, then enable the group(s), A or B, (or both) that correspond to the connection. If there is no connection to the corresponding pins on the XDP

connector, then the box may be enabled or disabled. If signals other than BPM[0-2]# are connected to any of the pins listed for ODLAT within a group, the box must be disabled.

Switches Tab

You should use this tab under the direction of Arium technical support personnel.

Network Tab

The **Network** tab provides a GUI for changing the emulator network settings. This only changes the network settings; it does not affect the entries in the **Emulator Connections** dialog box.

The screenshot shows the 'Emulator Configuration' dialog box with the 'Network' tab selected. The dialog has several tabs: General, JTAG, JTAG Clock, Target Reset, Switches, and Network. The Network tab contains the following fields and options:

- MAC address:** A text box containing '00 : D0 : A5 : 00 : 04 : A7'.
- Name:** An empty text box.
- IP address setup:** A section with two radio buttons:
 - ☒ Specify address
 - ☐ Obtain from a DHCP server
- IP address:** A text box containing '65 . 169 . 214 . 122'.
- Subnet mask:** A text box containing '255 . 255 . 255 . 0'.
- Default gateway:** A text box containing '65 . 169 . 214 . 254'.
- Description:** A text area containing the text: 'Configure network settings of emulator. Any changes will not take effect until after you reset the emulator (Options, Emulator Reset...). Be sure to update your emulator connection settings to match these (Options, Emulator Connection...)'.

At the bottom of the dialog are four buttons: OK, Cancel, Defaults, and Help.

Network dialog box

MAC address and Name. These text boxes identify the emulator on the network. The **Name** text box may be blank.

IP address setup section

- **Specify address.** If you want to use a fixed IP address, you need to select this button and fill in the **IP address**, **Subnet mask**, and **Default gateway** fields in this section. Contact your network administrator if you are unsure what information to use in these fields.

- **Obtain from a DHCP server.** Enabling this button automatically fills in the IP address, assuming you have a DHCP server.

Note: Arium recommends you check with your Network Administrator before enabling this option.

- **IP address, Subnet mask, Default gateway.** These are the network settings of the emulator.

Note: You can change these settings via this tab. You must reset the emulator for changes to take effect. The changes made here do not modify the emulator connection; you should update that to match. For more information on using the **Emulator Connections** dialog box, see, "[Options Menu - Emulator Connections Menu Item](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*.

Information Tab

The **Information** tab gives you information on the configured system you are running currently. The fields are read only and are usually used if you are having problems getting the emulator to work.

Emulator text box. This field gives you the name of the emulator to which you are attached.

Firmware text box. This field displays the revision level (vn.nn) for the two portions of emulator flash memory: boot and flash. Boot memory is the factory programmable portion of flash memory. Flash memory is the field programmable portion.

PBD text box. This field provides information on the type and revision number of the personality module (JTAG).

Board text box. This field provides information on the board inside the emulator.

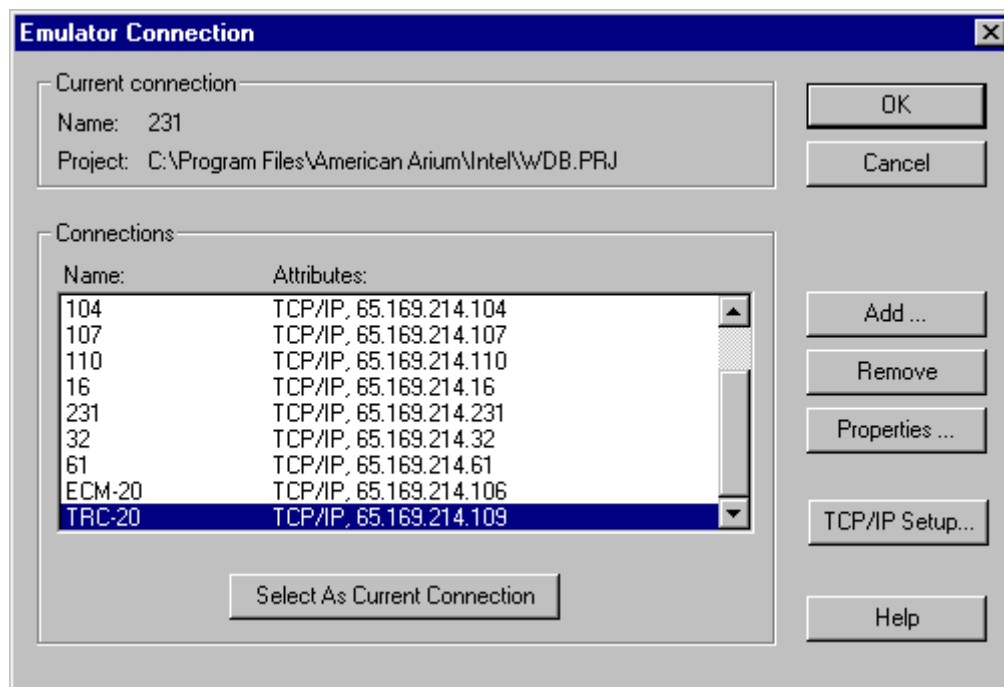
Serial No. text box. This field gives you the serial number of your emulator.

Options Menu - Emulator Connection Menu Item

An emulator connection is the communications link between SourcePoint software and hardware (emulator) connected to a user's target system. You may choose from: TCP/IP (direct or network) or USB.

The information below briefly describes each of the fields and buttons in the **Emulator Connection** dialog box. Actual setup depends on a number of variables, including your choice of connections and your network configuration. For this reason, a single set of connection instructions is insufficient, and multiple instructions placed one after the other can be confusing. For information on setting up a specific type of connection, see the last portion of this topic.

Select **Options|Emulator Connection** from the menu bar. The **Emulator Connection** dialog box displays. It is used to view and modify emulator connections.



Emulator Connection dialog box

Current Connection section. The **Current Connection** section displays the connection currently in use.

Connections list box. The **Connections** list box displays the available emulator connections. Connection names and selected attributes are displayed. To change the current emulator connection, highlight the connection desired, and then click the **Select As Current Connection** button and click the **OK** button. Alternatively, you can double-click the desired connection and click then click the **OK** button.

- **Add/Remove.** These buttons are self explanatory.
- **Properties.** This button takes you to the connection properties box of the highlighted connection.

- **TCP/IP Setup.** This button takes you to a wizard that guides you through the TCP/IP connection process. For more information, review the topic in "Installation Overview" found under *Installation* that corresponds to your emulator.

For More Information

- For information on how to set up an emulator connection for the first time, see the *Getting Started* guide that shipped with your unit.
- For detailed information on how to add an emulator connection, select the topic, "[Add Emulator Connections](#)" under the "How To - SourcePoint Environment," part of *SourcePoint Environment*.

Options Menu - Emulator Reset Menu Item

Select **Options|Emulator Reset** the menu bar. A reset is required to cause the emulator to begin using any parameters you may have made via the **Emulator Configuration** menu item. Any TCP/IP or USB connection is lost when this is done.

Options Menu - Confidence Tests Menu Item

To set view test results and change test parameters, go to **Options|Confidence Tests** on the menu bar. The **Confidence Tests** dialog box displays.

There are a number of confidence tests available in SourcePoint. Once enabled, additional setup options are available by clicking corresponding options in the **Test Setup** section. All tests have default setup configurations so that tests may be executed using the default test suite, skipping additional setup steps.

As the requested tests run, the test status block near the bottom of the dialog box changes to show the progress of the testing. At the end of the testing, **Status** buttons indicate test results. Click on the corresponding button to display additional test details.

For additional information regarding Confidence Tests, begin with the topic, ["Confidence Tests Window Introduction."](#)

Window Menu

Items in the **Window** menu are: **Close**, **Cascade**, **Tile Horizontally**, **Tile Vertically**, **Arrange Icons**, **Arrange Toolbars**, and **Close All**. They are described in detail below.

Close Menu Item

Select **Close** on the menu bar to close the current window. Repeat this as desired to close other windows or double-click on the corresponding window control box.

Cascade Menu Item

Select **Cascade** on the menu bar to align the windows from the top left and layer the open windows, making each title bar visible.

Tile Horizontally Menu Item

Select **Tile Horizontally** on the menu bar to resize and arrange the open windows in a top-to-bottom layout. All the elements of a tiled window may not be visible.

Tile Vertically Menu Item

Select **Tile Vertically** on the menu bar to resize and arrange the open windows in a side-to-side layout. All the elements of a tiled window may not be visible.

Arrange Icons Menu Item

Select **Arrange Icons** on the menu bar to align and evenly space any icon (minimized windows) present in the main window.

Close All Menu Item

Select **Close All** on the menu bar to close all of the currently open windows.

Help Menu

Select **Help** from the SourcePoint menu bar to access the following menu items: **Index**, **Using Help**, **License File**, and **About SourcePoint**.

Index Menu Item

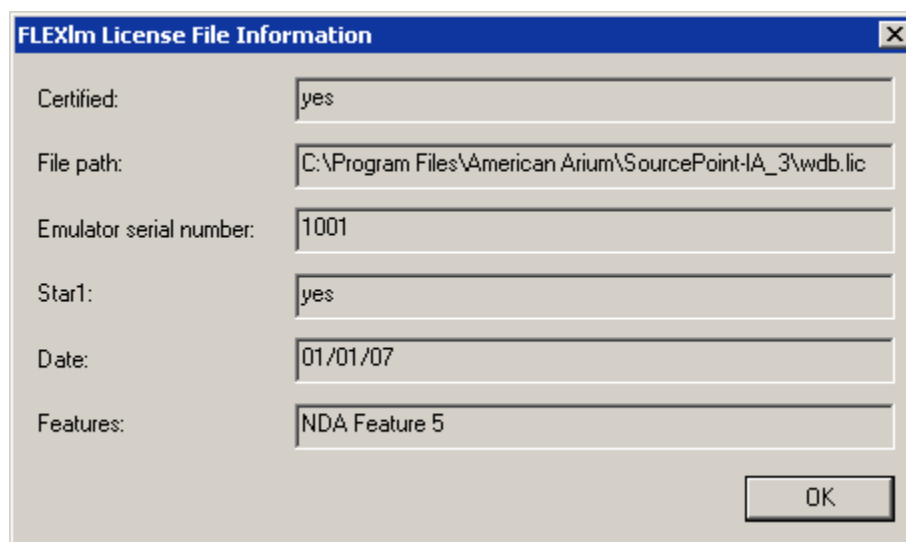
Select **Help|Index** on the menu bar to display an alphabetical list of help topics and related information.

Using Help Menu item

Select **Help|Using Help** on the menu bar to access detailed information on how to use **Help**.

License File Menu Item

Select **Help|License File** to open a text box containing information on the license available with your SourcePoint software.

The image shows a Windows-style dialog box titled "FLEXlm License File Information". It contains six text input fields with the following labels and values: "Certified:" with "yes", "File path:" with "C:\Program Files\American Arium\SourcePoint-IA_3\wdb.lic", "Emulator serial number:" with "1001", "Star1:" with "yes", "Date:" with "01/01/07", and "Features:" with "NDA Feature 5". An "OK" button is located at the bottom right of the dialog box.

FLEXlm License File Information dialog box

About SourcePoint Menu Item

Select **Help|About SourcePoint** on the menu bar to display the software version and copyright information for SourcePoint.

How To -- SourcePoint Environment

How to Add Emulator Connections

Once you have successfully established the first communication connection between the emulator and host system (as described in the *Getting Started* guide that shipped with your unit and available online at www.arium.com/support/techdocs.html), you can add emulator connections at any time. Three basic types of connections are described below: USB, TCP/IP, and Serial.

USB Connections

If you are adding a USB connection (assuming you currently have a TCP/IP connection):

1. Attach the USB cable. (This can be hot plugged; you do not need to disconnect your TCP/IP cable.)

If this is the first time you are establishing a USB connection, the standard Microsoft Windows hardware installation wizard appears asking you to install the USB driver. If not, skip to #3 below.

2. Click on the **Browse** button in this dialog box and browse to your SourcePoint working directory, click on "AriumUsb.inf", and complete the Microsoft installation wizard.
3. Select **Options|Emulator Connection** from the toolbar menu.

The **Emulator Connection** dialog box appears, and the USB connection automatically appears in the connection list.

Note: If you connect the USB cable while an **Emulator Connection** dialog box is open, you will not see the connection in the box. You must close the box, THEN connect the cable and open the dialog again to see the connection.

4. Select the USB connection from the list of connections if it is not already selected.
5. Double click the selected connection or click on the button labeled **Select As Current Connection** and click the **OK** button.

TCP/IP Connections

If you are adding a TCP/IP connection:

1. If necessary, change or add the hardware cable.

For a direct (non-network) connection between host computer and emulator, a crossover cable is required. (An orange crossover cable is included with new emulators). To connect the emulator to a network, a direct cable is required. (A blue direct cable is included with new emulators.)

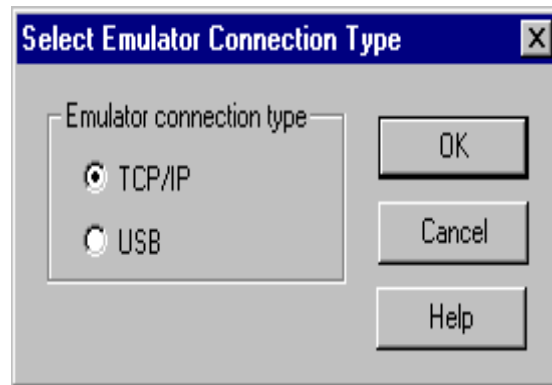
2. Select **Options|Emulator Connection** on the menu bar.

The **Emulator Connection** dialog box displays.

3. Press the **Add** button.

The **Select Emulator Connection Type** dialog box appears.

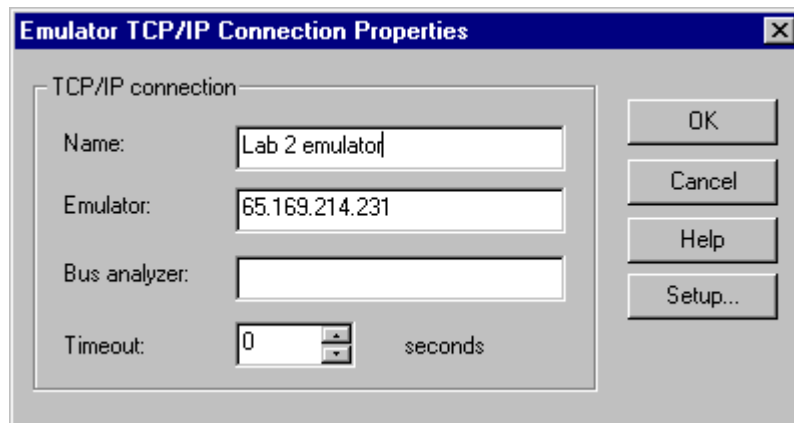
4. Select **TCP/IP**.



Select Emulator Connection Type dialog box with TCP/IP selected

5. Click the **OK** button.

The **Emulator TCP/IP Connection Properties** dialog box opens.



Emulator TCP/IP Connection Properties dialog box

6. Fill in the blanks.
 - **Name.** The **Name** text box is a required entry. Create a name that helps you recognize the emulator.
 - **Emulator.** The **Emulator** text box specifies the emulator IP address.
 - For direct IP and serial connections, use IP address 192.168.000.001.

- For network IP connections, get an address from your NetworkAdministrator.
- **Bus Analyzer.** The **Bus Analyzer** text box is used to place the IP address of an optional bus analyzer. Currently, SourcePoint supports only Agilent Technologies logic analyzers.
- **Timeout.** The **Timeout** control specifies the number of seconds to add to SourcePoint's internal communication timeout value for this emulator connection. The default value is 10 seconds.

7. Click the **OK** button.

The **Emulator Connection** dialog box redisplay. The new emulator connection information is highlighted.

8. Double-click the highlighted entry or click on **Select As Current Connection** button and then the **OK** button.

The name of the current connection is displayed at the top of the **Emulator Connection** dialog box under **Current Connection**.

Using Dynamic DNS (DDNS) for Addressing Emulators by Hostname

If your network includes a Microsoft Windows or Linux server that provides DHCP and DDNS services (Bind DNS), you can configure your emulator to request a dynamic IP address from the server (DHCP) and then configure SourcePoint to address the emulator by name (e.g., serial number) instead of by IP address.

Adding a dynamic TCP/IP connection by using a hostname:

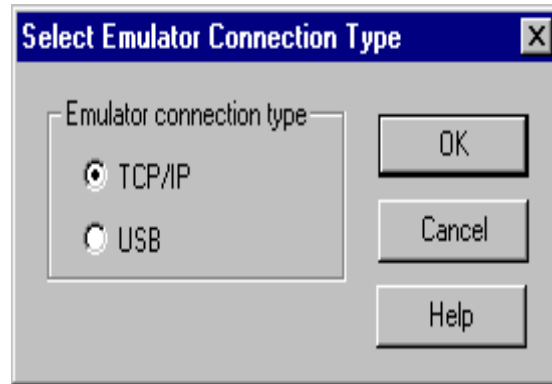
1. Make sure that the emulator is connected to the network with a direct cable (not a crossover cable). A blue direct cable is included with every new emulator.
2. Select **Options|Emulator Connection** on the menu bar.

The **Emulator Connection** dialog box displays.

3. Click the **Add** button.

The **Select Emulator Connection Type** dialog box appears.

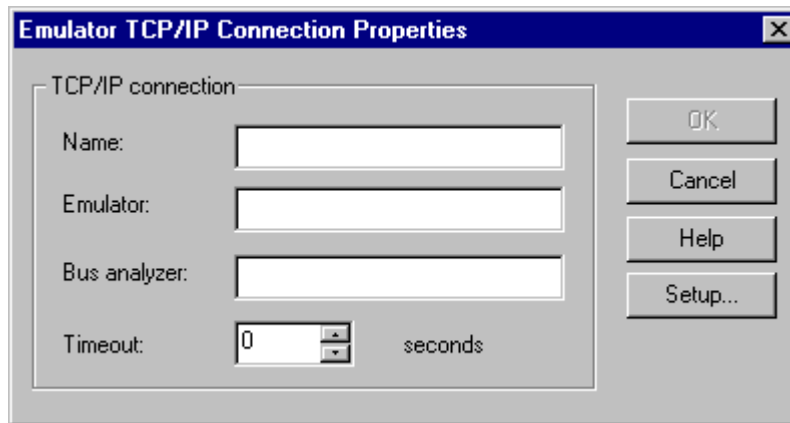
4. Select **TCP/IP**.



Select Emulator Connection Type dialog box with **TCP/IP** selected

5. Click the **OK** button.

The **Emulator TCP/IP Connection Properties** dialog box opens.



Emulator TCP/IP Connection Properties dialog box

6. Fill in the blanks.
 - **Name.** The Name text box is a required entry. Create a name that helps you recognize the emulator.
 - **Emulator.** In this configuration, the **Emulator** text box specifies the name of the emulator as it is registered in the DDNS service. This is in the format *ecm-serial number* (e.g., ecm-5123).

Specify the emulator name as a Fully Qualified Domain Name (FQDN). If using a fully qualified domain name (FQDN), key in "ecm-serial number.domain name" without quotation marks, inserting the actual serial number and your domain name in place of the italicized words shown (i.e., ecm-5123.abc.net).

- **Timeout.** The Timeout control specifies the number of seconds to add to SourcePoint's internal communication timeout value for this emulator connection. The default value is 10 seconds.

7. Click the **OK** button.

The **Emulator Connection** dialog box redisplay. The new emulator connection information is highlighted.

8. Double-click the highlighted entry or click the **Select As Current Connection** button and then the **OK** button.

The name of the current connection is displayed at the top of the **Emulator Connection** dialog box under **Current Connection**.

How to Configure Custom Macro Icons

SourcePoint allows you to associate macro files with toolbar buttons.

Configuring SourcePoint

To configure SourcePoint to automatically load macro files:

1. Select **File|Macro|Configure Macros** from the menu bar.

The **Configure Macros** dialog box displays.

2. In the **Select Macro** drop down list in the User defined macros section of the dialog box, choose the macro icon number with which you want to associate your macro.
3. Type the macro file path and name in the **Macro filename** text box.
4. Enable the **Echo file to command window** option to display the macro commands when loading.
5. Type a brief description in the **Macro button** text box.

This is the text that appears next to the icon on the toolbar if you have it enabled.

Note: To display this text on the macro toolbar, right-click on the toolbar and select **Icons & Text** from the context menu.

6. Click the **OK** button.

Adding Macro Icons

To add more than the default three macro icons to the **Macro** icon toolbar:

1. Right-click on the **Macro** icon toolbar.
2. Select **Customize** from the context menu.

The **Customize Toolbar** dialog box displays.

3. Select the icon to add from the **Available toolbar buttons** list.

The selected icon displays in the **Current toolbar buttons** text box.

4. Click the **Add** button.
5. Click the **Close** button.

Removing Macro Icons

To remove macro icons from the icon toolbar::

1. Right-click on the **Macro** icon toolbar.
2. Select **Customize** from the context menu.

The **Customize Toolbar** dialog box displays.

3. Select the icon to remove from the **Current toolbar buttons** list.
4. Click the **Remove** button.
5. Click the **Close** button.

Note: The **Reset** icon on the toolbar restores default buttons (**Execute Macro 0-3**).

How to Configure Autoloading Macros

SourcePoint allows you to specify macro files to be loaded when certain events occur. These events include:

Event	Macro Loaded
Startup	When SourcePoint is started (all projects)
Project Load	When the current project is loaded.
Go	When the processor(s) is started by SourcePoint toolbar or F5 key
Stop	When the processor(s) is stopped by SourcePoint toolbar or Alt-F5 key
Reset	When the target system is reset via the SourcePoint toolbar or Shift-F5 key
Breakpoint	When a SourcePoint breakpoint causes the target to stop

To configure SourcePoint to automatically load macro files:

1. Select **File|Macro|Configure Macros** from the menu bar.
2. Select the event from the **Select Event** drop down list in **Event macros** section of the dialog box.
3. Type the macro file path and name in the **Macro filename** box.
4. Enable the **Echo file to command window** option to display the macro commands when loading.
5. Click the **OK** button.

How to Display Text on the Icon Toolbar

To add text to a group of icons or to all of them, follow the directions below.

Display Text Next to All Icons

1. Right click anywhere on the toolbar to open the context menu.
2. In the menu enable the **Icons & Text** menu item.

A dialog box labeled **SourcePoint** displays.

3. Choose the **Yes** option to add text to all toolbar icons.

Display Text Next to a Group of Icons

1. Click specifically on the toolbar group against which you want to display text to open the context sensitive menu.
2. In the menu, enable the **Icons & Text** menu item.

A dialog box labeled SourcePoint displays.

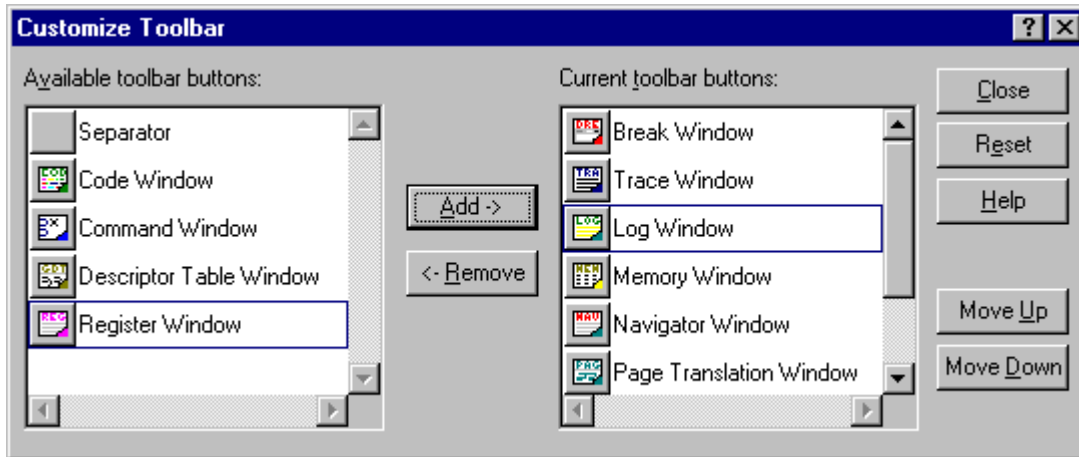
3. Choose the **No** option to add text to all toolbar icons.

SourcePoint adds text only to the icons in the group you have chosen.

How to Edit Icon Groups to Customize Your Toolbars

1. Right-click the mouse on any icon group.

This opens the **Customize Toolbar** window.



Customize Toolbar window

2. To add icons, select the desired buttons from the **Available toolbar buttons** list. Click the Add button. These icons are added to the **Current toolbar buttons** list and the icons toolbar.
3. To remove icons, select the desired buttons from the **Current toolbar buttons** list. Click the **Remove** button. These icons are removed from the **Current toolbar buttons** list and the icons toolbar.
4. Click the **Reset** button to return the toolbar selections to the SourcePoint default toolbar.
5. Use the **Move Up** and **Move Down** buttons to rearrange the toolbar buttons.

How to Hot Plug an Arium Emulator

1. Attach the emulator to the host system.

Caution: Do NOT connect the emulator to the target.

2. Connect the ground of the emulator to the ground of the target.

Caution: If this is not done, static discharge or a ground mismatch error may corrupt the target and prevent connection.

3. Bring up SourcePoint.
4. Select **Options|Emulator configuration** from the menu bar.
5. In the **General** tab, ensure that **Delay target acquisition** is disabled.
6. In the **JTAG** tab, ensure that **Auto detect JTAG devices** is enabled.
7. Open the **Command** window.
8. Key in the command **HotPlug()**.
9. Wait for a dialog box to display instructing you to attach the emulator probe. Do not close the dialog box.
10. Connect the probe to the target.

Note: The target does not stop on connection.

11. Click **OK** on the dialog box.

The target stops within three seconds, and the project reloads.

12. If the project does not reload successfully, reload the project.
13. Ensure that the target has been acquired and is ready for manipulation.

How to Modify a Defined Memory Region

Adding or Modifying a Currently Defined Memory Region

1. Select **Options|Target Configuration** from the menu bar.

The **Target Configuration** dialog box displays.

2. Click the **Memory Map** tab.
3. To add or modify a currently defined memory region, click on the **Add** or **Edit** button beneath the **Memory Map** list box.

The **Add Memory Map Entry** dialog box or **Edit Memory Map Entry** dialog box displays.

Add Memory Map Entry under the **Memory Map** tab of **Options|Target Configuration**

4. Enter the physical address where the memory map range begins in the **Starting address** field. Memory accesses to addresses not found within the memory map use the following rules: Memory writes are always allowed, and Memory reads are allowed unless Safe mode is enabled.
5. Enter the physical address where the memory map ends in the **Ending address** field. Memory accesses to addresses not found within the memory map use the following rules: Memory writes are always allowed, and Memory reads are allowed unless Safe mode is enabled.
6. Select the physical memory width (8, 16, or 32 bits) via the **Access size** drop down box. This is the access size that is used when memory within this range is read or written to.
7. Select a type of memory from the **Type** drop down list. Choices are: **SRAM**, **DRAM**, **ROM**, or **Flash**.
8. Select an option from the **Processor** drop down list. This field is only available on non-SMP targets. It allows you to select whether a memory range is local to a given processor or is accessible to all processors. Entering a processor number indicates that the defined range is only accessible by that processor. Entering ALL indicates that the memory range is shared by all processors.

Note: It is not possible to define a range shared by some, but not all, processors.

Removing a Currently Defined Memory Region

Select the **Remove** or **Remove All** button on the **Memory Map** tab to remove a defined memory region. The **Remove** button removes the currently selected memory map entry. The **Remove All** button removes all memory map entries.

How to Refresh SourcePoint Windows

- To refresh a single window, click the **Refresh** button on the window dialog box. Not all dialog boxes have this feature.
- To refresh all windows, click the **Snapshot** menu item on the **Processors** menu or the **Snapshot** icon on the icon toolbar.
- To set a timed refresh of all windows:
 1. Select **Options|Preferences|General** tab.
 2. The **General** tab displays.
 3. Check the box labeled **Enabled** in the **Timed window refresh** section.
 4. In the **Interval** box, select the number of seconds between refreshes, a number between 1-999 or leave it at the default of 10 seconds.
 5. Click the **OK** button.

How to Save a Program

1. Select **Save Program** to save the program.

The **Save Program** dialog box displays.

2. Select the destination directory in the **Folders** tree window.
3. Select a file from the **File name** text box to replace an existing file, or enter the name of a new file in the box above it. If the desired existing file is not visible, change the selected filter in the **List files of type** drop down list.
4. Enter the beginning address and length of the target memory range to be saved in the **Target memory address** and **Length** text boxes, respectively.
5. Press the **Save** button.

How to Use the EFI Address Finder

EFI applications and drivers are relocatable; the EFI loader decides the address of the program at loading time. SourcePoint provides several ways to find EFI load addresses. With the EFI address finder function, SourcePoint allows you to get the address automatically, assuming your target is running IA-64 EFI. To automatically find the EFI base address:

1. Copy the EFI application under test (*filename.efi*) onto a standard DOS diskette.
2. Click on the **Go** icon on the SourcePoint icon toolbar (or press F5 on your keyboard). Allow the target to load the EFI shell.

The EFI shell looks like a DOS command window.

3. To load the EFI application from the floppy, enter the following commands at the EFI shell:
 - a. >blk0: or >fs0, depending on which designates the floppy drive
 - b. >*filename.efi*, where the filename is the name of your file (e.g., "flat.efi")

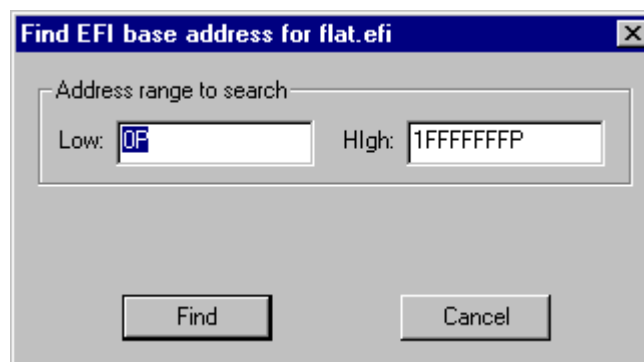
Note: The first command may be unnecessary, but it is included here for completeness.

4. Stop the target by clicking the **Stop** icon on the SourcePoint icon toolbar.

Note: The ".efi" file must be loaded for the address finder to work. The address finder cannot find the address of a program that is not loaded.

5. Select **File|Program|Load Program** or click on the **Load Program** icon on the icon toolbar.
6. In **List files of type**, from the drop down text box, select EFI(PE) format (*.efi).
7. Find your .efi file and place it in the **File name** text box.
8. Click the **Detect** button.

The **Find EFI base address for...** dialog box displays.



Find EFI base address dialog box

9. Edit the address range, if you wish, or use the range provided.
10. Click the **Find** button.

The address is displayed in the **Address** text box in the **Program Load** dialog box.

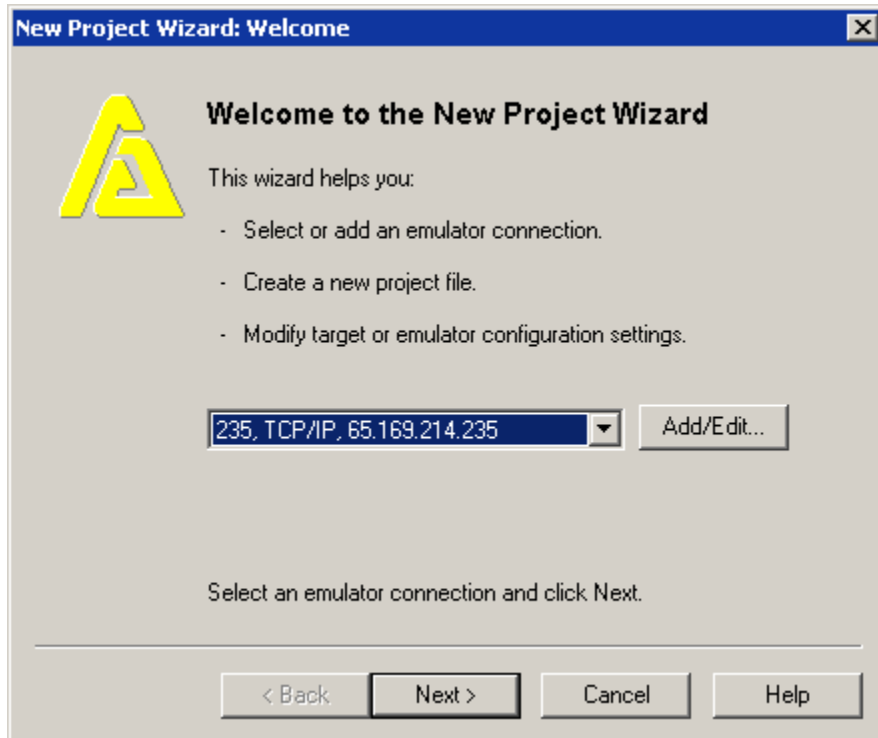
Note: Attempting to access memory where memory does not exist or is not yet initialized may result in an error. When this happens, you may need to reset your target. To prevent the EFI address finder from accessing a non-existing memory area, you can specify the memory range to search if you know the approximate location of the EFI table

How to Use the New Project Wizard

1. Select **File|Project|New Project** to open the wizard.

The **New Project Wizard: Welcome** screen displays.

2. Select, edit, or add the IP address of the emulator.

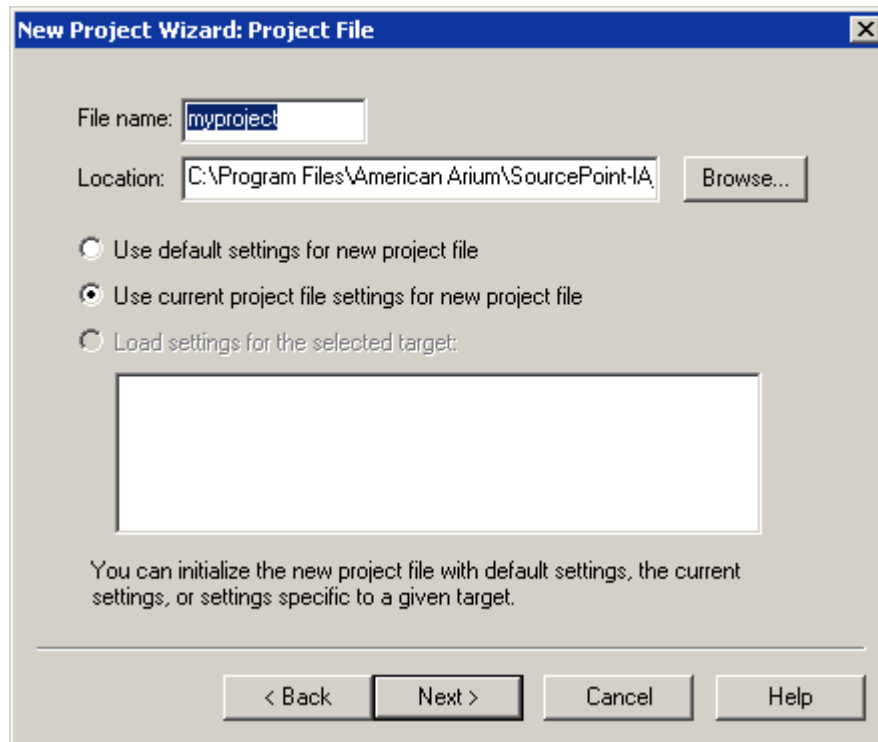


New Project Wizard: Welcome dialog box.

3. Click the **Next** button.

The **New Project Wizard: Project File** dialog box displays.

New Project Wizard: Project File Screen



New Project Wizard: Project File dialog box

1. In the File name text box, key in the name you want to give the project.

Note: Be sure to add the .prj extension.

2. To fill in the **Location** text box, decide where you want to save the file and key in or browse to the path that denotes that location.
3. Select one of three settings: **Use default settings**, **Use current settings**, or **Load setting for the selected target**.
4. If you select the third option, you must select a target from those listed in the text box immediately below that option.
5. Click the **Next** button.

The **New Project Wizard: Emulator and Target Configuration** dialog box displays.

New Project Wizard: Emulator and Target Configuration Screen

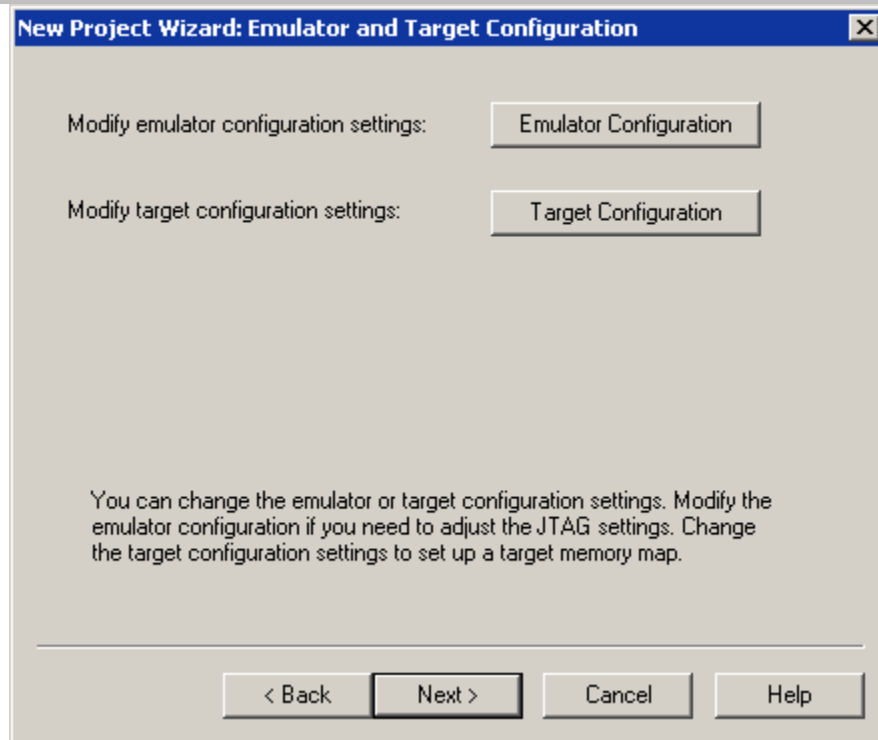
This dialog box allows you to change the default emulator and target configuration settings.

1. Click on the **Emulator Configuration** button to modify JTAG settings.

An **Emulator Configuration** dialog box displays, showing four or five tabs (depending on your target). These include **General**, **JTAG**, **JTAG Clock**, and **Target Reset** tabs. On targets with XScale processors, an XScale tab also displays.

2. Modify these emulator settings as needed.

Note: For details on the options in each tab, see "[Options Menu - Emulator Configuration Menu Item](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*.



New Project Wizard: Emulator and Target Configuration dialog box

3. Click on the **Target Configuration** button to change target configuration settings and to set up a memory map.

A **Target Configuration** dialog box displays, showing two tabs: **Memory Map** and **Operating System**.

4. Modify these settings, as needed.

Note: For details on the options in each tab, see "[Options Menu - Target Configuration Menu Item](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*.

5. Once you have modified emulator and configuration settings, lick the Next button.

The **New Project Wizard: Completing** dialog box displays.

New Project Wizard: Completing Screen

1. Click the **Finish** button to load your new project file.

How to Verify Emulator Network Connections

To verify emulator network connections:

- Verify that the proper cable is connected. For a direct connection from computer to emulator, a crossover cable is required. For connection to a network, a direct cable is required. Every new emulator ships with a blue direct cable and an orange crossover cable.
- From a **Command** window, (i.e., DOS box), use a Ping command to test the connection to the emulator. For example, type in "Ping 192.168.0.1" or "Ping ecm-5123" (without quotation marks).

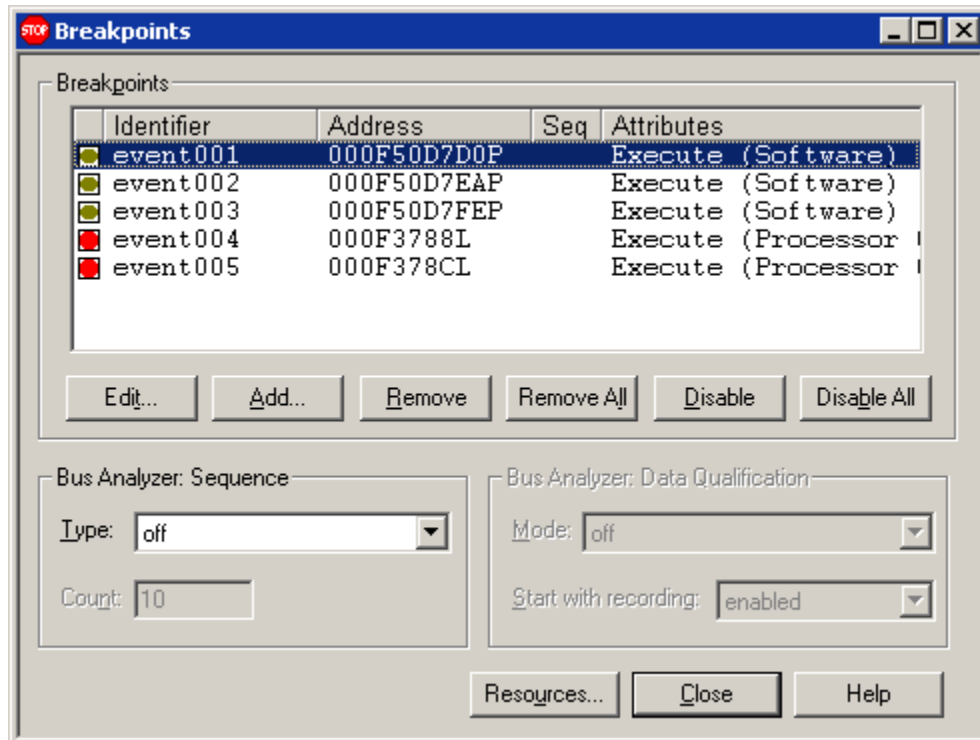
If the Ping command fails, you do not have a functioning emulator connection. You need to troubleshoot your network connection or switch to a USB connection. For troubleshooting information, refer to the *Getting Started* manual that shipped with your emulator (and is available at www.arium.com/support/techdocs.html)

Breakpoints Window

Breakpoints Window Overview

Breakpoints Window Introduction

The **Breakpoints** dialog box is used to manage the various breakpoint and bus analyzer data qualification settings. The dialog box can be opened by selecting **View|Breakpoints** from the menu bar or by clicking on the **Breakpoint** icon on the toolbar.



Breakpoints dialog box for most emulators

Note: If the dialog box shows part of it with scroll bars, you may choose to close the dialog box and bring it up again in order to avoid working with a scrolling box. The dialog box then opens to its full size.

The **Breakpoints** list box displays the address and type of each breakpoint (enabled or disabled), sequence, and any specified attributes. A Breakpoint icon precedes the **Address** field when a breakpoint is enabled. An empty check box indicates a breakpoint is disabled. New breakpoints are enabled automatically when added to the list. For more information on Breakpoint icons, refer to "[Breakpoints Window Icon Definitions](#)," found under "Breakpoints Overview," part of *Breakpoints Window*.

Identifier. This is a user-defined field set via the **Add Breakpoint** dialog box. It displays a name you create for each breakpoint. If the column has been resized and no longer displays the complete name, use the flyover help to display the complete name.

Address. The **Address** column displays the logical, linear, or physical address of the memory or I/O breakpoint. The column displays the address as it was entered. To view the translated address (physical for bus breaks and soft breaks, linear for debug register breaks), position the cursor over the address. A ToolTip window opens, displaying the address after translation, along with the address translation type (**Once** or **Every go**). If the address translation type is **Once**, then the address displayed indicates the result of the address translation that occurred when the address was first entered. If the address translation type is **Every go**, then the address displayed indicates the result of an address translation using the current processor context.

Values, by default, are specified in hexadecimal. To specify a binary value, append a 'B' character to the end of the value. If the breakpoint resource is a bus analyzer resource, then "Don't Cares" can be specified by entering an 'X' character in the bits to ignore. If a program file with symbolic debug has been loaded, then symbol address expressions can be entered.

Seq. Used in conjunction with bus analyzer breakpoints, this column lists the type of sequence. See below for more information.

Attributes. The **Attributes** column is used to display the type of activity on which the processor is specified to break. Each break activity is associated with resources that can be displayed by pressing the **Resources** button found at the bottom of the **Breakpoints** dialog box or via the **Resources** menu item in the context menu. These resources implement a break according to the break-on activity type and the parameters specified. For additional information, refer to ["Breakpoints Menu,"](#) part of "Breakpoints Overview," found under *Breakpoints Windows*.

Buttons Under the Breakpoints List Box

These are the six buttons below the breakpoints list box:

1	Edit	Brings up the Edit Breakpoint dialog box to edit the currently selected breakpoint.
2	Add	Brings up the Add Breakpoint dialog box to add a breakpoint.
3	Remove	Removes the currently selected breakpoint.
4	Remove All	Removes all breakpoints displayed in the text box at once.
5	Disable/Enable	Disables or enables the currently selected breakpoint.
6	Disable All/Enable All	Disables or enables all breakpoints displayed in the text box at once.

Bus Analyzer: Sequence Section

The Bus Analyzer: Sequence section in the **Breakpoints** dialog box (not to be confused with the **Sequence** field in the **Add Breakpoint** or **Edit Breakpoint** dialog boxes) is used whenever a bus analyzer breakpoint is in use. The **Type** field is used to select the way in which the bus analyzer breakpoints are used in stopping the processor.

Bus Analyzer: Sequence works in conjunction with any other breakpoints set. For example, if a processor breakpoint is set along with a sequence, whichever is detected first will cause the processor to stop. Two steps are involved in using this feature: selecting the type of sequence and defining the bus analyzer breakpoint(s) to be used in the sequence. The steps may be performed in either order.

The list of available **Type** selections is short and simple. A brief explanation of each selection follows.

Sequence Type	Explanation
Off	No states examined.
T1	Any Bus Analyzer breakpoint with SEQ=T1 participates in the breakpoint activity. All T1 breakpoints act in parallel (are "OR'ed" together). Up to six breakpoints can be active at T1.
T1 then T2	A two-level breakpoint sequence. A breakpoint with SEQ=T1 must occur, then another breakpoint with SEQ=T2 must occur subsequently before the sequence stops the processor. Up to two breakpoints can be active at each level.
T1 then T2 then T3	A three-level breakpoint sequence. Similar to T1 then T2, except that only one breakpoint may be active with SEQ=T3.
T1 then T2 without T3	T3 must not occur before T2 in order to break.

If too many or too few breakpoints are defined for terms T1-T3, an error message stating the cause of the error appears when processor execution begins. Breakpoints that are defined, but are disabled, do not count toward the number of breakpoints per term.






You must define a breakpoint to represent the term (T1-T3) desired. This is done via the **Add** or **Edit** buttons located in the **Breakpoints** dialog box (or via the **Add** or **Edit** menu item in the context menu). Click the **Add** button to define a new breakpoint; click the **Edit** button to modify the selected breakpoint. In the **Add Breakpoint** or **Edit Breakpoint** dialog box, define a bus analyzer break type. The **Sequence** field at the bottom of the dialog is enabled and the drop down list activates to allow terms T1-T3 to be selected. When the dialog box is closed (the breakpoint definition completed), the resulting breakpoint displays in the **Breakpoints** list box with the **Seq** column displaying the corresponding term information.

Resource Button

The **Resource** button lists the hardware and software resources available to you. The numbers vary, depending on the target processor. For more information, see "[Breakpoints Menu](#)," part of "Breakpoints Window Overview," under *Breakpoints Window*.

Breakpoints Window Icon Definitions

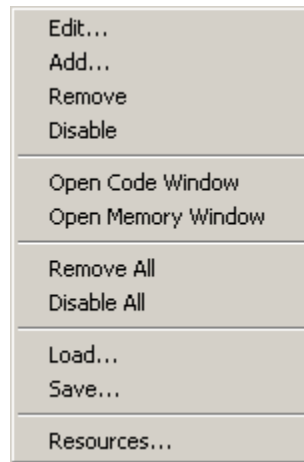
The icons you may find in the **Breakpoints** dialog box are as follows:

	Processor
	Front Side Bus
	Software
	Emulator (Reset)
	Special (SMM Entry/Exit)

Note: The **Breakpoint** icons are also displayed in the **Code** window.

Breakpoints Menu

Unlike many other windows, the **Breakpoints** window does not include a drop down menu from the menu bar. Instead, menu items are buttons on the window itself. A context menu may be accessed by right-clicking on the **Breakpoints** window.



Breakpoints context menu

Edit and Add menu items. The **Edit** and **Add** menu items serve the same function as the **Edit** and **Add** buttons on the **Breakpoints** dialog box. They take you to the **Edit Breakpoint** or **Add Breakpoint** dialog box, respectively.

For more information, see "[Edit Breakpoint and Add Breakpoint Dialog Boxes](#)," part of the "Breakpoints Window Overview" and "How to Edit or Add a Breakpoint in the Breakpoints Window," part of the "How To - Breakpoints," both under *Breakpoints Window*.

Remove and Disable menu items. These menu items are self-explanatory and serve the same function as the **Remove** and **Disable** buttons on the **Breakpoints** dialog box. You should consider disabling them if they will be used at a later time.

Open Code Window menu item. From a break on **Execute**, a **Code** window can be opened at the location of each breakpoint. This window can then be used to view surrounding code. To open a **Code** window displaying the location of a particular breakpoint, select the breakpoint, right-click to open the **Breakpoint** context menu, and select **Open Code Window**. A **Code** window opens with the breakpoint location indicated by a pointing hand.

Open Memory Window menu item. From a break on **Execute**, a **Memory** window can be opened at the location of each breakpoint. This window can then be used to view surrounding code. To open a **Memory** window displaying the location of a particular breakpoint, select the breakpoint, right-click to open the **Breakpoint** context menu, and select **Open Memory Window**. A **Memory** window opens with the breakpoint location indicated by a pointing hand.

Remove All and Disable All menu items. These menu items are self-explanatory.

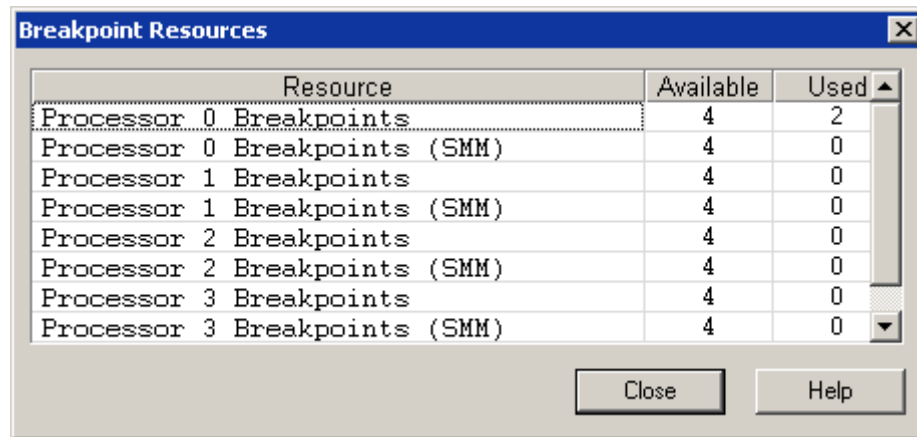
Load menu item. You can load the breakpoints you have saved in a ".brk" or ".prj" file by clicking on the **Load** menu item.

Note: Using this command replaces the breakpoints that currently exist in the **Breakpoints** text box.

Save menu item. Clicking on the **Save** menu item opens a **Save As** dialog box. From there you can create and save your current breakpoints in a ".brk" file.

Breakpoint Resources Dialog Box

Breakpoint Resources menu item. The **Breakpoint Resources** dialog box is made up of three columns: **Resource**, **Available**, and **Used**. For each breakpoint type, the maximum number of breakpoints available and the number currently in use are displayed. With most processors, you can have up to four debug register (Processor) breakpoints per processor, unlimited Software breakpoints, and six Bus Analyzer breakpoints.



Resource	Available	Used
Processor 0 Breakpoints	4	2
Processor 0 Breakpoints (SMM)	4	0
Processor 1 Breakpoints	4	0
Processor 1 Breakpoints (SMM)	4	0
Processor 2 Breakpoints	4	0
Processor 2 Breakpoints (SMM)	4	0
Processor 3 Breakpoints	4	0
Processor 3 Breakpoints (SMM)	4	0

Breakpoint Resources dialog box

Note: Bus Analyzer breakpoint resources are only available on Intel® Pentium® M or Intel® Celeron® M processors. Intel does not include this function in Intel® Pentium® 4 or Intel® Xeon® processors. With the Intel Pentium M processor, you can set only 3 bus breaks.

Breakpoint Types

The breakpoint types and the resources required for each breakpoint type are listed in the following table.

Break Type	Break Resource	
	IA-32/AMD64 w/o Trace	IA-32 with Trace
Data Access	Processor	Processor Bus Analyzer
Data Access in SMM	Processor	Processor
Data Read	N/A	Bus Analyzer
Data Write	Processor	Processor Bus Analyzer
Data Write in SMM	Processor	Processor
Execute	Processor Software	Processor Software
Execute in SMM	Processor	Processor
Fetch	N/A	Bus Analyzer
I/O Access	Processor	Processor Bus Analyzer
I/O Access in SMM	Processor	Processor
I/O Read	N/A	Bus Analyzer
I/O Write	N/A	Bus Analyzer
Instruction Set Transfer	N/A	N/A
Interrupt Acknowledge	N/A	Bus Analyzer
Reset	Emulator	Emulator
SMM Entry	Processor	Processor
SMM Exit	Processor	Processor
Special Transaction	N/A	Bus Analyzer

*Includes Intel 64-bit extensions to 32-bit processors.

Note: Depending on your hardware configuration, your breakpoint choices may be limited.

Processor (Debug Register) Breakpoints

Processor breakpoints rely on processor-specific registers to recognize events, such as instruction execution or data reads/writes at a memory or I/O address. Processor breakpoints cause the processor to stop immediately; there is little or no "slide" for non-execution breaks (i.e., breaks occurring on **Data Access**, **Data Write**, and **I/O Access** break on types). Pre-fetched but unexecuted instructions do not cause the processor to stop. The code location of an execution breakpoint can be in ROM. Each processor has a maximum of four processor breakpoints. Data values are not part of a breakpoint condition. Using **Processor** breakpoints may result in slower execution due to processor architecture.

When you are attached to a 32-bit processor, there are 10 processor breakpoint types: **Execute**, **Execute in SMM**, **Data Access**, **Data Access in SMM**, **Data Write**, **Data Write in SMM**, **I/O Access**, **I/O Access in SMM**, **SMM Entry**, and **SMM Exit**.

Processor breakpoints can be set in the **Add Breakpoint** or **Edit Breakpoint** dialog boxes or, for execution breaks (only), from the **Code** window.

Note: Processor breakpoints do not accept physical addresses.

Software Breakpoints

Software breakpoints are implemented by placing a special instruction (such as a software interrupt) in memory. Software breakpoints cause the processor to stop immediately (there is no "slide"). Software breakpoints do not stop the processor on unexecuted pre-fetches.

Software breakpoints are limited to execution breaks. The location of the instruction to be executed must be writable (i.e., located in RAM). Code at the breakpoint location cannot be loaded or modified on the fly. Care must be taken to insure breakpoints are set at the first byte of an instruction.

The only parameter required for this breakpoint type is **Location**. This parameter must be specified. The location specified for a break on Execute (Software) breakpoint must be in writable RAM. For more information on this kind of breakpoint, see the Intel data books.

After the **Go** command is issued, the instruction at the breakpoint location is replaced with a special instruction. When the processor stops, the original instruction is written back to the breakpoint location.

Note: If the processor writes a different (new) value to the breakpoint location before executing there, the breakpoint is ineffective until the processor is stopped and restarted with another **Go** command.

Emulator Breakpoints

Emulator breakpoints are set by using the break-on instruction **Reset..**

Bus Analyzer Breakpoints

Bus analyzer (bus event) breakpoints are based on processor bus activity. A bus event may be a single occurrence of a bus cycle with or without a certain status, address, or data pattern. It may also be a sequence of bus cycles with different patterns (i.e., a bus sequence breakpoint).

Bus event breakpoints do not use processor resources, nor do they impose restrictions as to where breakpoints may be placed. Bus event breakpoints allow triggering on locations, data values, interrupt acknowledges, and special bus transactions. Bus event breakpoints allow full speed sequential triggering.

Bus event breakpoints allow recognition of activity by other bus agents. They are unaffected by load-on-the-fly code.

Bus event breakpoints always result in "slide;" the processor does not stop until several instructions after the bus breakpoint sequence is satisfied. A **Bus Event** breakpoint triggers on a processor read of an instruction (fetch), even if the instruction does not execute. Finally, **bus event** breakpoints cannot detect activity that is confined to on-chip cache (no external bus activity).

The bus analyzer breakpoint types are: **Fetch**, **Data Access**, **Data Read**, **Data Write**, **I/O Access**, **I/O Read**, **I/O Write**, **Interrupt Acknowledge**, and **Special Transaction**. These are described in more detail below.

Fetch

The only parameter for this breakpoint type is location. The processor breaks after reading code from the memory address specified. If the **Location** field is left blank, the processor breaks on the next fetch cycle regardless of the address.

Data and I/O Access or Read/Write

The breakpoint types in this category are **Data Access**, **Data Read**, **Data Write**, **I/O Access**, **I/O Read** and **I/O Write**. The parameters for these breakpoint types are location, length, and data. If these parameters are left blank, the processor breaks on the first occurrence of the breakpoint type specified. **Location** must be specified (not left blank) in order to specify **Length** and **Data**.

Interrupt Acknowledge

The only parameter required for this breakpoint type is a vector; i.e., the vector number (0 through FFH) returned by the interrupting agent. If this parameter is left blank, the processor breaks on the first interrupt acknowledge cycle encountered. For more information on interrupt acknowledge cycles, see the applicable Intel data books.

Special Transaction

The only parameter required for this breakpoint type is one of the nine special bus cycles, as selected from the **Cycles** drop down list on the **Add Breakpoint** or **Edit Breakpoint** dialog boxes. Each special bus cycle is described below. For more information on special bus cycles, see the applicable Intel literature.

P6-Class Special Transactions

Special Cycle	Description
Any Cycle	Processor breaks on first special transaction encountered
NOP	No operation for one or more clocks

Shutdown	Exception during call of double fault handler
Flush	Internal caches invalidated, modified lines not written back
Halt	Execution of HLT instruction
Sync	Execution of WBINVD instruction
Flush ACK	Result of cache sync and flush (FLUSH# pin assertion)
Stop CLK ACK	Issued when Stop Clock mode is entered
SMI ACK	Issued upon the entry or exit of the SMM handler

Intel Pentium Processor Special Transactions

Special Cycle	Description
Any Cycle	Processor breaks on first special transaction cycle encountered
Shutdown	Exception during call of double fault handler
Flush	Internal caches invalidated, modified lines not written back
Halt	Execution of HLT instruction
Writeback	Due to WBINVD instructions
Flush ACK	Result of cache sync and flush (FLUSH# pin assertion)

Edit Breakpoint and Add Breakpoint Dialog Boxes

To add a new breakpoint, click on the **Add** button in the **Breakpoints** main window. The **Add Breakpoint** dialog box displays. Enter parameter information to the displayed fields and click the **OK** button to save the changes and return to the **Breakpoints** dialog box. The breakpoint will display in the **Breakpoints** list box.

To edit a breakpoint, highlight it from the **Breakpoints** list box, and click the **Edit** button. The **Edit Breakpoint** dialog box displays. Edit the parameters as desired, and click the **OK** button to save the changes and return to the **Breakpoints** dialog box. The breakpoint displays in the **Breakpoints** list box.

*Apart from the title of the window, the **Add Breakpoint** and **Edit Breakpoint** dialog boxes look the same and operate in the same fashion.*

Identifier Text Box

The **Identifier** text box lets you specify an identifier for a breakpoint. If no value is entered, a default identifier (event# where # is some number) is used.

Binary Buttons

Binary buttons are available that apply to the **Location**, **Data**, and **External** fields. These buttons open dialog boxes that allow you to edit a field value in binary, and also to "Don't Care" individual bits.

Break On Drop Down List

The type of activity on which the processor is to break is selected from the **Break on** drop down list. Each break activity is associated with resources listed in the **Resource** drop down list. These resources implement a break according to the break on activity type and the parameters specified.

Note: The **Advanced** button opens additional features. These features are not available without a specific non-disclosure agreement from Intel. Contact Arium for details.

For more information on breakpoint types (listed in the **Break on** drop down list), please refer to the topic, "[Breakpoint Types](#)," part of "Breakpoints Window Overview," found under *Breakpoints Window*.

Resource Drop Down List

The **Resource** drop down list contains the resources used to implement selected break on activities and their parameters. Of the four resources available (**Processor**, **Software**, **Bus Analyzer**, and **Emulator**), the **Resource** drop down list displays only those resources that can be used with the selected break on activity. Breakpoint activities associated with the **Processor** resource are debug register breakpoint types; breakpoint activities associated with the **Software** resource are software breakpoint types. Breakpoint activities associated with the **Bus Analyzer** resource are bus event breakpoint types.

Some activities have only one resource available, while others have a choice of resources. For example, a break on **Execution** activity can be implemented by choosing **Processor** or **Software** as the resource. A break on **Data Read** activity, however, can be implemented only by the **Bus Analyzer** resource.

If the breakpoint resource is **Bus Analyzer**, the **Advanced** button is enabled if you have an NDA with Arium. (This is determined via the certification file you are sent with the system.)

Processor Drop Down List

SourcePoint supports up to eight processors (P0 – P7). **Processor**-type breakpoints can be set on individual processors or all of them.

Location Text Box/Cycle Drop Down List/Vector Text Box

Depending on the resource and breakpoint type you select, the **Add Breakpoint** or **Edit Breakpoint** dialog box displays a **Location** text box, **Cycle** drop down list, or **Vector** text box.

The **Location** text box is where the logical, linear, or physical address of the memory or I/O location of the processor break is entered. Memory locations are limited to 32 bits of address; I/O locations to 16 bits.

Note: When specifying a bus breakpoint, leaving the **Location** text box blank causes the address field to be "Don't Care." The processor then breaks on the first occurrence of a bus breakpoint, regardless of the address.

Note: If an odd word address is specified, the effective address is the address 1. For example, specify a word at location 3459 to include the bytes at locations 3458 and 3459. Specify a doubleword at location 5677 to include the bytes at locations 5674 through 5677. If a doubleword address is not divisible by four, the effective address that is used will be the next lowest address that is divisible by four.

The **Cycle** drop down list is available for bus breakpoints only, and only when you specify a **Special Transaction** break. Otherwise, it is replaced with the **Location** field. Several options are available in the **Cycle** drop down list, depending on the type of breakpoint. They are: **Any Cycle**, **Shutdown**, **Flush**, **Halt**, **Sync**, **Flush ACK**, **NOP**, **Stop CLK ACK**, and **SMI ACK**.

The **Vector** field is available for bus analyzer breakpoints only, and only when you specify a break on **Interrupt Acknowledge**. Otherwise, it is replaced with the **Location** field. Valid entries in the **Vector** text box are 8-bit interrupt vector values from 0 to 255.

Note: The **Binary** button opens dialog boxes that allow you to edit a field value in binary, and also to "Don't Care" individual bits.

Translate Drop Down List

Two options are available per breakpoint in the **Translate** drop down list. By selecting **Once**, the address is translated to a linear address from the current content of the processor at the time the breakpoint is created. By selecting **Every Go**, the address translation is calculated every time the processor is started. When a breakpoint address is specified, it is difficult to know if it should be translated immediately, using the current processor context, or if it should be translated every time the processor is started. To allow maximum flexibility in setting breakpoints, SourcePoint allows both kinds of address translation. The **Translate** field specifies which kind of address translation to perform. **Once** indicates that the address is translated immediately, using the current processor context. **Every Go** indicates that the addresses is re-translated every time the processor is started.

Length Drop Down List

For processor breakpoints, the **Length** drop down list is available when you specify a break on **Data Access**, **Data Write**, or **I/O Access**. For **Processor** breakpoints, select **Byte**, **Word**, or **Dword** from this list to establish an access range. For **Bus Analyzer** breakpoints, the drop down list is available when you specify a location for a break on **Data Access**, **Data Read/Write**, **I/O Access**, or **I/O Read/Write**. Otherwise, the **Length** parameter is dimmed and does not apply in this case. For **Bus Analyzer** breakpoints, select **Byte**, **Word**, or **Dword** to specify the minimum width of the data transaction.

Data Text Box

The **Data** text box is available for **Bus Analyzer** breakpoints only, and only when you specify a location for a break on **Data Access**, **Data Read/Write**, **I/O Access**, or **I/O Read/Write**. Otherwise, the **Data** parameter is dimmed and unavailable for use.

Note: The **Binary** button opens dialog boxes that allow you to edit a field value in binary, and also to "Don't Care" individual bits.

External Text Box

Currently not available.

Sequence Drop Down List

The **Sequence** drop down list is used to specify breakpoint sequence types. Valid sequence options are T1-T3. The default selection of this field is **Not in Sequence**. The selection determines if and how the breakpoint is involved in breakpoint sequencing or data qualification.

Macro Text Box

The **Macro** text box and associated **Browse** button, available for *Execute* and *Execute in SMM* breakpoints, displays the macro associated with the breakpoint. The macro is run when the breakpoint hits. A blank filename indicates no macro is associated with the breakpoint. A file name can be added to the box by keying in the name of the macro or using the **Browse** button to select a file. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly.

Breakpoints Window Preferences

Go to **Options|Preferences** on the menu bar and select the **Breakpoints** tab to select Breakpoints window preferences options. For detailed information, see "[Options Menu - Preferences Menu Item](#)" part of "SourcePoint Overview" under *SourcePoint Environment*.

How To - Breakpoints

How to Disable and Enable Breakpoints in the Breakpoints Window

Note: Breakpoints are enabled by default when added to the **Breakpoints** list. The processor breaks on any enabled breakpoint in the list. Currently unused breakpoints need not be removed to be ignored, simply disabled.

To Disable a Breakpoint

1. Go to **View|Breakpoints** on the menu bar.
2. In the **Breakpoints** dialog box, click on the breakpoint you want to disable from the **Breakpoints** list area.
3. Click the **Disable** button.

To Enable a Breakpoint

1. Go to **View|Breakpoints** on the menu bar.
2. In the **Breakpoints** dialog box, click on the breakpoint you want to enable from the **Breakpoints** list area.
3. Click the **Enable** button.

Note: Breakpoints can also be enabled/disabled by using your mouse to click on the icon directly to the left of the location column. This works as a toggle.

Note: Disabled breakpoints do not count as break resources and, therefore, are not considered in the displayed counts on the **Breakpoint Resources** screen.

How to Remove Breakpoints

Removing a Single Breakpoint

1. Go to **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Choose the breakpoint to remove from the **Breakpoints** list box.
3. Click the **Remove** button.

The specified breakpoint will be deleted from the displayed list immediately upon clicking the **Remove** button.

Note: There will be no warning message after the **Remove** button is clicked. The command cannot be undone.

Removing All Breakpoints

1. Go to **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Click the **Remove All** button.

Note: There is no warning message after the **Remove All** button is clicked. Be sure that you want ALL breakpoints to be removed from the **Breakpoints** list area PRIOR to clicking the **Remove All** button. The command cannot be undone.

How to Set a Breakpoint on an Interrupt

Setting a Breakpoint in Real Mode

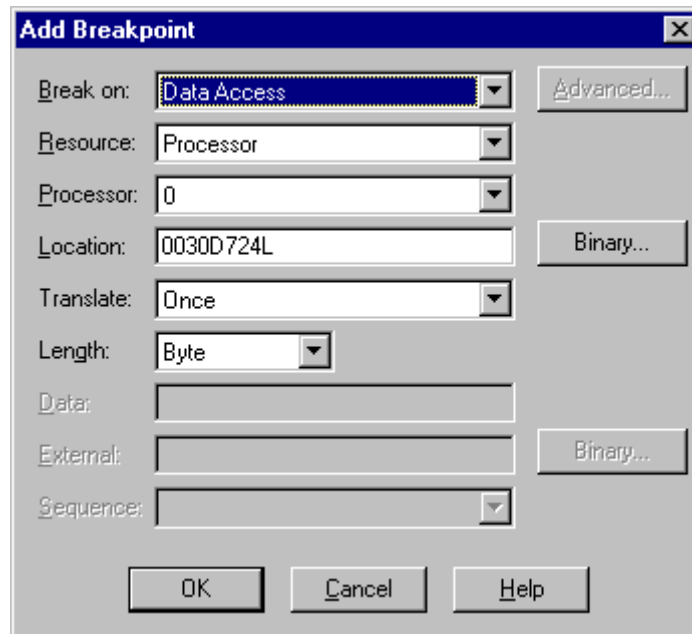
1. In Real mode, choose **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Click on the **Add** button.

The **Add Breakpoint** dialog box opens.

3. Select **Data Access** from the **Break on** drop down list.



Add Breakpoint window

4. Fill in the **Location** field with the address that equals the vector location (where the $\text{Int \#} * 4 = \text{the vector location}$).

For example, to break on Interrupt 3, use the following equation: $\text{Int \#} * 4 \text{ (bytes)} = 0\text{Ch}$.

Selecting a Breakpoint in the Protected or v86 Mode

1. For Protected mode or v86 mode, go to **View|Descriptors** on the menu bar.

The **Descriptors** table window opens.

2. Click on the **IDT** tab to view the descriptor entry.
3. Select the entry of interest to determine the address of the access.

The **Values** column contains the selector value and offset that make up the address.

IDT Descriptors [Base 00036400L Limit 07FF]					
	Offset	Type	Attributes		Values
0	0000	Gate	Trap Gate 32-bit P DPL=0		S=0008 O=000012C0 C=0
1	0008	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=000012CF C=0
2	0010	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=000012DE C=0
3	0018	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=000012ED C=0
4	0020	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=000012FC C=0
5	0028	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=0000130B C=0
6	0030	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=0000131A C=0
7	0038	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001329 C=0
8	0040	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001338 C=0
9	0048	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001345 C=0
10	0050	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001354 C=0
11	0058	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001361 C=0
12	0060	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=0000136E C=0
13	0068	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=0000137B C=0
14	0070	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001388 C=0
15	0078	Gate	Trap Gate 32-bit P DPL=0		S=0058 O=00001395 C=0
16	0080	Reserved			H=00000000 I=00000000
17	0088	Reserved			H=00000000 I=00000000
18	0090	Reserved			H=00000000 I=00000000

Interrupt Descriptor Table

For more information on Descriptors Tables, start with "[Descriptors Window Introduction](#)," part of "Descriptors Tables Window Overview," found under *Descriptor Tables Window*.

How to Set a Bus Breakpoint

1. Select **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Highlight a previously defined breakpoint and click the **Edit** button, or click the **Add** button to define a new breakpoint.

The **Edit Breakpoint** or **Add Breakpoint** dialog box appears.

3. Select the appropriate bus breakpoint type from the **Break on** drop down list.
4. Select **Bus Analyzer** from the **Resource** drop down list.
5. In the **Location** list box, enter the logical, linear, or physical address at which you want the processor to break.
6. Select **Once** or **Every Go** from the **Translate** drop down list.
7. Select **Byte**, **Word**, or **Dword** from the **Length** drop down list.
8. In the **Data** list box, enter the desired data value.

Note: When an address or data field is left blank, it represents a "Don't Care."

9. Select **T1-T3** from the **Sequence** drop down list. (**T1** is selected by default.)
10. Click the **OK** button.

The **Breakpoints** dialog box displays again. The breakpoint has been enabled and added to the **Breakpoints** list box.

11. To activate the breakpoint, select **T1** from the **Type** drop down list in the **Bus Analyzer: Sequence** field at the bottom of the **Breakpoints** dialog box.

How to Set an Emulator Breakpoint

There are three **Emulator** breakpoint types: **Reset**, **SMM Entry**, and **SMM Exit**. These breakpoints are dedicated hardware resources in the emulator, and, therefore, do not use **Processor**, **Software**, or **Bus Analyzer** resources. Also, entries for these breakpoints do not display in the **Breakpoint Resources** screen. They are set simply by selecting them in the **Break on** list in the **Add Breakpoints** or **Edit Breakpoints** dialog box; no parameters are required.

1. Go to **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Highlight a previously defined breakpoint and click the **Edit** button, or click the **Add** button to define a new breakpoint.

The **Edit Breakpoint** or **Add Breakpoint** dialog box appears.

3. Select **Execute** from the **Break on** drop down list.
4. Select **Software** from the **Resource** drop down list.
5. In the **Location** list box, enter the logical, linear, or physical address at which you want the processor to break.

Note: This address must be the first byte of the instruction.

6. Click the **OK** button.

The **Breakpoints** dialog box displays again. The breakpoint has been enabled and added to the **Breakpoints** list box.

How to Set a Processor (Debug Register) Breakpoint

1. Go to **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Highlight a previously defined breakpoint and click the **Edit** button, or click the **Add** button to define a new breakpoint.

The **Edit Breakpoint** or **Add Breakpoint** dialog box appears.

3. Select the appropriate **Processor** breakpoint type from the **Break on** drop down list.
4. Select **Processor** from the **Resource** drop down list.
5. In the **Location** list box, enter the logical or linear address at which the processor will break.
6. Select **Byte**, **Word**, or **Dword** from the **Length** drop down list.
7. Click the **OK** button.

The **Breakpoints** dialog box displays again. The breakpoint has been enabled and added to the **Breakpoints** list box.

To set a Processor (Debug Register) Breakpoint From the Code Window

1. Go to **View|Code** on the menu bar.

The **Code** window displays.

2. Insert the blinking caret anywhere in the line of code that you have selected as a breakpoint.
3. Click on the **Set Break** button on the local toolbar.

A red rectangle appears in the breakpoint icon column to mark the breakpoint.

Note: To view the breakpoint in the **Breakpoints** dialog box, go to **View/Breakpoints** on the menu bar.

How to Set a Software Breakpoint

1. Go to **View|Breakpoints** on the menu bar.

The **Breakpoints** dialog box displays.

2. Highlight a previously defined breakpoint and click the **Edit** button or click the **Add** button to define a new breakpoint.

The **Edit Breakpoint** or **Add Breakpoint** dialog box appears.

3. Select **Execute** from the **Break on** drop down list.
4. Select **Software** from the **Resource** drop down list.
5. In the **Location** list box, enter the logical, linear, or physical address at which the processor breaks.

Note: This address must be the first byte of the instruction.

6. Click the **OK** button.

The **Breakpoints** dialog box displays again. The breakpoint has been enabled and added to the **Breakpoints** list box.

Code Window

Code Window Overview

Code Window Introduction

Select **View|Code** on the menu bar or click on the **Code** icon on the icon toolbar to access the **Code** window. The **Code** window is used to view code at specific addresses, run the processor, and track program execution. Various functions include setting and viewing breakpoints, viewing **Disassembly**, **Mixed**, and **Source** modes, and viewing existing data values in registers and memory locations. Multiple **Code** windows can be open simultaneously.

Code window

Note: You can cursor to addresses before the IP only if source code and/or symbols are loaded.

Display Columns

The **Code** window has four line-oriented display fields: **Address**, **Object Code**, **Mnemonic**, and **Operand**.

Address. The **Address** field contains the code segment (CS) selector and the code segment offset (EIP) for each instruction's address.

Object Code. The **Object Code** field contains the instruction's object code as read from memory. This field is toggled on and off using the **Code Bytes** menu item from **Code|Display|Code Bytes** on the menu bar.

Mnemonic. The **Mnemonic** field contains the instruction mnemonic as disassembled from memory.

Operand. The **Operand** field contains the operands involved in the instruction.

Dialog Bar

Address text box. In the lower left-hand corner of the window, any valid code address can be entered to disassemble that location in memory.

Code View drop down list box. In the box to the right of the **Address** text box is the **Code View** drop down list box. Choices are **Disassembly**, **Mixed**, and **Source**. **Disassembly** simply reads memory and displays the opcodes and data as mnemonics and operands. A **Mixed** selection displays a mixed source code/memory disassembly. A **Source** selection displays the source code only.

Go cursor button. This function sets a temporary breakpoint at the cursor location in the **Code** window and starts a processor.

Set/Clear Break button. This button sets or clears the execution breakpoint at the cursor location in the **Code** window.

Track IP check box. If **Track IP** is checked, the **Code** window always displays code at the processor instruction pointer (IP). If the processor stops beyond the address range currently visible in the **Code** window, the **Code** window is redrawn showing the code at the new IP. If **Track IP** is not enabled, the **Code** window retains its contents and address range, and a new **Code** window is opened when the processor stops if no other **Code** window with **Track IP** enabled is already opened. In effect, if **Track IP** is enabled, the window always shows a range of code that includes the IP. If it is disabled, you can step through code until the IP is not in the range of code visible to the **Code** window without the display changing to the new IP location.

View IP button. This button displays the code at the current instruction pointer location.






Refresh button. Click on the **Refresh** button to update the **Code** window by re-reading from target memory the instructions in the current address range. This menu item is useful when code resides in RAM and may be subject to change.

Finding Source Code

If SourcePoint cannot find your source code, a **Find Source** dialog box displays that lets you point SourcePoint to your source code.

Find Source dialog box

Code Window Icon Definitions

Breakpoints	
	Processor
	Software
	Bus
Pointers	
	Instruction pointer
	Pointer from another window (e.g., Trace)

Note: Pointers may appear on top of breakpoint icons when both apply at the same point. Only one type of breakpoint icon is shown at a time for a particular point.

Code Window Menu

Once a **Code** window is open, a **Code** menu displays on the SourcePoint menu bar. The same menu can be accessed as a context menu by right-clicking within the **Code** window.



Code window menu

Source [Code] menu item. Select the **Source** menu item to enable you to view C or assembler source code. **Source** functions the same way as the **Source** option on the **Code View** drop down list on the dialog bar of the **Code** window.

Mixed [Code] menu item. Select **Mixed** to concurrently view both source code and the associated processor instructions as disassembled from memory. The **Mixed** menu item functions the same way as the **Mixed** option on the **Code View** drop down list on the dialog bar of the **Code** window.

Disassembly [Code] menu item. Select **Disassembly** to view processor instructions as disassembled from memory. The menu item functions the same way as the **Disassembly** option on the **Code View** drop down list on the dialog bar of the **Code** window.

Open Code Window menu item. This menu item allows you to click on a function and open a second **Code** window displaying its code.

Open Memory Window menu item. When this menu item is selected, the field at the current caret position in the **Code** window is evaluated as a data address, and a **Memory** window is opened at that address. This includes addresses, symbols, operand values, register values, and constants.

Copy to Watch menu item. This menu item allows you to copy a variable name, register name, or expression to a **Watch** window.

Quick Watch menu item. This menu item allows you to copy a variable name, register name, or expression to a **Quick Watch** window.

Set/Clear Breakpoint menu item. Select **Set Breakpoint** or **Clear Breakpoint** (the menu items toggle) to set or clear a breakpoint quickly and easily.

Set Alternate Breakpoint menu item. In the **Breakpoints** tab of the **Options|Preferences** dialog box, you selected a default type. **Set Alternate Breakpoint** lets you override that default on a one-time basis without having to change the default in the **Breakpoints** tab.

Enable/Disable Breakpoint menu item. Select **Enable Breakpoint** or **Disable Breakpoint** (the menu items toggle) to enable or disable processor register breakpoints at the caret position.

Add/Edit Breakpoint menu item. Place the caret at the position on the **Code** window where you want to add or edit a breakpoint. From the options menu, click **Add Breakpoint** or **Edit Breakpoint** (the menu items toggle) to bring up the **Breakpoints** window. Click on the **Add** or **Edit** button to access the **Add Breakpoint** or **Edit Breakpoint** dialog box.

For more information on how to add or edit a breakpoint, see "[Edit Breakpoint and Add Breakpoint Dialog Boxes](#)," part of "Breakpoints Window Overview," found under *Breakpoints Window*.

Go Until Cursor menu item. Select **Code|Go Until Cursor** to set a temporary breakpoint at a caret position and let the processor run (starting at the current instruction pointer) until it encounters a breakpoint. The **Go Until Cursor** menu item functions the same way as the **Go Cursor** button on the **Code** window dialog bar.

Set IP menu item. Select **Set IP** to change the EIP quickly and easily. The EIP value is modified to reflect the selected instruction, and the yellow EIP icon is moved to the instruction, as well. Applications for this feature include skipping over instructions (without executing them) or re-executing previously executed instructions.

Display menu item. Select the **Display** menu item to access the following options:

- **Line Number/Address.** Changes the display of line numbers (**Source** or **Mixed**) and instruction addresses (**Mixed** or **Disassembly**). When enabled, line numbers and/or instruction addresses are shown. When disabled, the line number and/or instruction addresses are not shown.
- **Code Bytes.** Toggles the display of an instruction object code field in the **Code** window.
- **Symbols.** Displays/hides symbols.
- **Pseudo-Ops.** Pseudo-ops are mnemonics that appear like register or instruction names but are really shorthand for a more memorable name.
- **Annotation.** Indicates boundaries between source files and areas of memory that have no corresponding source. All annotation lines have a line of underscores before and after the annotation text.

- **Line Highlights.** Options are **Current IP** or **None**.
- **Disassembly Case.** Options are **Mixed**, **Upper**, and **Lower**.
- **Radix.** Options are **Command Default**, **Binary**, **Octal**, **Decimal**, and **Hexadecimal**.
- **Radix Indicators.** Options are **Prefix**, **Suffix**, and **None**.
- **Tab Spacing.** Options are **2**, **3**, **4**, **5**, **6**, **7**, and **8**.

Refresh menu item. Select **Refresh** to update the **Code** window by re-reading from target memory the instructions in the current address range. This menu item is useful when code resides in RAM and may be subject to change. The **Refresh** menu item found within the **Code** menu functions the same way as the **Refresh** button on **Code** window dialog bar.

Disassembly Mode menu item. Select **Disassembly Mode** to select the 16- or 32-bit instruction set for disassembly purposes. Options are **Current Processor Default**, **16-bit**, and **32-bit**.

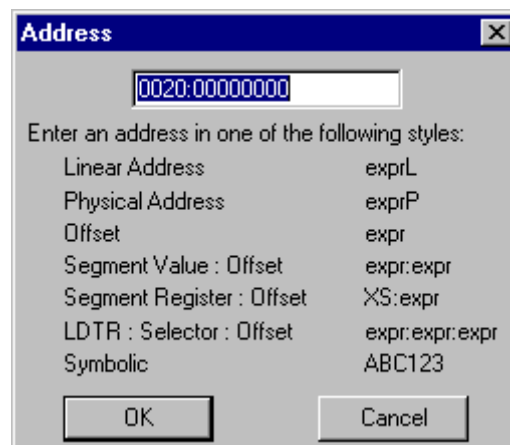
Disassembly Uses menu item. Use this menu item to determine whether you want to view target memory or cached memory associated with disassembly mode. Options are **Target Memory** and **Cached Program**.

- **Target Memory** causes disassembly operations to read target memory. Use this mode if there is a potential for the code space in target memory to be changing while the target is running.
- **Cached Program** allows you to view disassembly without reading target memory. When this option is enabled, SourcePoint reads target memory from a cached copy, thus eliminating the need to refresh the **Code** window. Enabling this option minimizes the use of resources and speeds up single stepping.

Note: Program caching only works for Elf/Dwarf files.

Address menu item. Select the **Address** menu item to modify the view within the **Code** window. Options are **Track IP**, **View Code at Address**, and **View Code at IP**.

- **Track IP.** When this option is selected, it toggles the function of the **Code** window to always show the address of the IP.
- **View Code at Address.** When this option is selected, the **Address** dialog box displays. When an address is entered in the text box, it causes the **Code** window to bring this address into view.



*View Code at **Address** text box*

Address Style	Description
Linear Address (exprL)	Real or Protected mode.
Physical Address (exprP)	Real or Protected mode (same as the linear address if paging is not in effect).
Offset (expr)	Offset relative to selector CS.
Segment Value: Offset (expr:expr)	Value selected for segment plus value selected for offset.
Segment Register: Offset (XS:expr)	Uppercase designation for CS, DS, ES, FS, GS, or SS register plus value selected for offset.
LDTR: Selector: Offset (expr:expr:expr)	Value selected for LDTR plus values selected for selector (segment register) and offset. (This style is used in Protected mode only.)
Symbolic	When symbols are loaded through SourcePoint, this option is available.

- **View Code at IP.** When this option is selected, it causes the **Code** window to bring the IP address into view if it is not currently showing.

Viewpoint menu item. This menu item indicates the status of the processor viewpoint. If you have enabled one of the processor options, that processor is tracked. If you have enabled the **Track Viewpoint** option, the current processor is tracked.

Copy menu item. This menu item allows you to copy data from the **Code** window to another source (e.g., Notepad).

Add Code Profiling Function(s) menu item. Not functional.

Code Window Preferences

To set preferences for the **Code** window, go to **Options|Preferences** and select the **Code** tab. For details, go to the topic entitled, "[Options Menu - Preferences Menu Item](#)," found under "SourcePoint Overview," part of *SourcePoint Environment*.

How To - Code Window

How to Open a Code Window

Opening the First Code Window

1. Reset your target by clicking the **Reset** button on the icon toolbar or go to **Processor|Reset** on the menu bar. The **Code** window displays.

Opening Additional Code Windows With the Code Menu Item

Repeatedly go to **View|Code** on the menu bar or click on the **Code** icon on the icon toolbar several times to open the desired number of **Code** windows.

Open a Code Window Corresponding to a Disassembled Instruction From the Trace Window

1. In the **Trace** window, position the caret on the instruction in question.
2. Go to **Trace|Open Code Window** on the menu bar or right click in the Trace window to access the context menu and click on the **Open Code Window** menu item..

Note: If a program with source code has been loaded, the source code corresponding to the disassembly is shown. The **Code** window becomes a tracking **Code** window that updates its location every time the caret is repositioned in the **Trace** window.

Opening Additional Code Windows From the Symbols Window

1. Go to **View|Symbols** on the menu bar.
2. Select a function in the **Symbols** window and double-click the mouse.

A **Code** window displays.

3. Repeat steps 1 - 3 as necessary to open the desired number of **Code** windows.

How to Disassemble Code at a Specific Location

Code disassembly can be viewed in the **Command** window or in the **Code** window.

Disassemble Code in the Command Window

1. Go to the **Command** window.
2. At the prompt, type "asm cs:ip length 3" (without the quotation marks).

This disassembles three instructions in memory at the current IP.

Disassemble Code Using the Code Window

1. Open a **Code** window.
2. Type in a valid address in the lower left-hand corner text box on the dialog bar.
3. Press Enter or Return on your keyboard.

OR

1. Right-click the mouse to use the **Code** context menu.
2. Select **Address|View Code at Address**.
3. Type in a valid address in the **Address** dialog box.
4. Click on the **OK** button.

How to Save Code Window Settings

To save the position, size, and parameter settings of the **Code** window (and any other open window):

1. Go to **File|Save As** on the menu bar.

The **Save As** dialog box displays.

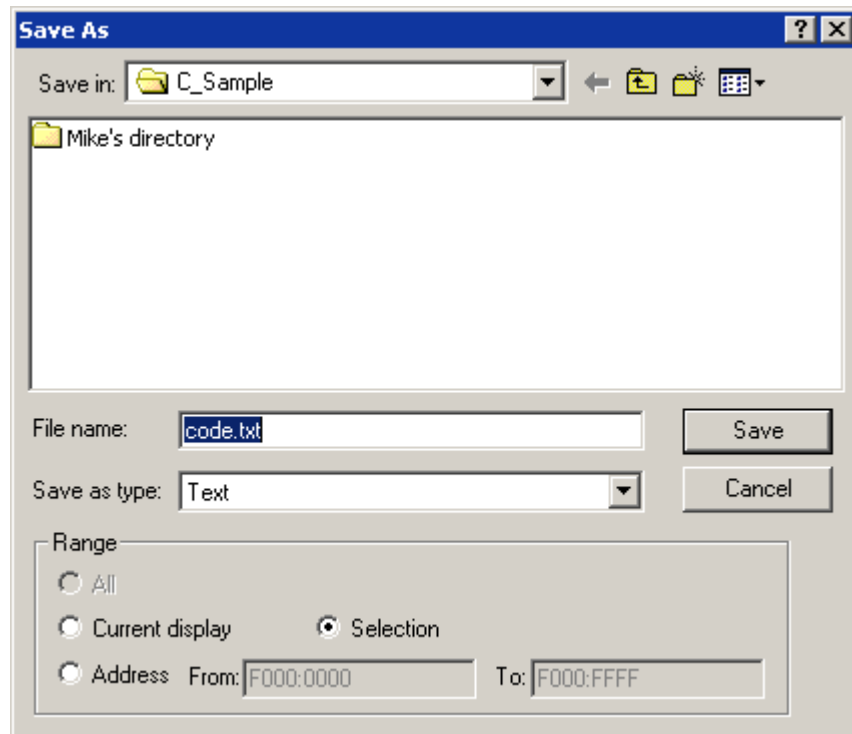
2. Enter a file name with a ".prj" extension
3. Click the **OK** button.

Note: Displayed data are not saved, as new data are read from the processor each time the **Code** window is opened.

How to Save Code Window Contents

1. Go to **File|Save As** on the menu bar.

The **Save As** dialog displays.



Save As dialog box

2. Specify **File name** and the **Range** of addresses to save.

Note: Specifying a large range may take a significant amount of time saving to a file depending on how large the range is.

Command Window

Command Window Overview

Command Window Introduction

Select **View|Command** from the menu bar or click on the **Command** icon on the toolbar to display the **Command window**.

Command window

The elements of the **Command window** include a blinking caret that resides on a command line prompt. When a command line other than the last line (at the bottom of the **Command** window) is edited, it turns from black to green to indicate that it has been changed. The response lines are shown in blue and are not editable.

Cursor movements via the arrow keys can be made to editable fields only. The mouse allows movement to any field. For instance, if the cursor is at the beginning of the prompt line in the example above, then pressing the left arrow key once places the cursor at the end of the line that contains "P0>."

Since up arrow and down arrows are interpreted as history retrieval commands, you must move away from the prompt line in order to use them as cursor movement commands. Pressing the left arrow key while at the beginning of the prompt line takes the cursor back to the previous command line, as described above. Further cursor movement can be accomplished with repeated use the arrow keys.

Any command line may be edited (even one that is back in history). New characters can be inserted at the current cursor location. The Delete key deletes the character immediately to the right. The Back (backspace) key deletes the character immediately to the left of the cursor.

The Home key takes the cursor to the beginning of the line. The end key takes the cursor to the end of the line.

Command Syntax and Conventions

The following notational conventions must be followed when entering commands into the **Command** window.

- Verbatim text appears in lowercase and must be entered exactly as shown except that case is not significant unless so noted.
- Italics indicate placeholders. Substitute a value or a symbol for these items.
- Braces enclosing items separated by a single pipe (|) indicate that one of the items must be selected.

For example: {item 1 | item 2 | item n}

- Brackets enclosing items separated by a single pipe (|) indicate that one of the optional items can be selected or all can be omitted.

For example: [item 1 | item 2 | item n]

- Ellipses enclosed in parentheses indicate that the preceding item can be repeated or that more than one optional item can be chosen.

For example: (...)

- Punctuation must be entered exactly as shown, except for brackets ([]) and ellipses (...) are used to denote syntax.

Note: Semicolons are optional everywhere except where two or more commands are on a single line; then the semicolon is mandatory between commands.

- Use standard prefixes and suffixes to identify the base.

Prefix	Base	Suffix
0y	Binary	y
0o	Octal	q,o
0n	Decimal	n,t
0x	Hexadecimal	h

Command Window Menu

Once a **Command** window is open, a **Command** menu displays on the SourcePoint menu bar. The same menu can be accessed as a context menu by right-clicking within the **Command** Window.

Command menu

Execute Command menu item. Use **Execute Command** to execute a command on the line with the cursor (caret). You can also execute a command by pressing the Enter key. When a command is executed, it is echoed at the bottom of the **Command** window and then is followed by the lines of output that it produces. The window is scrolled so that the last line is visible.

Commands are "remembered" for later recall using the command history manipulation described below. A command is saved for later recall if and only if it is different from the previously saved command. Thus, redundant commands do not waste storage. About one hundred commands are saved in each session. Commands are not saved between sessions.

Clear Command menu item. Use **Clear Command** to clear a command entered on the line and place the cursor on the bottom of the **Command** window.

Cancel Command menu item. Use **Cancel Command** to delete a command under the cursor and place the cursor at the bottom of the **Command** window. This action may scroll the window.

Clear Command Window menu item. This menu item is self-explanatory.

Retrieve Oldest Command menu item. Use **Retrieve Oldest Command** to retrieve the first command in the current work session. Alternatively, pressing Page Up from the keyboard performs the same function.

Retrieve Most Recent Command menu item. Use **Retrieve Most Recent Command** to retrieve the most recent command. Alternatively, from the **Command** window, use the Page Down key to scroll through the list of entered commands to retrieve the most recent command.

Retrieve Previous Command menu item. Use **Retrieve Previous Command** to retrieve the previous command. Alternatively, from the **Command** window, use the Up Arrow key to scroll through the list of entered commands to retrieve the previous command. As you continue to scroll through the list, all commands entered during the current work session display.

Retrieve Next menu item. Use **Retrieve Next Command** to retrieve the next command. Alternatively, from the **Command** window, use the Down Arrow key to scroll through the list of entered commands. As you continue to scroll through the list, all commands entered during the current work session display.

Cut, Copy, Paste menu items. These are standard Microsoft® Windows® commands.

How To - Command Window

How to Abort a Command

To abort a command, use the Ctrl+Break keys.

Note: The **Command** window error message, "Command execution must be serial," indicates that a command is executing currently and must be completed or aborted before new commands can be entered. This commonly occurs when using the looping commands **For** and **While**.

How to Display and Delete Procs

In the **Command** window, type **Show proc** or **Remove proc** at the prompt. The **Show proc** command displays all procs that have been loaded into memory. The **Remove proc** command deletes procs from memory. See the examples below.

Command input:	
<i>show proc</i>	<i>// displays all defined procedures</i>
<i>show proc t*</i>	<i>// displays all procedures that start with 't'</i>
<i>proc tproc</i>	<i>// displays full definition of procedure tproc</i>
<i>remove proc t*</i>	<i>// removes all procedures that start with 't'</i>

How to Edit a Macro From a Command Window

If you have created a macro you want to edit, you may do so from the **Command** window. To do so:

1. Type "edit" (without the quotation marks) and specify a filename (e.g., "edit foo.txt" - without quotation marks) at the command prompt or type "edit" (without the quotation marks) and, when your editor opens, select **File|Open** to select your file.
2. Make your changes.
3. Save your changes and close the file.
4. Load the macro. (For details on loading macros, see "[How to Load Macros and Procs](#)," part of "How To - Command Window" found under *Command Window*.)

How to Load Macros and Procs From a Command Window

The following methods may be used to load a macro file that contains macros and/or procs:

- Select **File|Macro|Load Macro** from menu bar.
- Click **Load Macros** icon on the toolbar
- Command line (**include** statement)

Command input:

include foo.txt

Note: You can embed multiple **Include** commands in a macro file to load multiple files at once.

How to Use REXX to Automate and Control SourcePoint

REXX is a cross-platform programming language that was developed in the 1970s. Although it is a robust language capable of being used for application development, it is primarily used to control other applications and system services. SourcePoint has been designed with an interface that allows REXX programs to provide control over SourcePoint. This allows Arium in-circuit emulators to be used in test and validation environments where automation of a process is needed.

The interface between REXX and SourcePoint is straightforward. REXX programs are able to pass the same commands to SourcePoint that can be entered on the SourcePoint command line and receive back any results or output from the execution of these commands. Additionally, any parameters from the SourcePoint command line that invoked the REXX program can be passed as well.

In order for this feature to work properly, you must have objectREXX installed on your system. Only the interpreter version of objectREXX is necessary to control SourcePoint using objectREXX programs.

Note: The tools and libraries that come with objectREXX v 1.0 are not available with the standard version of REXX.

REXX programs are invoked from the SourcePoint command line as described below.

Syntax

REXX Myprog.rex(ParameterList)

Where:

REXX The SourcePoint command that invokes the REXX program.

Myprog.rex Pathname of the REXX program being invoked.

ParameterList A comma-delimited list of parameters being passed as strings to the REXX program.

Passing Commands and Receiving Results

As mentioned previously, REXX passes commands to SourcePoint for execution just as if they were entered directly on the command line. Any result or output from the SourcePoint command can be returned to the REXX program. This is done in a REXX program using the SourcePoint function.

Syntax

Result = SourcePoint(Command)

Where:

SourcePoint The REXX function that issues a command to SourcePoint.

Command	A string which represents a valid SourcePoint command (a Command Language command).
Result	The REXX variable that receives any result or output from the SourcePoint command.

Examples

Command inputs:

```

reg_eax = SourcePoint('eax')           // Gets the value in the EAX register
port71 = SourcePoint('port 71h')       // Gets the value from I/O port 71h
vect03 = SourcePoint('dword 0CL')      // Gets the Int03 vector at 0000000C
SourcePoint('eax = A5A5A5A5h')         // Puts a value of A5A5A5A5H in EAX
SourcePoint('ebx = ' tst1')            // Puts the value of variable tst1 into EBX
SourcePoint('step')                    // Single steps the processor
SourcePoint('reset')                   // Resets the target system
SourcePoint('go til 80io')              // Run & stop on any access to I/O port 80h

```

Passing Command Line Parameters

Parameters from the SourcePoint command line that that invoke a REXX program can be easily retrieved using the ARG instruction.

Syntax

ARG VariableList

Where:

ARG The REXX instruction to retrieve parameters passed from the SourcePoint command line.

VariableList Comma-delimited list of variables that receive the passed parameters.

Example

Consider a REXX program that is invoked with the following command line.

```
P0>REXX myprog.rex(0,A5,"Test Interrupts")
```

The following line in a REXX program retrieves the parameter list.

```
ARG Num1,Num2,Str1,Num3
```

After the above line is executed, the variables in the list have the following values.

- Num1 will contain '0'
- Num2 will contain 'A5'
- Str1 will contain 'Test Interrupts'

Num3 will be unchanged from its previous value since the parameter list only contained three items.

Console Output

When a REXX program has been invoked from the SourcePoint command line, console output from the REXX program is directed to the SourcePoint **Command** window.

Example

The example shows a simple REXX program to display the parameter passed to it.

Command input:

ARG parmstring

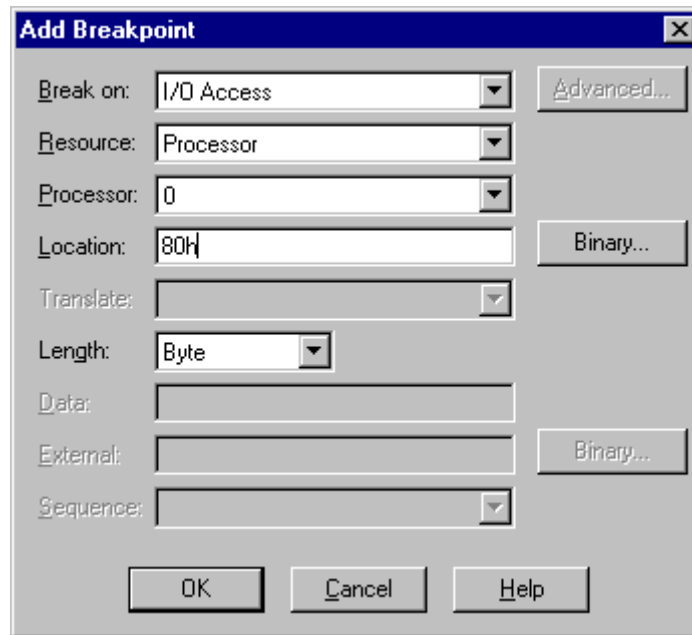
SAY 'The command line parameter was: ' parmstring

How to Use the Wait Command

Oftentimes there are activities to be performed on the target upon a hardware event. The easiest way to perform this task is to set your hardware triggers in the **Breakpoints** window, tie your proc (or command string) to the breakpoint, and wait for the breakpoint to be triggered.

As an example, the following method displays all POST codes (assuming Port 80h is used in the conventional way):

1. Reset the target and verify that it is stopped.
2. Open the **Add Breakpoint** window and set an I/O access breakpoint at Port 80h.



***Breakpoints/Add Breakpoint** dialog box setup for I/O Access breakpoint at 80h.*

3. Execute the following from either the command line, a proc, or macro:

Command input:

```
do {go; wait; regs} while (1)
```

You can decrease granularity by modifying the **do while** construct to stop on a register or memory value, or (for this example) by specifying a value in the **Breakpoints** window that is written to the I/O port.

Confidence Tests Window

Confidence Tests Window Overview

Confidence Tests Window Introduction

Confidence tests are designed to provide confidence that the emulator and target are both working reliably by exercising various fundamental features of the emulator in an automated fashion. There are a number of confidence tests available in SourcePoint. The **Confidence Tests** window can be opened by selecting **Options|Confidence Tests** from the menu bar or the related icon from the toolbar.

Note: Close all windows before running any tests. Open windows may not always be updated and may display incorrect data during and after a confidence test has been run. Most tests modify the state of the target.

Confidence Tests

JTAG | Target Memory

☐ PBD test Status

☐ JTAG ID test Status

☐ JTAG pattern test Status

Pattern:
AA556943

Test setup

☐ Enable logging

☒ Stop on failure

☐ Run continuous

Passes to run:
1

Passes completed:
0

Test status
Warning: close all windows and disable all breakpoints. Most tests modify the target.

Select All Clear All Run Close JTAG Config... Help

Confidence Tests dialog box

Dialog Box Overview

The tests are divided into two categories, as indicated by the tabs in the dialog box – **JTAG** and **Target Memory**. The **Test Setup** section at the right on each tab lets you set various options and tells you how many passes have been completed on any currently running test. The **Status** buttons indicate the status of a test as it runs. These are described in more detail below.

Tests and Test Status Buttons

As each test runs, the text on the associated button changes to show the progress of the testing - **Pass**, **Fail**, **Skipped**, or **Aborted**. At the end of the testing, the buttons indicate test results. Click on the corresponding button to display additional test details. If the test failed, details include the last cause of failure. For detailed information on each test, see "[Confidence Tests Tabs](#)," part of "Confidence Tests Window Overview," found under *Confidence Tests Window*.

Test Setup Section

Enable logging. When this option is enabled, select steps are logged (viewable in the **Log** window). By logging only select steps, and not all of them, tests cycle through passes at a much faster rate than in previous versions of SourcePoint.

Stop on failure. Enabled by default, this option stops the tests after the first failure.

Run continuous. When this option is enabled, the test runs continuously until the user cancels it or until it finds an failure if the **Stop on failure** option is enabled in the **Test setup** section of the dialog box.

The **Passes to run** option defaults to "1." Enter the desired number of trials in the text box. As the test is performed, the iteration number appears in the text box.

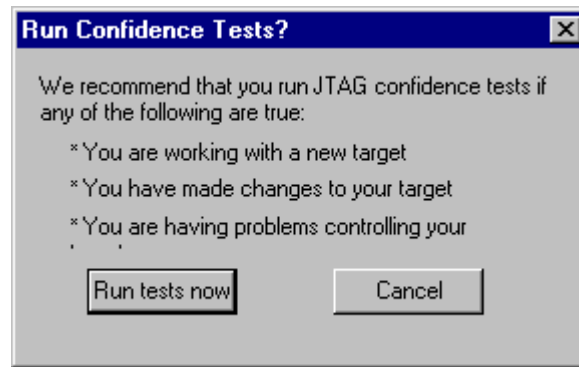
Passes completed. This is not an option but a counter. As the name implies, the text box displays the number of passes the test has completed.

Reset target first. Available only on the **Target Memory** tab, **Reset target first** allows you to reset the target before running the target memory tests.

Note: **Run continuous** is the alternative to **Passes to run**, not to **Stop on failure**. If you enable the **Run continuous** option, the **Passes to run** option is grayed out, and vice versa. If you select **Run continuous** and **Stop on failure**, the emulator still stops on the first failure. In other words, either choose the number of passes you want to run or enable the **Run continuous** option.

Pop-Up Dialog Box

Run Confidence Tests? dialog box displays when you add or change a connection (e.g., from TCP/IP to USB) or when you change settings on the **Emulator Configuration JTAG** or **JTAG Clock** tabs. Arium encourages you to run the JTAG confidence tests at those times.



Run Confidence Tests? pop-up dialog box

Confidence Tests Tabs

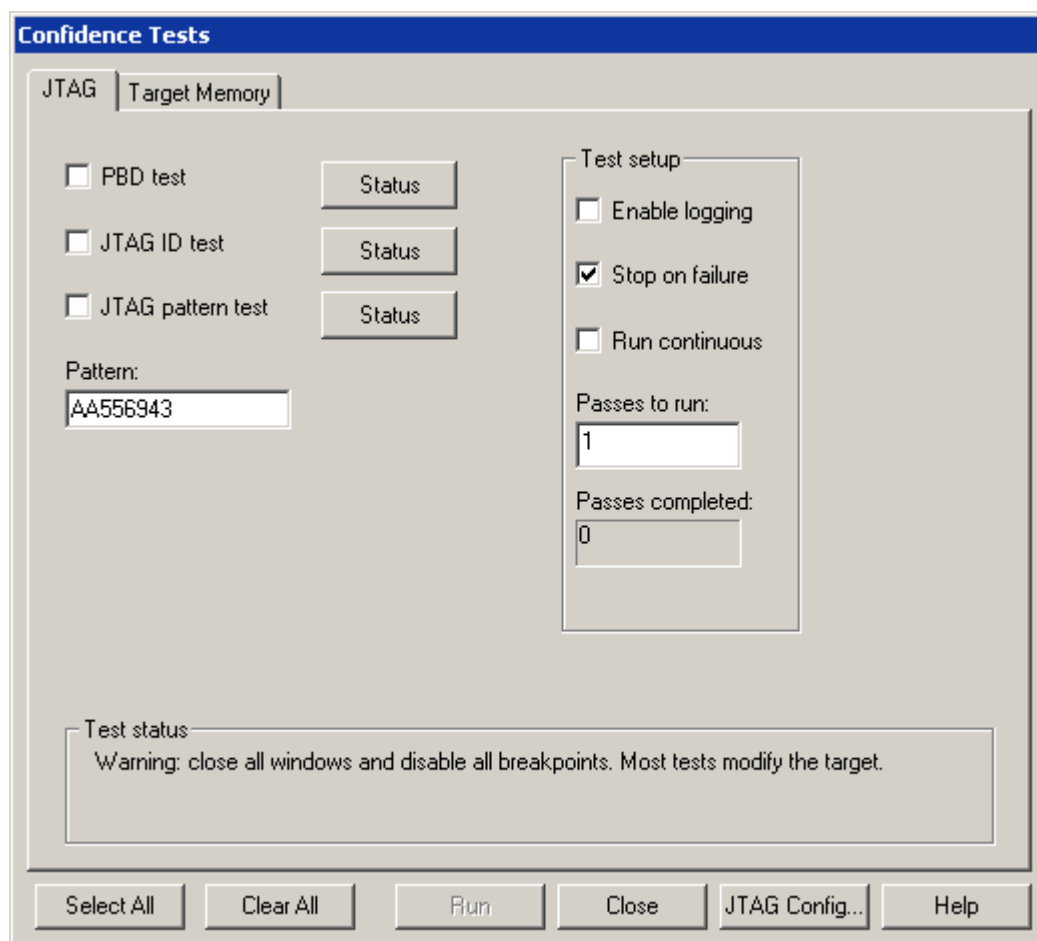
JTAG Tab

PBD test. Run this test to determine whether the PBD is operating properly.

JTAG ID test. This test causes all the JTAG IDs to be read from the JTAG scan chain of the target. When the test ends, you can open a report that lists the IDs that were received.

JTAG pattern test. This test shifts a known pattern through the data register (DR), reads it back, and uses the return value to calculate the chain size. This is cross-checked against the results of the JTAG ID test. This is a good test for stressing the JTAG circuitry to be sure that it is working reliably.

Pattern. This pattern can be any 32-bit hexadecimal pattern. Choose a pattern and define it in the text box.



Confidence Tests dialog box showing the JTAG tests

Target Memory Tab

Read target memory. This reads target memory from a given start address to a given end address. The data that are read are not checked for validity. This test can be used to uncover JTAG-related memory read problems.

Write target memory. This test first writes, then reads target memory from a given start address to a given end address. The read data are checked for validity. To determine the nature of a problem, open a **Memory** window and view the results.

Start address/End address. Determine the range of memory you want to test. Place the start and end addresses in the appropriate text boxes.

Write Data Pattern section. This section offers the data pattern options:

- **Address as Data** is most useful for exposing problems with memory address lines.
- **Checkerboard** is useful for exposing problems with memory data signals. (The Checkerboard pattern looks like: 55555555 AAAAAAAAAA 55555555 AAAAAAAAAA.)
- **Fill With:** allows you to set a data pattern. For this last option, fill in the text box with a data value. That value is then written to every memory location within the selected address range.

The screenshot shows the 'Confidence Tests' dialog box with the 'Target Memory' tab selected. The dialog is divided into several sections:

- Test Selection:** Two checkboxes are present: 'Read target memory' and 'Write target memory'. Each has a corresponding 'Status' button to its right.
- Address Range:** Two text input fields labeled 'Start address:' and 'End address:'.
- Write data pattern:** A group box containing three radio buttons: 'Address as data' (selected), 'Checkerboard', and 'Fill with:'. The 'Fill with:' option has an associated text input field.
- Test setup:** A group box containing:
 - Checkboxes for 'Enable logging' and 'Run continuous'.
 - A checked checkbox for 'Stop on failure'.
 - A 'Passes to run:' label followed by a text input field containing the value '1'.
 - A 'Passes completed:' label followed by a text input field containing the value '0'.
 - A checkbox for 'Reset target first'.
- Test status:** A text area at the bottom containing the warning: 'Warning: close all windows and disable all breakpoints. Most tests modify the target.'
- Buttons:** A row of buttons at the bottom: 'Select All', 'Clear All', 'Run' (highlighted), 'Close', 'JTAG Config...', and 'Help'.

Confidence Tests dialog box showing **Target Memory** tests

Table of Confidence Test Failures and Symptoms

Test Name	Common Failures
PBD test	Wrong PBD. PBD jumpered incorrectly.
JTAG ID test JTAG pattern test	Poor connection from emulator to target. JTAG clock rate too high. Wrong JTAG current level. Electrical problems with JTAG circuitry on the target.
Read target memory Write target memory	Illegal address range given, or problems with target memory. Target DRAM controller or chipset not initialized. Processor not stopped, and test could not stop it.

Descriptors Tables Window

Descriptors Tables Window Overview

Descriptors Window Introduction

The **Descriptors** windows are used to examine and modify descriptor table entries. The elements of a **Descriptors** window include the title bar, data area, and option tabs for selecting the descriptor table type. The **Descriptors** window can be opened by selecting **View|Descriptors** on the menu bar or by clicking on the **Descriptors** icon on the toolbar. The **GDT Descriptors** table opens automatically.

GDT Descriptors window

Window Structure

Offset Column

The **Offset** column lists the value of a selector index field within a segment register that points to a descriptor. A selector index is the decimal entry number multiplied by 8 and displayed as a hexadecimal value.

Type Column

The **Type** column lists the descriptor type. Code and data descriptor types include abbreviations that define the set/not-set state of their status bits.

- Code descriptor types are listed with abbreviations for **Conforming** (C), **Readable** (R), and **Accessed** (A) status bits. An exclamation mark (!) precedes an abbreviation if the bit is cleared (e.g., !A=Segment has not been accessed).
- Data descriptor types are listed with abbreviations for **Expand-down** (E), **Writable** (W), and **Accessed** (A) status bits. An exclamation mark (!) precedes an abbreviation if the bit is cleared (e.g., !E=Expand-up segment).
- Task State Segment (TSS) descriptor types are listed as 16- and 32-bit TSS and with the word **Busy** when the TSS is not available.
- Gate descriptor types are listed as 16- or 32-bit call-gates, 16 or 32-bit interrupt-gates, task-gates, and 16- or 32-bit trap-gates.

Note: Status is not defined for LDT, task-gate, call-gate, and IDT descriptor types. TSS types may include a **Busy** status.

Attributes Column

The **Attributes** column defines the Descriptor Privilege Level (DPL) and the **Present** (P) bit for all descriptors. Other attributes, defined for certain descriptor types, are identified below.

- The Granularity (G) bit is defined for code, data, TSS, and LDT descriptors (i.e., segment limit is G=page granular or !G=byte granular).

- The operand/address-mode default size is defined for code (D) and data (B) descriptors (i.e., operand/address-mode is D/B=32-bit or !D/B= 16-bit).
- The Available (Avl) bit is defined for code, data, and TSS descriptors (i.e., segment is Avl=available or !Avl=not available).
- The Dword (Dword) count is defined for call-gate descriptors.

Values Column

The **Values** column lists the base address and limit for code, data, TSS, and LDT (table) descriptors, or the selector and offset for call, interrupt, and trap-gate descriptors; it lists the selector (only) for task-gate descriptors.

- **Base** defines the location of a segment within the 4 Gbytes of physical address space. A base address is displayed as an 8-digit hexadecimal value.
- **Limit** is a 20-bit value representing the size of the memory segment. A limit is displayed as a 5-digit hexadecimal value.
- Interrupt and trap gate descriptors use a 16-bit selector and a 32-bit offset as destination fields that point to the start of an interrupt or trap routine. A selector is displayed as a 4-digit hexadecimal value; offsets are displayed as 8-digit hexadecimal values.

Note: Task gate descriptors use only the selector field to refer to a TSS.

Tabs

The tabs at the bottom of the window are used to select the descriptor tables to display in the window. The title bar changes to reflect the selected descriptor table and shows the base and limit values of that table.

- Select the **GDT** tab to view the status of or modify the contents of a Global Descriptor Table (GDT) entry.
- Select the **IDT** tab to view the status of or modify the contents of an Interrupt Descriptor Table (IDT) entry.
- Select the **LDT** tab to view the status of or modify the contents of a Local Descriptor Table (LDT) entry.
- Select the **LDTR** tab to view the status of or modify the contents of a Local Descriptor Table Register (LDTR) entry.

The **LDT** or **LDTR** option tabs may not be enabled in all situations. The **LDTR** tab displays the currently active local descriptor table based upon the LDTR register value. The **LDT** tab is used to display any local descriptor table that is referenced in the **GDT**. The **LDT** tab works only when an **LDT** entry is selected in the **GDT** display.

Descriptors Window Menu

The Descriptor Table menu, a context menu accessed by right-clicking on a table entry, features an easy way to segue into either a **Code** window or **Memory** window based on the highlighted descriptor.

Descriptors menu

View as Code menu item. This selection opens a **Code** window at the address of the selected descriptor table entry. Code can then be viewed and breakpoints can be set through the open **Code** window.

View as Memory menu item. A **Memory** window opens at the selected descriptor table entry. Memory can then be examined or changed.

Properties menu item. Clicking on the **Properties** menu item causes a **Descriptors** dialog box to display. The information in each dialog box varies, depending on the type of descriptor, but there are three sections in each: **Descriptor type**, **Attributes**, and **Values**. They are described in more detail below.

Note: The information in these columns can be edited to modify existing descriptors or to add new descriptors.

Sample Properties dialog box

- **Descriptor Type Section.** The **Descriptor Type** section displays the descriptor type selected from the **Descriptor Type** drop down list box. The displayed descriptor type is one of several included in a drop down list. For application descriptor types (i.e., **Code** and **Data** descriptors), this dialog box includes check boxes whose default states define the descriptor status. With exception of the TSS **Busy** bit, no status is defined for system descriptor types (i.e., the LDT, TSS, and gate-type descriptors).

Note: Enabling or disabling these options changes the values displayed for Bytes 7-4 in the **Values** section of dialog box.

- **Attributes Section.** The **Attributes** section contains check boxes where default enabled/disabled states define the attributes of each descriptor. This area also contains a text box that displays the default Descriptor Privilege Level and, for call-gate descriptors, a second text box that displays the default Dword count.

Note: Enabling or disabling the attributes in the **Attributes** section of the dialog box changes the values displayed for **Bytes 7-4** in the **Values** section.

- **Values Section.** The **Values** section contains text boxes that are labeled **Base** and **Limit** or **Selector** and **Offset**. These text boxes display the base address and limit values for code, data, TSS, and LDT descriptors and the selector and offset values for call, interrupt, and trap-gate descriptors. Only the selector is displayed for task-gate descriptors.

The **Values** section also contains text boxes labeled **Bytes 7-4** and **Bytes 3-0**. These text boxes display the values defined for each field in a descriptor's 8-byte data structure. These fields, in addition to defining a descriptor's base address and limit or selector and offset, define its type, status, and attributes.

How To - Descriptors

How to Replace a Descriptor Entry

Replacing a descriptor entry is the process of selecting a descriptor, modifying its status and attributes, and replacing the selected descriptor with the modified version.

To Replace a Descriptor Entry

1. In the descriptor table, double-click on the entry you want to modify.

The **Properties** menu item displays.

2. Click on **Properties**.

A **Descriptor** dialog box displays.

3. From the **Descriptor Type** drop down list, select the descriptor to be replaced with a modified version.

Note: The descriptor's type and status appear in the **Descriptor Type** dialog box; corresponding attributes appear in the **Attributes** section of the dialog box. The descriptor's **Base address and Limit** or **Selector and Offset** and the values defined for **Bytes 7-4** and **Bytes 3-0** appear in the **Values** section of the dialog box.

4. In the **Descriptor Type** dialog box, select the check boxes that provide the desired status.
5. In the **Attributes** section of the dialog box, type in the desired **Descriptor privilege level**.
6. Select the check boxes that provide the desired attributes.
7. In the **Values** section of the dialog box, enter the desired values for **Base** and **Limit** or **Selector** and **Offset**.
8. Click the **Apply** button.
9. Click the **OK** button.

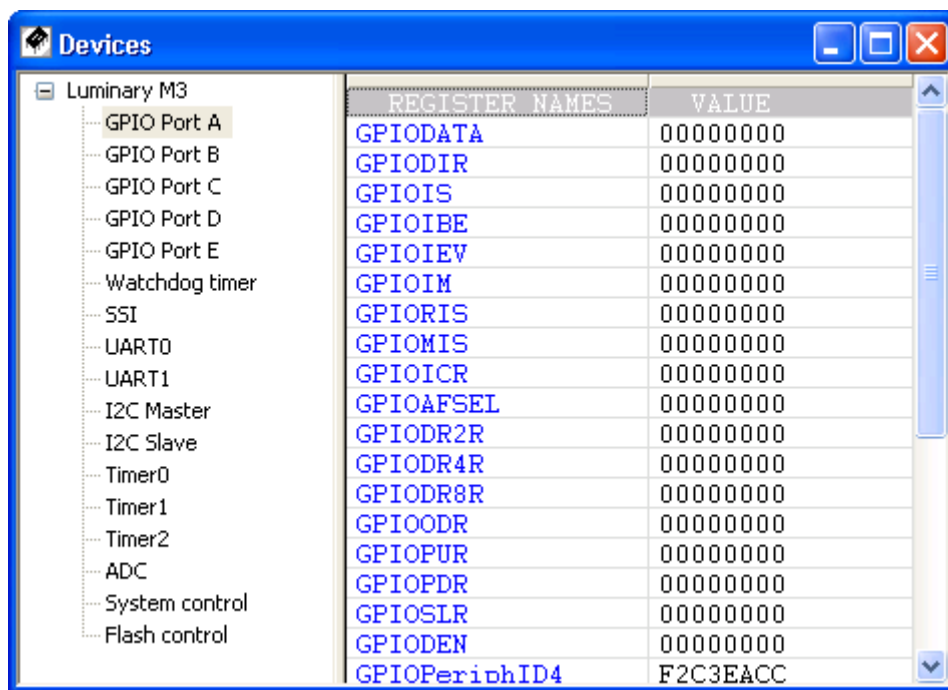
Devices Window

Devices Window Overview

Devices Window Introduction

To open the **Devices** window, select **View|Devices**. If it is your first time opening the **Devices** window, you are prompted to select a device file.

The left-hand pane of the **Devices** window is called the Devices pane. This is a simple tree structure of devices. The right hand pane is referred to as the Grid pane. Once you have selected a device from the Devices pane, the corresponding predefined cells are displayed in the Grid pane.



Devices window

Device View Files - Overview

The **Devices** window allows you to define a custom view of memory. A common use of this view is to display the memory mapped I/O of the devices within a system. The format of this view is defined by one or more text files called device view files. The extension for these files is ".dev". Each file contains definitions for one or more devices. Each device contains a number of cell definitions.

Arium provides device view files for many common processors. These are located in Targets\Device_view under the directory where SourcePoint was installed. For other processors, a text editor can be used to create your own device view files. The Arium-provided files provide examples of how to create these files.

Device View File Structure

Device view files are simple text files. White space is ignored. Keywords are case-insensitive. Standard C++ comments (//) are allowed.

Each Device view file contains one or more device definitions. The syntax for a device definition is as follows:

```
[Device#]  
<device directives>  
<enumerations>  
<cell definitions>
```

The Device# entry specifies the device number. Device numbering begins with 0 and must be numbered consecutively in the file.

Cell Definitions

The general syntax for a cell definition is as follows:

Cell# = <row#>, <column#>, cell-type, options

Where:

cell-type = {TEXT | REG | MEM | IO | USER | CHILD}
options = see Device/Cell Options below

Row and column numbers are 0-based.

Text cells

Text cells allow you to display a label in a cell.

Syntax:

CELL#=<row#>, <column#>, TEXT, <text enclosed in quotes>

Example:

CELL0=1, 1, TEXT, "Hello world!"

The above example creates a text cell in the second column of the second row and inserts the phrase "Hello world!" into it. The maximum text length is 100 characters.

Memory cells

Memory cells allow you to display the contents of a memory location in a cell. The memory location is limited to lengths of 1, 2, 4, or 8 bytes. The memory location is re-read every time the target stops.

Syntax:

CELL#=<row#>, <column#>, MEM,<address>, <length in bytes>

Example:

```
CELL0=0, 0, MEM, 1000p,1
```

The above example creates a memory cell in the first column of the first row and displays 1 byte of memory starting at physical address 1000.

Note: When SourcePoint reads memory, it reads 128 byte blocks to speed up display. On some systems this may be a problem (e.g., reading the area between memory-mapped I/O). If this is the case, create a memory map entry encompassing the memory cells (**Options|Target Configuration**) and set the type to I/O. This forces SourcePoint to read each cell separately, and only read the exact number of bytes defined in the cell.

Register cells

Register cells allow you to display and change the contents of a register. The register value is re-read every time the target stops.

Syntax:

```
CELL#=<row#>, <column#>, REG, <register name>
```

Example:

```
CELL0=0, 1, REG, CPSR
```

The above example creates a register cell in the second column of the first row and displays the value of the CPSR register in the cell.

MSR cells (IA-32 processors only)

MSR cells allow you to display and change the contents of an MSR.

Syntax:

```
CELL#=<row#>, <column#>, MSR, <msr address>
```

Example:

```
CELL0=0, 0, MSR, 80
```

The above example creates an MSR cell in the first column of the first row and displays the value read from MSR 80H. The MSR value is re-read every time the target stops.

I/O cells (IA-32 processors only)

I/O cells allow you to display the contents of an I/O port in a cell.

Syntax:

```
CELL#=<row#>, <column#>, IO, <port address>, <port size>
```

Example:

```
CELL0=0, 0, IO, 80, 1
```

The above example creates an I/O cell in the first column of the first row and displays an 8-bit value read from I/O port 80H. The port value is re-read every time the target stops.

User-defined cells

User-defined cells allow you to enter an expression to be evaluated every time the target stops. Any expression that can be evaluated in the Command window can be specified.

Syntax:

CELL#=<row#>, <column#>, USER, expression

Example:

CELL0=0, 0, USER, li + uli

The above example creates a user-defined cell in the first column of the first row and displays the sum of program symbols li and uli. The expression is re-evaluated every time the target stops.

Child cells

Child cells display a portion of another register, memory or MSR cell (within the same device).

Syntax:

CELL#=<row#>, <column#>, CHILD, <parent row>, <parent column>, <offset>, <length>, <name>

Example:

CELL0=0, 1, REG, CPSR
CELL1=5, 1, CHILD, 0, 1, 5, 1, T

The above example creates two cells. First it creates a register cell and places the value of CPSR into that cell. The second cell's definition creates a child of the first cell, displaying Bit 5 of CPSR (the T bit).

Child values are automatically shown in the tooltip help of parent cells. See below for more information. If you specify a negative row or column number for a child cell, then a cell will not be created. This is useful when the only place you want to see the child value is in the tooltip of the parent.

Device Directives

Device directives specify different attributes of a device.

Name directive

Specifies the name for a device (the name that appears in the Devices pane). This directive is required.

Syntax:

Name = <name>

Example:

[Device0]
Name = Uart

Base directive

Specifies a base address for all memory cells within a device definition. Once defined, memory cells can specify addresses relative to this base address. See the **RepeatDevice** directive below for an example of where this might be useful.

Syntax:

Base = <address>

Example:

```
[Device0]
Name = Uart
Base = 3FFF0000
Cell0 = 0, 0, MEM, base+1CF8, 4
```

In this example a memory cell is defined at address 3FFF1C8 (3FFF000 + 1CF8).

Processor Directive

In a multi-processor target, specifies which processor to use when reading memory and registers. If this directive is not specified, then the current viewpoint processor active when the Device view file was loaded is used.

Syntax:

Processor = #

Example:

```
[Device0]
Name = Uart
Processor = 1
```

RepeatDevice directive

This directive allows creation of a device that has a definition identical to a previously defined device. This is useful when a system has two identical devices (e.g., two uarts), where the cell definitions are identical, and only the addresses differ.

Syntax:

RepeatDevice = <device#>

Example:

```
[Device1]
Name = Uart1
Base = E0000000
< cell definitions>

[Device2]
Name = Uart2
RepeatDevice = 1      // use cell definitions from device 1
Base = E1000000
```

In this example two devices are defined. The second is identical to the first with the exception of the name and the base address used when reading memory.

Note: When using **RepeatDevice**, cell names will not be unique, which limits the usefulness of the **AddSymbols** directive.

AddSymbols directive

This directive adds the names of memory-mapped I/O to the command language. If this directive is not specified, then the names are not added to the command language.

Memory-mapped I/O displayed in the **Devices** window consists of pairs of cells, a text cell displaying a name, and a memory cell displaying the contents of memory.

Note: TEXT cells should not contain spaces in the text string.

This directive may appear in the [Group] section at the top of a Device view file, in which case it applies to all devices in the file, or it can appear within a [Device] section, in which case it applies to only that device.

Note: This directive increases the load time of Device view files.

Syntax:

AddSymbols = [true | false]

Example:

```
[Device1]
Name = Uart1
AddSymbols = true
Cell0 = 0, 0, text, "ctrlReg"
Cell1 = 0, 1, mem, 1000, 4
```

In this example, the symbol ctrlReg is added to the command language and is equal to address 1000. Typing "ord4 ctrlReg" will read memory at address 1000.

Enumerations

Enumerations can be defined in a device file and be used by the cells to display a readable string value rather than a raw number. An enumeration is defined by an [ENUM#] section, where # is an integer number. Each entry in the enumeration must be sequentially numbered starting with 0. Enumerations apply to all devices within a file.

Syntax:

```
[Enum#]
Name=<name>
Key#=<value>, <string>
```

Example:

```
[Enum0]
name="Level"
key0=0, "Low"
key1=1, "High"

[Enum1]
name="version"
key0=0, "Version 1.0 Rom"
```

```
key1=1, "Version 1.1 Rom"
key2=2, "Version 1.2 Rom"
```

In this example two enumerations (level and version) are defined.

Example:

```
Cell0=1, 1, MEM, 1000, 1, enum=version
```

The above example causes a version string to be displayed in a cell rather than a number.

Groups

Groups allow you to group devices together (in the Devices pane) to help organization. A group is defined by adding the section [GROUP] in the device file. The group must also be given a name. After adding this to your device file, all devices in that file are grouped in the Devices pane under the given group name. Multiple groups require multiple device view files, one per group.

Example:

```
[Group]
Name="UARTS"
```

Device/Cell Options

The following directives are dual purpose. They can be inserted in a device section to affect all cells within a device definition, or they can be added at the end of a specific cell definition to affect just that cell.

Background Color

Background color can be set using an RGB value or using a keyword. The available color keywords are as follows: black, white, red, green, blue, yellow, orange, gray, magenta. If not specified, the background color specified in **Options|Preferences|Colors** is used.

Syntax:

```
BACK=<Hex Value for Red><Hex Value for Green><Hex Value for Blue>
```

or

```
BACK=<Color Keyword>
```

Example:

```
BACK=00FF00 // set background color of all cells to green
```

Example:

```
BACK=green // set background color of all cells to green
```

Example:

```
Cell0=0, 0, text, "Name", back=white // set background color of this cell only to white
```

Text Color

Text color can be set using an RGB value or using a keyword. The available color keywords are as follows: black, white, red, green, blue, yellow, orange, gray, magenta. If not specified, the text color specified in **Options|Preferences|Colors** is used.

Syntax:

TEXT=<Hex Value for Red><Hex Value for Green><Hex Value for Blue>

or

TEXT=<Color Keyword>

Example:

TEXT=FFFFFF // set text color of all cells to white

Example:

TEXT=white // set text color of all cells to white

Example:

Cell0=0,0,text,"Name",text=white // set text color of this cell only

Justification

Cell justification can be set using three keywords: left, center, or right. The default is left justification.

Syntax:

JUSTIFY=<Justification Keyword>

Example:

JUSTIFY=center // set justification for all cells in a device

Example:

Cell0=0, 0, justify, "Name", justify=center // set justification of this cell only

Tooltip

Tooltip text can be defined for each cell. When the mouse pointer hovers over a particular cell the text will be displayed.

Syntax:

Tooltip=<Tooltip text>

Example:

Tooltip="This is a sample of tooltip text."

Note: If tooltip text is not defined for a register cell and the register has subfields, these subfields are shown automatically in the tooltip text (similar to what happens in the Registers window). If tooltip text is not defined for a memory cell and there are child cells pointing to the memory cell, the child values are shown automatically.

Accessibility

Cell accessibility indicates whether the cell is read only, write only, or read/write. The default is read/write.

Syntax:

ACCESS=<R, W, RW>

Example:

```
ACCESS=R                                // set accessibility for all cells to read-only
```

Note: This attribute applies only to memory cells.

Format

The display format of a cell can be altered using a PRINTF format specification string. This allows you to display a value in a base other than hex.

Syntax:

FORMAT=<PRINTF format specification string>

Example:

```
FORMAT="%d"                             // set display format for all cells in device
```

Example:

```
cell0=0, MEM, 100p, I, FORMAT="\"%d\"    // set display format for a single cell
```

Enumeration

A cell can be tied to an enumeration that has been defined elsewhere in the device file. This can be used to display multi-bit fields as meaningful text strings rather than raw hex values. To do this, simply name the enumeration. (For more information, see enumerations above.)

Syntax:

ENUM=<Enumeration name>

Example:

```
ENUM="StatusBits"
```

Combined Examples

The following examples illustrate using multiple options.

Example:

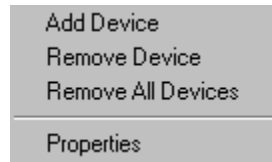
```
cell0=0, 0, mem, 1000p, 4, bkgnd=black, text=FFFFFF, justify=center, tooltip="Memory Mapped Device"
```

Example:

```
cell1=1, 0, mem, 7FFFC3B0p, 4, access=r, enum="Status"
```

Devices Window Menu

Devices Pane



Devices window menu - Devices pane

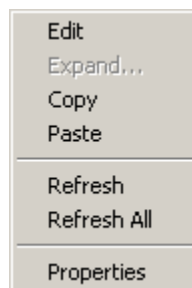
Add Device Menu Item. Prompts for a device view file to load. More than one file can be loaded at a time. Each file can contain one or more device definitions. Currently displayed devices are saved in the project file, so when you exit and restart SourcePoint the devices displayed are remembered.

Remove Device Menu Item. Removes a device from the display. This action causes the Device view file containing the device to be unloaded. If more than one device was defined in the file, then multiple devices are removed.

Remove All Devices Menu Item. Removes all devices from the display. This action unloads all Device view files.

Properties Menu Item. Shows the Device view file associated with a device.

Grid Pane



Devices window menu - Grid pane

Edit Menu Item. Edits the value of a register or memory cell. Memory cells must have write access enabled to be editable.

Expand Menu Item. Expands a register cell into binary. This view also displays any bit fields (flags) defined within the register.

Copy Menu Item. Copies the contents of a cell to the clipboard.

Paste Menu Item. Pastes a register or memory value into a cell. This applies to non-TEXT cells.

Refresh Menu Item. Refreshes a register or memory cell. This menu item also forces a re-read of the value from the target.

Refresh All Menu Item. Refreshes all cells within a device.

Properties Menu Item. Displays the properties (attributes) of a cell. The list of properties varies by cell type.

Accessing Devices Window Cells in the Command Window

Often times the **Devices** window is used to display memory-mapped I/O locations containing the register definitions for the internal peripherals inside a device. Column 0 typically contains text cells with register names and column 1 contains memory cells with register values. (See C:\Program Files\American Arium\ARM\Samples\Device for examples of these kinds of device files.)

SourcePoint can optionally make these register names available in the command language. This allows you to access individual registers via the **Command** window or manipulate registers within a command script (include file).

To enable this feature for all devices in a device file, add the following entry to the file:

```
[Group]
AddSymbols=1
```

Device files often contain definitions for a group of devices. To enable this feature for a single device (e.g., Device3), add the following entry to that devices section:

```
[Device3]
AddSymbols=1
```

Note: To speed up the processing of device files, the cells for a particular device are processed only when that device is displayed in the **Devices** window. Devices whose memory-mapped registers are to be added to the command language must be processed when the device file is loaded, so device file load times may rise.

Note: Memory cells are only added to the command language if the preceding cell (same row, previous column) is a text cell.

Example:

The following two cell definitions are part of the definition of the memory-mapped registers used to configure a DMA controller:

```
[Device0]
Name=DMA Controller
AddSymbols=1

cell2=1,0,text,"DCSR0",text=blue,tooltip="DMA Control/Status Register"
cell3=1,1,mem,0x40000000,4,access=rw,
```

Adding the "AddSymbols=1" entry will result in the following alias being defined in the command language:

```
#define DCSR0 0x40000000
```

It thus types the following in the **Command** window:

```
word DCSR0
```

This causes the command interpreter to replace DCSR0 with 0x4000000 and display 4 bytes of memory read from address 0x40000000.

How To - Devices Window

How to Create Simple Devices Windows

The **Devices** window is a versatile user content-defined window to display memory mapped I/O devices.

Creating a Devices File

The file used for the **Devices** window is an ascii text file with particular formatting described in detail in the **Devices** window introduction. This file can be created using your favorite text editor.

1. Open your text editor.
2. The first active lines in your devices file should begin with the keyword [Group]. This will be the primary text displayed in the **Devices** window. We will use the Altera Excalibur SOPC as our example.

Input:

[Group]

Name=Altera Excalibur EPXA1/4/10 Device Registers

3. The next portion of your device file describes groupings of particular devices. You can have multiple devices groupings using the syntax Device0, Device1,...

Input:

[Device0]

Name=Reset and Mode Control Regs

4. Type the following to create the **Device** window shown below:

Note: Each device is composed of cell entries built in a row/column grid fashion. Cells MUST be numbered in sequential fashion. You can create your cell contents with text, memory data, or register values. Flyover tooltips can also be included for each cell. Spaces are not allowed between fields. Comment lines are delineated with a // at the beginning of the line.

```
// Column Headers
//format is shown as:
// Cell#, row, column, type is text, string, text color (background and foreground), tooltip
text
//
cell0=0,0,text," REGISTER NAMES ",back=gray,text=white
cell1=0,1,text," VALUE ",back=gray,text=white,tooltip= "BASE= 0x7FFFC000"
// Register Names
//format is shown as:
// Cell#, row, column, type is text, string, text color, tooltip text
//
cell2 =1,0,text,"BOOT_CR",text=blue,tooltip="boot control register"
cell3 =2,0,text,"RESET_SR",text=blue,tooltip="reset status register"
cell4 =3,0,text,"IDCODE",text=blue,tooltip="identity and version register"
cell5 =4,0,text,"SRAM0_SR",text=blue,tooltip="single-port SRAM0 size register"
cell6 =5,0,text,"SRAM1_SR",text=blue,tooltip="single-port SRAM1 size register"
cell7 =6,0,text,"DPSRAM0_SR",text=blue,tooltip="dual-port SRAM0 size register"
cell8 =7,0,text,"DPSRAM0_LCR",text=blue,tooltip="dual-port SRAM0 lock control register"
```

```

cell9 =8,0,text,"DPSRAM1_SR",text=blue,tooltip="dual-port SRAM1 size register"
cell10=9,0,text,"DPSRAM1_LCR",text=blue,tooltip="dual-port SRAM1 lock control register"
// Memory Mapped locations for registers listed above
//format is shown as:
// Cell#, row, column, type is memory, memory location, access limits, tooltip text
//
cell11=1,1,mem,0x7ffc000,4,access=rw,tooltip="base + 000H R/C BUS 2"
cell12=2,1,mem,0x7ffc004,4,access=rw,tooltip="base + 004H R/C BUS 2"
cell13=3,1,mem,0x7ffc008,4,access=r,tooltip="base + 008H R BUS 2"
cell14=4,1,mem,0x7ffc020,4,access=r,tooltip="base + 020H R BUS 2"
cell15=5,1,mem,0x7ffc024,4,access=r,tooltip="base + 024H R BUS 2"
cell16=6,1,mem,0x7ffc030,4,access=r,tooltip="base + 030H R BUS 2"
cell17=7,1,mem,0x7ffc034,4,access=rw,tooltip="base + 034H R/W BUS 2"
cell18=8,1,mem,0x7ffc038,4,access=r,tooltip="base + 038H R BUS 2"
cell19=9,1,mem,0x7ffc03C,4,access=rw,tooltip="base + 03CH R/W BUS 2"

```

5. Save the file with the extension ".dev". Your complete file should look like this:

Result:

[Group]

Name=Altera Excalibur EPXA1/4/10 Device Registers

[Device0]

Name=Reset and Mode Control Regs

// Column Headers

cell0=0,0,text," REGISTER NAMES ",back=gray,text=white

cell1=0,1,text," VALUE ",back=gray,text=white,tooltip="BASE= 0x7FFFC000"

// Register Names

cell2 =1,0,text,"BOOT_CR",text=blue,tooltip="boot control register"

cell3 =2,0,text,"RESET_SR",text=blue,tooltip="reset status register"

cell4 =3,0,text,"IDCODE",text=blue,tooltip="identity and version register"

cell5 =4,0,text,"SRAM0_SR",text=blue,tooltip="single-port SRAM0 size register"

cell6 =5,0,text,"SRAM1_SR",text=blue,tooltip="single-port SRAM1 size register"

cell7 =6,0,text,"DPSRAM0_SR",text=blue,tooltip="dual-port SRAM0 size register"

cell8 =7,0,text,"DPSRAM0_LCR",text=blue,tooltip="dual-port SRAM0 lock control register"

cell9 =8,0,text,"DPSRAM1_SR",text=blue,tooltip="dual-port SRAM1 size register"

cell10=9,0,text,"DPSRAM1_LCR",text=blue,tooltip="dual-port SRAM1 lock control register"

// Memory Mapped locations

cell11=1,1,mem,0x7ffc000,4,access=rw,tooltip="base + 000H R/C BUS 2"

cell12=2,1,mem,0x7ffc004,4,access=rw,tooltip="base + 004H R/C BUS 2"

cell13=3,1,mem,0x7ffc008,4,access=r,tooltip="base + 008H R BUS 2"

cell14=4,1,mem,0x7ffc020,4,access=r,tooltip="base + 020H R BUS 2"

cell15=5,1,mem,0x7ffc024,4,access=r,tooltip="base + 024H R BUS 2"

cell16=6,1,mem,0x7ffc030,4,access=r,tooltip="base + 030H R BUS 2"

cell17=7,1,mem,0x7ffc034,4,access=rw,tooltip="base + 034H R/W BUS 2"

cell18=8,1,mem,0x7ffc038,4,access=r,tooltip="base + 038H R BUS 2"

cell19=9,1,mem,0x7ffc03C,4,access=rw,tooltip="base + 03CH R/W BUS 2"

Loading the Devices File

1. Select the **Devices** window icon, or from the **View** menu select **Devices**.

At the first initiation, a file dialog displays asking for the file.

2. Select your device.dev file and click on **Open**.

Note: You can add additional views by right clicking in the devices side (left) of the **Devices** window pane.

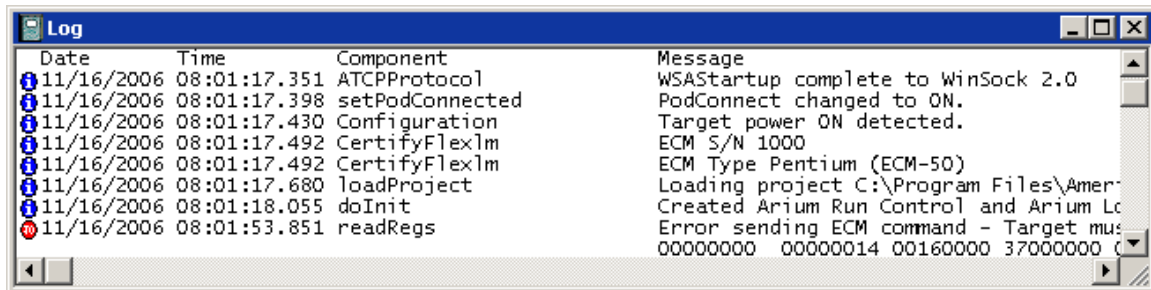
Sample device files may be found in the /Samples/Devices directory of your SourcePoint installation.

Log Window

Log Window Overview

Log Window Introduction

Select **View|Log** on the menu bar to open a **Log** window that displays SourcePoint event information. Each event is associated with the time that it occurred, the system component that recorded the event, and the event itself.



Log window

The **Log** window provides a convenient display of information that is contained in the log file to which SourcePoint continually writes. SourcePoint writes to a file named SPLOG00.txt in the default directory. This file is stored in clear text and can be read directly by any text editor.

The window provides a display area for warnings and errors that occur during the operation of SourcePoint. Not all errors are logged in this window. The primary purpose is to log warnings and errors for diagnostic purposes. The information contained in the messages is designed to aid the Arium technical support staff in troubleshooting customer difficulties.

The columns can display the date and time at which the error/warning occurred, the type of message logged, and several columns about the software components of SourcePoint that originated the message. The **Log** window may be ignored in most situations; the Arium technical supports staff may ask for the contents of this window to assist them in solving a particular problem with SourcePoint.

The **Log** window fully supports **Copy**, **Print**, **Print Preview**, and **Save** functions. For more information, please refer to "File Menu" in "SourcePoint 4.0 Overview" under **SourcePoint Environment**.

Log Display Columns

The **Log** window consists of columns that are labeled in the display. The **Log** window displays columns from left to right are: **Type**, **Date**, **Time**, **File [Line]**, **Component**, and **Message**. Any entry may be displayed over multiple lines. If an entry spans multiple lines, only the message column will display on subsequent lines.

Display of some columns is optional. For more information on which columns can be enabled/disabled and how to enable/disable them, see "[Log Window Menu](#)," part of "Log Window Overview," found under *Log Window*.

Type Column

Entries in the log are provided via icons and classified by type.

Information. These entries are purely informational in content. Examples of the entries of this type include log start, log end, initialization, and target acquisition.

Warning. These entries contain information about exceptional conditions that were handled successfully.

Error. These entries are errors that were not successfully handled. The system may recover, but an error usually indicates that either a request was left unsatisfied or a response was incomplete. Data may be corrupted.

Fatal. These entries are probably the last entries before SourcePoint crashes.

For more information on the icons, see, "[Log Window Icon Definitions](#)," part of "Log Window Overview," found under *Log Window*.

Date/Time Column

The **Date/Time** column contains the date and time that the entry was made in the log. This column is colored blue. Display of this column is optional.

File [Line] Column

The **File [Line]** column contains an abbreviated display of the name of the SourcePoint source file followed by the line number in the source file where the entry originated. This column is colored gray. Display of this column is optional and disabled by default.

Component Column





The **Component** column contains the logical part of SourcePoint that generated the event. This column is colored green. Display of this column is optional.

Message Column

The Message column contains the bulk of the event message. This column is colored black.

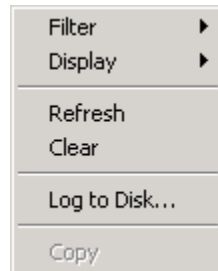
Log Window Icon Definitions

Entries in the **Log** window are classified by type. The types are as follows:

	Information symbol. These entries are purely informational in content. Examples of the entries of this type include log start, log end, initialization, and target acquisition.
	Warning symbol. These entries contain information about exceptional conditions that were successfully handled.
	Error symbol. These entries are errors that were not successfully handled. The system may recover, but an error usually indicates that either a request was left unsatisfied or a response was incomplete. Data may be corrupted.
	Fatal symbol. These entries are probably the last entries before SourcePoint crashes. These are extremely helpful to the development team.

Log Window Menu

To display a **Log** window menu, a **Log** window must be on your screen. The **Log** window menu can then be found on the menu bar or made available by right-clicking in the **Log** window.



Log window menu

Filter menu item. Various filters can be used to collect different types of data, including **Fatal Errors**, **Non-Fatal Errors**, **Warnings**, **Information**, and **Log Errors Only**.

Display menu Item. The **Display** menu item allows you to choose between icons and text to describe the type of log entry and enable/disable the **Date/Time** column, the **Code** or **File [Line]** column and the **Component** column. It also allows you to display a single line of code. The column-related options are described in more detail below.

- **Type Icon option.** Select the **Type Icon** option to change the method of displaying the type of log event. If enabled, small icons are displayed. If disabled, the corresponding word (e.g. **Error**, **Fatal**) is displayed. The **Type** column contents display in black by default. Switching to a text display may be helpful before saving the log contents to a file or for a **Copy** command.
- **Date/Time option.** Enable the **Date/Time** option to show the date and the time that the entry was made in the log. The **Date** and **Time** columns display in blue by default.
- **Code option.** Enable the **Code** option to show the SourcePoint source file name and the line in the source file that generated the entry. This information makes the source of the log entry completely unambiguous and has been found to be very effective in pinpointing trouble spots. The **Code** column contents display in gray by default.
- **Component option.** Enable the **Component** column option to display the logical SourcePoint component making the log entry. The component column contents display in green by default.

Refresh menu item. The **Refresh** menu item causes the **Log** window to reinitialize completely and redisplay the log. Use this menu item if the **Log** window contents appear corrupted or out of date.

Clear menu item. The **Clear** menu item simply clears the current log file display.

Log to Disk menu item. Clicking on this menu item brings up the **Log to Disk** dialog box. From the dialog you can select the size of the log file and the number of files.

Copy menu item. The **Copy** menu item allows you to copy the **Log** window. It is accessible only in the context menu (available by right-clicking on the **Log** window).

Memory Window

Memory Window Overview

Memory Window Introduction

The Memory window is used to display and edit target memory. To open a **Memory** window, select **View | Memory** or click on the **Memory** window icon on the toolbar.

Several **Memory** windows can be opened to view different areas of memory at the same time. There is no maximum to the number of Memory Windows that can be opened; however, each opened window is refreshed from the target on run, stop, or step. The more memory that must be refreshed, the slower the windows will update.

Memory window

Display Fields

The **Memory** window has three areas: the address area, the data area and the ASCII area.

Address Area

The left side of the **Memory** window lists the starting addresses for the row of memory objects (data) to the right. All addresses are displayed as hexadecimal values. The address of the current data object at the cursor location is displayed in the Address control in the dialog bar (at the bottom of the window).

Data Area

The data area is to the right of the address. The number of memory objects in a row, the memory object size and the display radix are chosen from the drop down lists in the dialog bar.

NOTE: A question mark may be displayed in place of a data value. This indicates the target was unable to read memory (because the target is running, the address is invalid, etc.).

ASCII Area

If desired, character equivalents for the data area can be displayed on the far right of the Memory window. Options include **7-Bit ASCII**, **8-Bit ASCII**, **UTF-16LE** (Unicode 16-bit little endian) and **UTF-16BE** (Unicode 16-bit big endian). Unprintable ASCII characters are replaced with a ‘.’ character. Unprintable Unicode characters are usually replaced with a square character (this depends on the Unicode font selected).

The default Unicode font is Arial Unicode MS. If not available on the Host system, the operating system will attempt to find a comparable font. If the selected font does not work for your application, it can be overridden by adding the following entry in the SourcePoint INI file (sp.ini):

[Fonts]

Unicode=MS Mincho // select MS Mincho font for better Kanji characters

Dialog Bar

The dialog bar is found at the bottom of the Memory window.

Address Text Box

This text box displays the current address of the PC. It can be modified by over-typing to move to a new address. Recently viewed addresses can be selected from the dropdown list. Symbolic addresses can be entered directly, or the **Find Symbol** button to the right of the text box can be used. Clicking on this button causes the Find Symbol window to display. The **Find Symbol** window allows you to quickly maneuver and find any program symbol and its memory address.

For more information on the **Find Symbol** window, go to the topic, [Edit Menu](#), part of "SourcePoint Overview," under SourcePoint Environment.

Preference Drop Down Lists

The four drop down list boxes allow you to change (for the current window only) the **Size**, **Base**, **Width**, and **ASCII** preferences.

Refresh Button

The Refresh button forces the Memory view to re-read memory from the target.

Memory Window Menu

The Memory window menu can be displayed by selecting **Memory** on the top-level menu, or by right-clicking in a **Memory** window.

Size

Selects the size of memory objects to display. Memory size options are **8-bit**, **16-bit**, **32-bit**, or **64-bit**. Size can also be selected directly from the dialog bar.

NOTE: Multi-byte objects are displayed little-endian data format.

Radix

Selects the display base for memory objects. Radix options are **Hex**, **Signed 10**, or **Unsigned 10**. Radix can also be selected directly from the dialog bar.

Width

Selects the number of bytes of memory to display per row of display. Width options are from 1-byte to 64-bytes. A **fit to window** selection is also available. Width can also be selected directly from the dialog bar.

ASCII

Selects a character display mode. Options include: **No ASCII**, **7-Bit ASCII**, **8-Bit ASCII**, **UTF-16LE** (Unicode 16-bit little-endian) and **UTF-16BE** (Unicode 16-bit big-endian). Unprintable ASCII characters are replaced with a '.' character. Unprintable Unicode characters are usually replaced with a square character (this depends on the Unicode font selected).

The default Unicode font is Arial Unicode MS. If not available on the Host system, the operating system will attempt to find a comparable font. If the selected font does not work for your application, it can be overridden by adding the following entry in the SourcePoint INI file (sp.ini):

```
[Fonts]
Unicode=MS Mincho           // select MS Mincho font for better Kanji characters
```

ASCII can also be selected directly from the dialog bar.

Refresh

Forces the Memory view to re-read memory from the target. This option is also available directly on the dialog bar.

View At Address

Brings up the **Address** dialog box and allows you to view memory at the address specified. You can also type a new address in the **Address** text box in the dialog bar to change a memory address.

Viewpoint

SourcePoint 7.7.1

Allows you to track a specific processor (P0, P1, etc.) or the current viewpoint processor.

Copy/Paste

Used to copy and paste data values in the window. Ctrl-C and Ctrl-V also work.

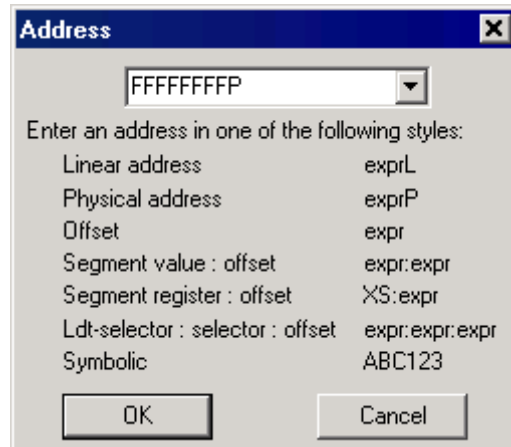
Memory Window Preferences

Open **Options|Preferences** from the menu bar and select the **Memory** tab to set preferences. For details, see the topic, "[Options Menu - Preferences Menu Item](#)" in "SourcePoint Overview" under *SourcePoint Environment*.

How To - Memory Window

How to Open a Memory Window

To open a **Memory** window, go to **View|Memory** from the menu bar or click on the **Memory** window icon. The **Address** dialog box displays showing the address of the current DS register value.



Address dialog box

Enter a starting address in the **Memory Address** text box, using one of the following address styles:

- **Linear Address (exprL)** = Real or Protected Mode.
- **Physical Address (exprP)** = Real or Protected Mode (same as Linear address if paging is not in effect).
- **Offset (expr)** = Equivalent of DS:Offset.
- **Segment Value: Offset (expr:expr)** = Value selected for segment plus value selected for offset.
- **Segment Register: Offset (XS:expr)** = Uppercase designation for CS, DS, ES, FS, GS, or SS register plus value selected for offset (e.g., CS:EIP).
- **LDTR: Selector: Offset (expr:expr:expr)** = Value selected for LDTR plus values selected for selector (segment register) and offset. This style is used in Protected mode only.

Note: Several **Memory** windows can be opened to view different areas of memory at the same time. The maximum number of open **Memory** windows is limited only by available memory in the host and available screen space. However, each opened window is refreshed from the target on run, stop or step. The more memory that must be refreshed, the slower the windows update.

How to View Memory at an Address

There are a number of ways to view memory at a particular address, depending on where you are in SourcePoint.

Getting to a Memory Window

1. If you are in a non-memory window, go to **View|Memory** on the menu bar.

The **Address** dialog box opens.

2. Enter the address you want to view in the text box.
3. Click the **OK** button.

This will bring up a **Memory** window containing the address.

Getting an Address From a Memory Window

If you are in a **Memory** window, enter the address you want to view in the text box in the left-hand corner of the dialog bar.

Alternatively, if you are in a Memory window, go to **Memory|View at Address** menu item to open the **Address** dialog box.

Address Styles

You can type in an address using any of the following address styles:

- Linear Address (exprL) = Real or Protected Mode
- Physical Address (exprP) = Real or Protected Mode (same as linear if paging is not in effect).
- Offset (expr) = Offset relative to selector CS.
- Segment Value: Offset (expr:expr) = Value selected for segment plus value selected for offset.
- Segment Register: Offset (XS:expr) = Uppercase designation for CS, DS, ES, FS, GS, or SS register plus value selected for offset.
- LDTR: Selector: Offset (expr:expr:expr) = Value selected for LDTR plus values selected for selector (segment register) and offset (this style is used in Protected Mode only).

How To Change Memory Values

1. To open the **Memory** window, go to **View|Memory** on the menu bar.
2. If desired, change the memory address range by entering a new starting address in the **Address** dialog box.

Note: If you are already in a **Memory** window, you do not need to open another one. Just change the address in the dialog bar. The **Memory** window refreshes and displays a new range of memory beginning with the specified address.

3. Insert the blinking caret immediately before the memory object to be changed.
4. Enter the new values.

The old values are overtyped. As new values are entered, the changed field turns light green.

5. Press the Enter key or click on a different address string to activate the new values.

The changed field is displayed in bright green.

Operating System Window

Operating System Windows Overview

Linux OS-Aware Debugging

The Linux-aware feature of SourcePoint provides a number of important new capabilities for users who are working on Linux-based embedded systems:

- Allows full symbolic, source-level debugging of Linux kernel code.
- Allows source-level debugging of Linux embedded applications, including the ability to start or stop a Linux process, attach to a process, view source and symbols for a process, and set breakpoints within a process.
- Allows Process breakpoints to stop the execution of a process without stopping the processor or causing it to enter debug mode
- Debugs relocatable/dynamically-loaded kernel modules.
- Hosts Linux console devices from within SourcePoint (the **Target Console** window), eliminating the need for a serial port or video device on the target and simplifying the debugging of ”headless systems."

SourcePoint allows concurrent debugging of Linux kernel code and Linux application processes. Within SourcePoint, two new views provide the user interface to Linux-aware debugging features. The **Operating System** window lists Linux processes and serves as the primary interface for task debugging. The **Target Console** window emulates multiple terminals which serve as the Linux system console and as the standard input and output device for processes launched for debugging.

The Arium debug agent must be running on the target as a prerequisite to debugging applications.

Modes of Operation

Note: In SourcePoint, the term "task" is often used to refer to a Linux process. For the purpose of this documentation, the term task and Linux process should be considered synonymous.

The Linux-aware feature of SourcePoint operates in two modes: Halt mode and Task mode.

- Halt mode is familiar to experienced SourcePoint users. The **Stop** icon causes the target processor to stop and enter debug mode. Breakpoints also cause the target processor to stop and enter debug mode. Processor registers and memory are accessible only when the target is stopped. This mode allows breakpoints to be set in the Linux kernel.
- Task mode is new with SourcePoint 7.7.1. You can start one or more Linux processes for debugging or attach to existing processes for debugging. The **Stop** icon stops only the process that is currently under SourcePoint control. The target processor is not stopped and does not enter debug mode. Other processes continue to run, as does the Linux kernel. Task-specific breakpoints can be set. Task registers and memory are accessible only when the target processor is running and the specific task is stopped.

The **Task** List view is available only in Task mode.

Debugging Context

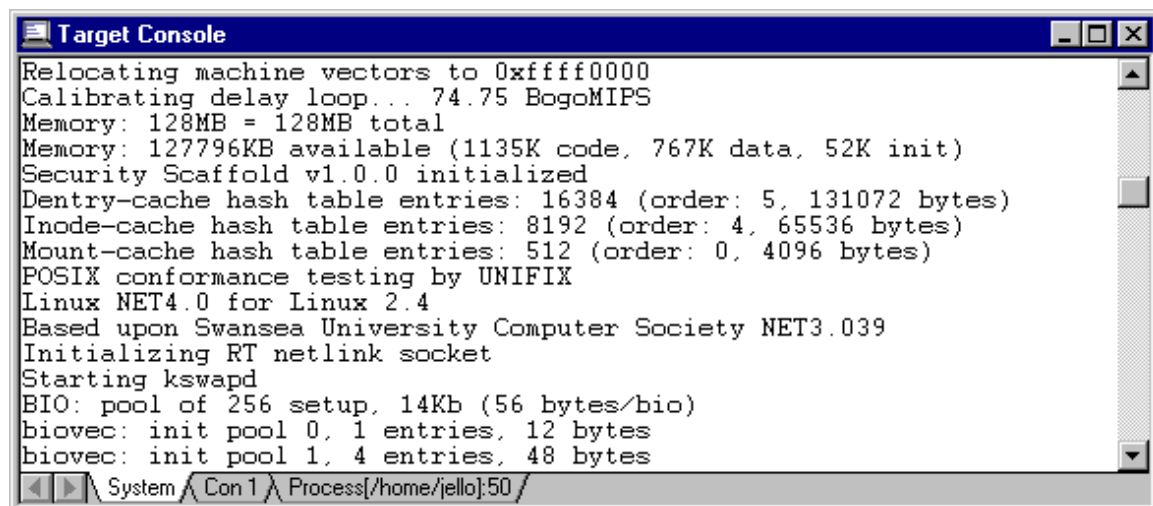
In SourcePoint, *Processors* and *Tasks* are each associated with a debugging context which is used to access pertinent state information such as memory, registers, program symbols, and breakpoints. SourcePoint operates by default on the “focus” or *Viewpoint* context, which is selectable within the constraints of modes of operation. All views displaying context-related state information show the context name in the title bar.

Introduction to OS Aware Windows

There are two windows associated with Linux debug: the **Target Console** window and the **Operating System Resources** window. They are described in more detail below.

Target Console Window

Select **View|Operating System|Target Console** on the menu bar or click on the **Target Console** icon on the toolbar to access the **Target Console** window. The window contains tabbed views for each Linux console. Each view implements an ANSI VT100 serial terminal display/keyboard device. There are two categories of consoles: static and dynamic. Each static console has a dedicated channel and tab; the number of these is a Linux-aware configuration option. Dynamic consoles exist only for the lifetime of the corresponding agent on the target, which can be performing task debugging or a file transfer.



Target Console window

Target Console Menu

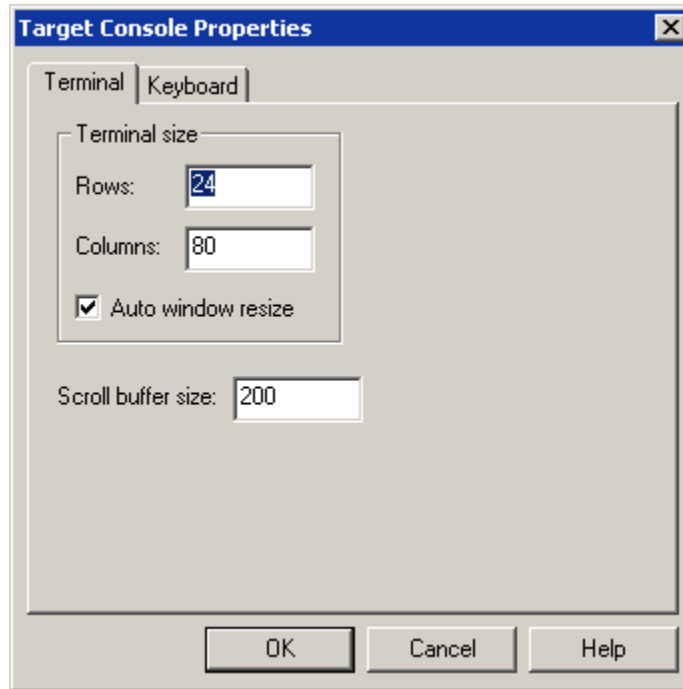
Right-clicking in the **Target Console** window displays a menu with two items: **Clear buffer** and **Properties**.

Clear Buffer menu item. This option clears the buffer.

Properties menu item. This option displays a **Target Console Properties** dialog box with two tabs: **Terminal** and **Keyboard**.

- **Terminal Tab**
 - **Rows.** You can key in the number of rows you want to show in the window.
 - **Columns.** You can key in the number of columns you want to show in the window.

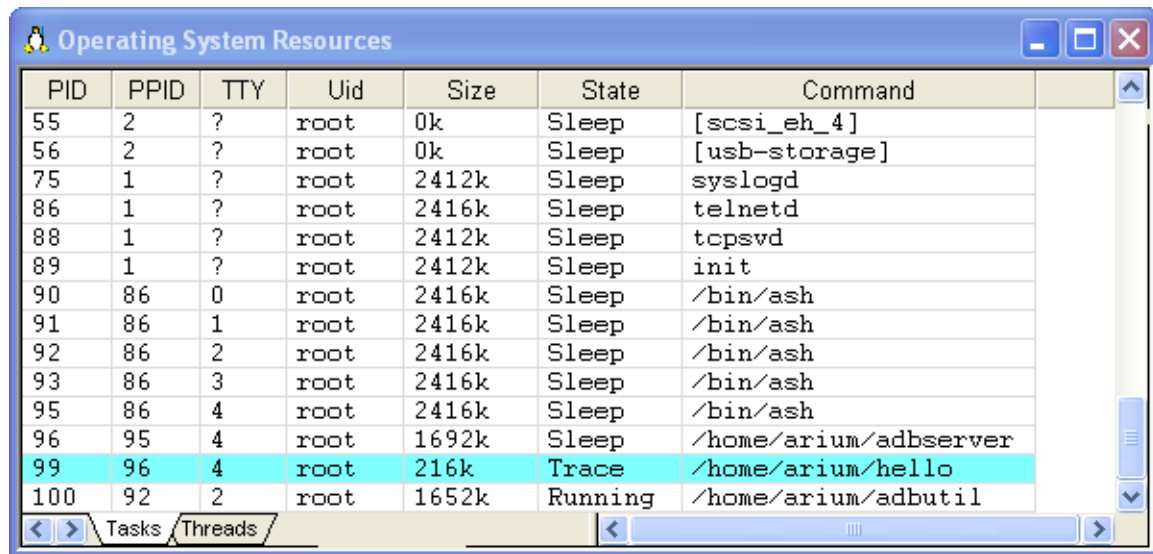
- **Auto window resize.** You enable the option that automatically resizes the window after you have changed the number of rows and/or columns you want to see in the Target Console window. It is enabled by default.
- **Scroll buffer size.** This options let you choose how many lines of history you want to keep before the buffer is emptied.
- **Keyboard Tab** - From this option you can send a hex value, (depending on target TTY settings) or a standard ASCII delete command.



*Target Console Properties dialog box showing **Terminal** tab*

Operating System Resources Window

To open an **Operating System Resources** window, select **View|Operating System|Resources**. The **Operating System Resources** window displays the tasks running under Linux. Tasks being debugged by SourcePoint are denoted by a light blue background.

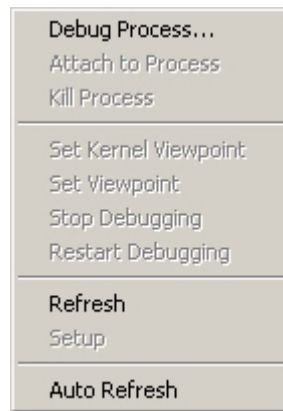


PID	PPID	TTY	Uid	Size	State	Command
55	2	?	root	0k	Sleep	[scsi_eh_4]
56	2	?	root	0k	Sleep	[usb-storage]
75	1	?	root	2412k	Sleep	syslogd
86	1	?	root	2416k	Sleep	telnetd
88	1	?	root	2412k	Sleep	tcpsvd
89	1	?	root	2412k	Sleep	init
90	86	0	root	2416k	Sleep	/bin/ash
91	86	1	root	2416k	Sleep	/bin/ash
92	86	2	root	2416k	Sleep	/bin/ash
93	86	3	root	2416k	Sleep	/bin/ash
95	86	4	root	2416k	Sleep	/bin/ash
96	95	4	root	1692k	Sleep	/home/arium/adbserver
99	96	4	root	216k	Trace	/home/arium/hello
100	92	2	root	1652k	Running	/home/arium/adbutil

After a task is launched, it is added to the **Operating System** window **Task List**

Task Context Menu

To display the context menu for a task, select a process and right-click in the **Operating System Resources** window.



Linux task context menu

Debug Process Menu Item. Use this menu item to initiate a program in a new process for debugging.

Attach to Process Menu Item. Use this command to intercept the selected process for debugging.

Kill Process Menu Item. This menu item sends a SIG_TERM signal to the selected process. Not functional at this time.

Set Kernel Viewpoint Menu Item. This menu item causes SourcePoint to switch focus to the *Processor* context and enter Halt mode.

Set Viewpoint Menu Item. This menu item causes SourcePoint to switch focus to the *Task* context for the selected task.

Stop Debugging Menu Item. This menu item abandons debugging on the selected task. If debugging was initiated by the Debug Process command, the process ends. If debugging was initiated by the Attach to Process command, the process continues to run.

Restart Debugging Menu Item. This menu item causes the program under debug to be restarted. Not functional at this time.

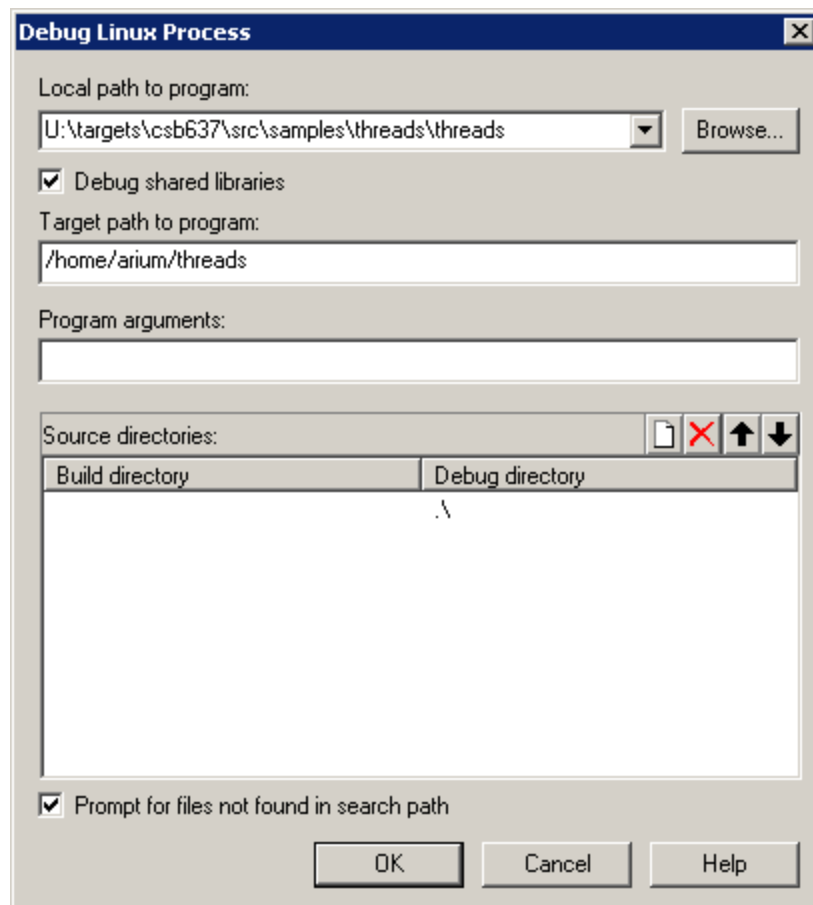
Refresh Menu Item. This menu item refreshes contents of the Operating System view.

Setup Menu Item. This menu item currently is not functional.

Auto Refresh Menu Item. This menu item enables the timed refresh of the **Task List** view. The view refreshes every 5 seconds..

To Begin Linux Debug

To debug a Linux program, right-click in the window to summon the context menu. From the menu select **Debug Process**. This displays the **Debug Linux Process** dialog box.



Debug Linux Process dialog box

Local path to program. This field specifies the full path on the local system to the executable with debugging symbols. Either select an entry from the combo drop-down list or use the **Browse** button to navigate to the desired program.

Debug shared libraries. Enable this option if you want to debug shared libraries.

Target path to program. This field specifies the full path on the target system to the executable to be debugged. The value in this field is used to launch the process executable on the target. It is disabled for *Attach* operations.

Program arguments. This field is used to specify any command line arguments to be supplied to the new process. It is disabled for *Attach* operations.

Source directories. This list is a map relating the directories in the symbols from the Build system top to the corresponding directories on the local (Debug) system. It is used for the purpose of locating source files. If the Build and Debug paths are identical, leave the list blank. Otherwise, specify only the leftmost part of the path that varies.

Example: /home/fred/linux-42 -> c:\linux-42

This map is also populated automatically by SourcePoint whenever you are asked to locate source.

Press the **OK** button to start the process.

Prompt for files not found in search path. Enabling this option causes SourcePoint to search for files not found in the search path.

Debugging Under Linux

The primary goal of the SourcePoint Linux-aware feature is to provide concurrent source-level debugging support for the Linux kernel and application processes, using the same user interfaces with the capability to transition seamlessly between the two modes.

When Task mode debugging is initiated by **Debug Process**, SourcePoint switches the focus context to the new task, which is then stopped at its entry point. The title bars for all views tracking the focus context display the program name and Linux process ID (PID). When Task mode debugging is initiated by **Attach to Process**, the selected task is stopped at the point of intercept, which is often at the return from a system call.

Once Task mode is entered, SourcePoint behavior for most views is essentially the same as Halt Mode. The most notable difference is in the lifetimes of Task and Processor contexts. Processors are perpetual while Tasks are transient. When a Task ends, its context is destroyed, and SourcePoint switches its focus to another Task. When the last Task context exits, SourcePoint switches to the kernel (Processor) context and enters Halt mode running. When the kernel hits a breakpoint in, say, a device driver or system call, SourcePoint switches to the kernel context and enters Halt mode. The usual run controls then can be used to debug the kernel. To resume Task debugging, use the **Go** command, then select the desired task in the System Resources Task list and use its context menu **Set Viewpoint** menu item to switch to the task.

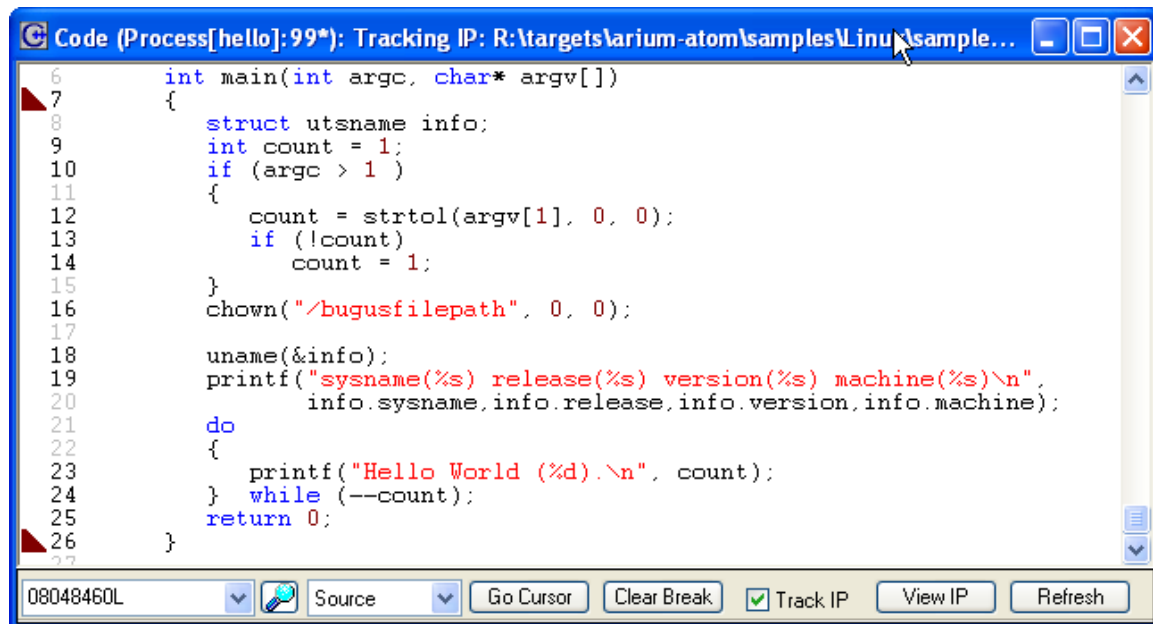
Open a **Code** view. From there you can go and step and set task breakpoints. You can also set breakpoints from the standard SourcePoint **Breakpoints** window. For more information, see,

"How to Set Breakpoints in Linux From the Breakpoints Window," part of "How To - Operating System Windows," found under *Operating System Windows Overview*.

Debugging with Shared Libraries

Support for shared library debugging is activated automatically whenever the process executable contains an ELF .dynamic section. During activation, symbols are loaded for the dynamic linker named in the ELF .interp section. The dynamic linker resolves shared library symbol references at runtime. It also provides hooks that enable a debugger to track the loading and unloading of shared libraries in the local process. SourcePoint uses the macro-on-breakpoint mechanism to provide the appropriate symbol handling.

When symbols are loaded for a shared library, the **Symbols** and **Symbol Finder** windows can be used to navigate source and set task breakpoints. Task breakpoints loaded from a project file are reactivated automatically when the associated executable or library is loaded. This feature facilitates the debugging of shared library constructor (init) functions.

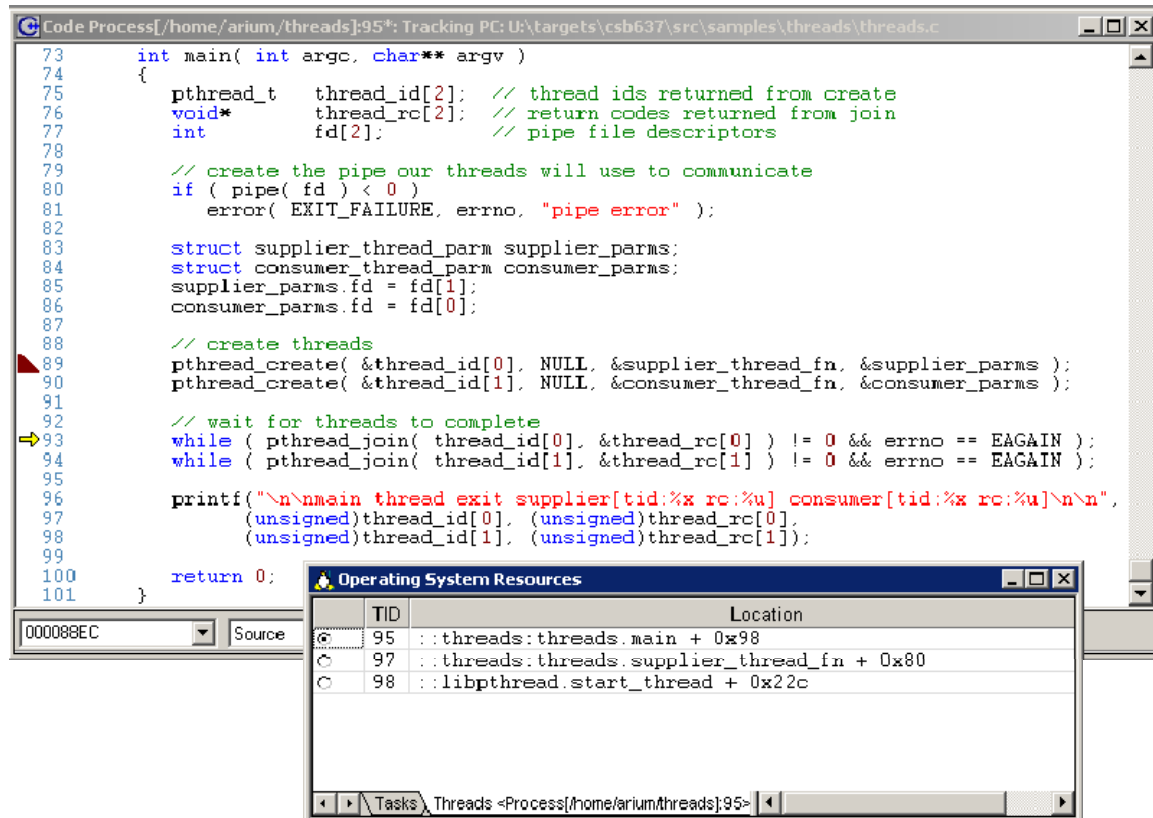


Shared library debugging with breakpoints set in library constructor

Thread Support

A task always begins with a single thread. Additional threads may be created and destroyed by an application process during its lifetime. A list of the active threads for the viewpoint task context is provided by the **Threads** tab in the **Operating System Resources** window. The window contents are updated continuously as threads are created and destroyed by the viewpoint task. The view is completely refreshed whenever the viewpoint context changes. This is the UI control for selecting the thread context to be displayed in all other views pertinent to a task (e.g., registers, code, stack, etc.).

Note: Thread debugging requires that the **Debug Shared Library** option be enabled for the task.



Active threads in process with focus on main thread

How To - Operating System Windows

How to Set Breakpoints in Linux From the Breakpoints Window

SourcePoint includes a new class of breakpoints known as task breakpoints. As the name indicates, this break type applies to a specific task and is available only when you are in Task mode.

To set a task breakpoint from the **Breakpoints** window:

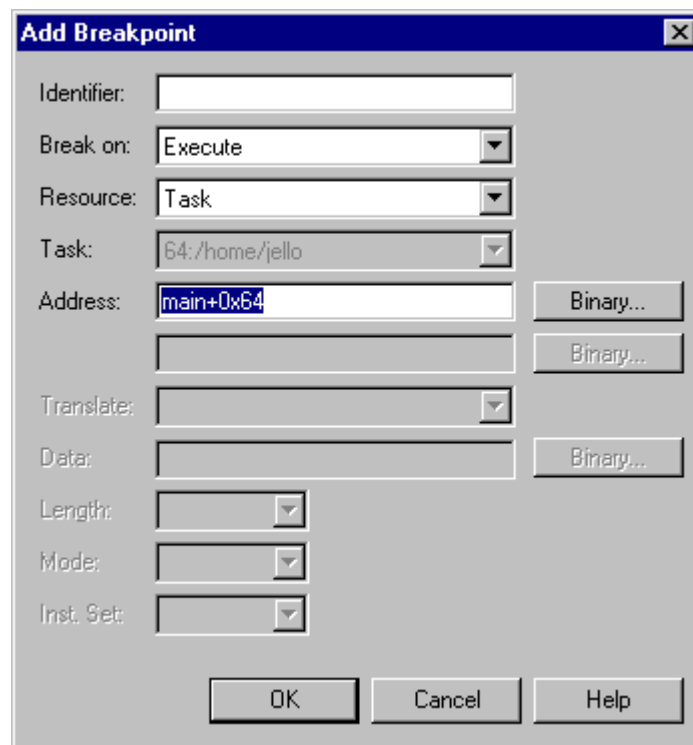
1. Select **View|Breakpoints**.

The **Breakpoints** window displays.

2. Click on the **Add** button.

The **Add Breakpoint** dialog box displays.

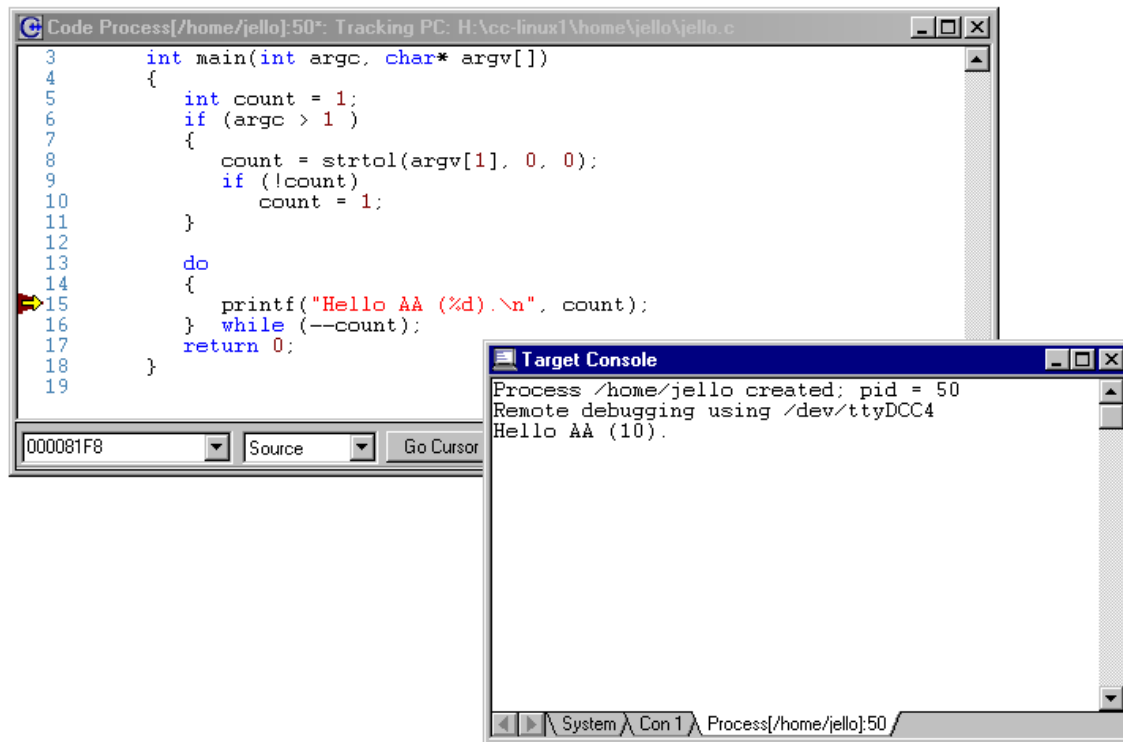
3. From the Identifier dialog box, specify an identifier for a breakpoint. If no value is entered, a default identifier (event# where # is some number) is used.
4. From the **Break On** drop down box, select **Execute**.
5. From the **Resource** drop down box, select **Task**.
6. In the **Address** text box, key in the location of the task breakpoint you want to add.



Breakpoints window dialog box

The **Breakpoints** window re-displays with the breakpoint listed in the Breakpoint list box.

Note: A target reset causes SourcePoint to switch from Task mode to Halt mode. However, all breakpoints previously set still display in the Breakpoints list box, including task breakpoints. These task breakpoints become active again if and when the applicable Linux process becomes the current process again.



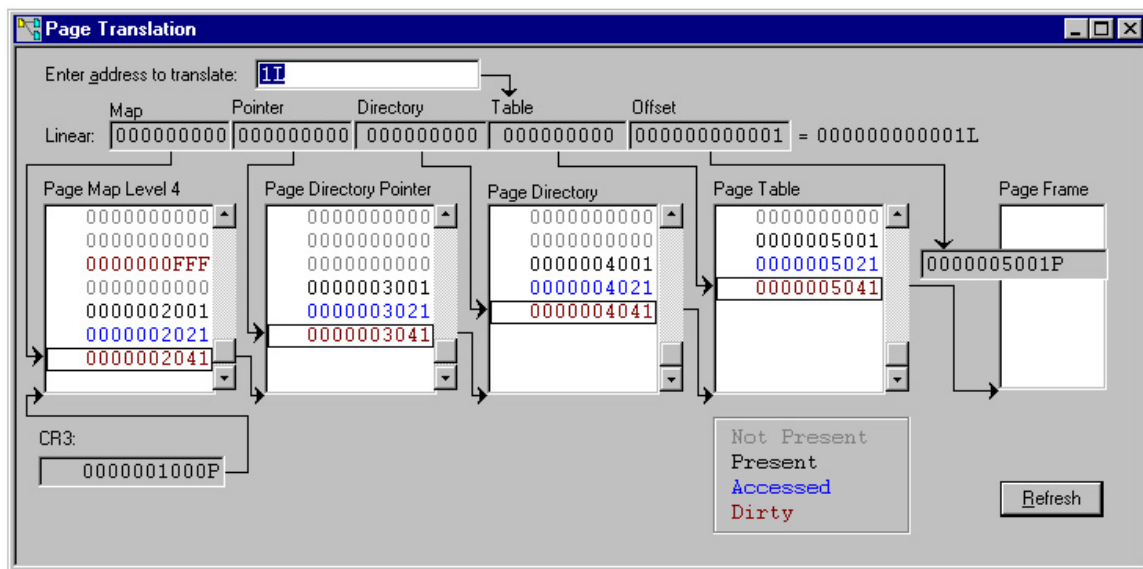
Code Process window showing breakpoints executed under Linux as indicated in the **Target Console** window.

Page Translation Window

Page Translation Windows Overview

Page Translation Window Introduction

The **Page Translation** window is used to look at the memory paging feature of a processor. The window displays a pictorial representation of the address translation process that occurs in paging. The exact representation in the window may change as different varieties of paging are in use (e.g., 4K vs. 2M pages) or the processor type (e.g., Intel or AMD), but, in general, it is a good representation of what is displayed. To open the **Page Translation** window, go to **View|Page Translation** or click on the **Page Translation** icon on the toolbar.



Page Translation dialog box (AMD processor)

Page Translation Window Elements

Address Field

The **Enter address to translate** field allows entry of address in various formats (e.g., linear, segment:offset, selector:offset, segment register:offset). This address is translated into a linear address and inserted into the **Linear** field.

Linear Fields

The **Linear** fields are displays of the resultant linear address from the **Address** field. The display is in binary, divided into the various components. The hex value is shown to the right.

The **Linear** address is divided into various components that depend on page size or processor type. As a result, **Page Map Level 4**, **Page Directory Pointer**, and **Page Table** may not be visible.

Tables

The **Page Translation** dialog box allows scrolling in the **Page Map Level 4**, **Page Directory Pointer**, **Page Directory** and **Page Table** fields. This enables exploration of the mapping of pages without having to enter a new value in the **Address** field. Simply click the mouse on or scroll to an entry in the one of the table list boxes, and the corresponding entry is activated.

The entries in these tables are color-coded, which speeds up the interpretation of the table state and structure.

- Grey = Not Present
- Black = Present
- Blue - Accessed
- Red = Dirty

Placing the cursor on a current value causes a flyover tooltip to display the attributes of the entry.

Page Frame

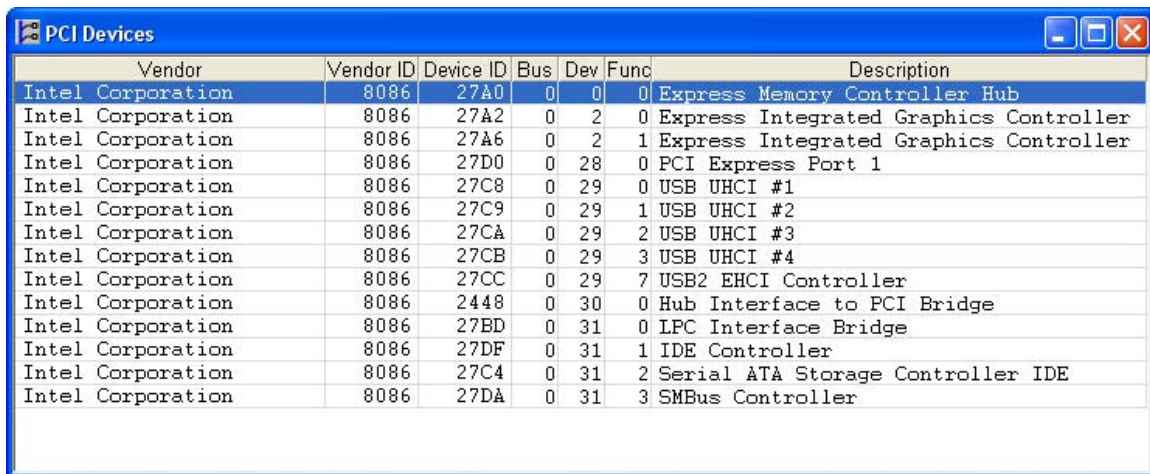
This field displays the resultant translated physical address of the corresponding linear address. In some cases the linear and physical addresses may be the same.

PCI Devices Window

PCI Devices Window Overview

PCI Devices Window Introduction

The PCI Devices window displays basic information for the PCI devices on the target. It scans the PCI buses you specify, using a process called PCI device enumeration, and displays a summary of each PCI device found ordered by its bus, device, and function numbers. Select View | PCI Devices in the menu or click the PCI Devices icon on the toolbar to access the PCI Devices window.



The screenshot shows a window titled "PCI Devices" with a table of PCI devices. The table has columns for Vendor, Vendor ID, Device ID, Bus, Dev, Func, and Description. The data is as follows:

Vendor	Vendor ID	Device ID	Bus	Dev	Func	Description
Intel Corporation	8086	27A0	0	0	0	Express Memory Controller Hub
Intel Corporation	8086	27A2	0	2	0	Express Integrated Graphics Controller
Intel Corporation	8086	27A6	0	2	1	Express Integrated Graphics Controller
Intel Corporation	8086	27D0	0	28	0	PCI Express Port 1
Intel Corporation	8086	27C8	0	29	0	USB UHCI #1
Intel Corporation	8086	27C9	0	29	1	USB UHCI #2
Intel Corporation	8086	27CA	0	29	2	USB UHCI #3
Intel Corporation	8086	27CB	0	29	3	USB UHCI #4
Intel Corporation	8086	27CC	0	29	7	USB2 EHCI Controller
Intel Corporation	8086	2448	0	30	0	Hub Interface to PCI Bridge
Intel Corporation	8086	27BD	0	31	0	LPC Interface Bridge
Intel Corporation	8086	27DF	0	31	1	IDE Controller
Intel Corporation	8086	27C4	0	31	2	Serial ATA Storage Controller IDE
Intel Corporation	8086	27DA	0	31	3	SMBus Controller

PCI Devices window

When you open the PCI Devices window, it first displays the Refresh PCI Devices dialog box, which requires you to specify the starting and ending PCI bus numbers to scan. Click the Refresh button to start PCI device enumeration.

While the PCI buses are being scanned, the PCI Devices window is filled in with the PCI functions found while a progress bar is displayed. You can cancel the scan prematurely by clicking the Cancel button. There can be a maximum of 256 PCI buses on a target. Each bus can have a maximum of 32 devices, and each device can have a maximum of 8 functions. The PCI Devices window displays a function on each row of the grid. Each PCI device's function has 256 bytes of configuration registers.

PCI Devices Window Columns:

- The Vendor column displays the manufacturer's name string.
- The Vendor ID column displays the manufacturer's unique 16-bit identifier in hexadecimal.
- The Device ID column displays the device's unique 16-bit identifier in hexadecimal.
- The Bus column displays the bus number in decimal.

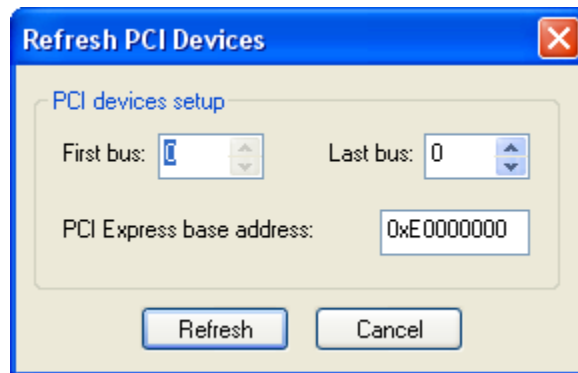
- The Dev column displays the device number in decimal.
- The Func column displays the function number in decimal.
- The Description column displays the device's description string.

The PCI Devices window is resizable. All columns are of fixed width except the Description column, which automatically resizes to fit the window.

Note: Opening the PCI Devices window immediately after target reset may not reveal all PCI devices on the target. Some chipset initialization may be required to enable all devices to be found during PCI device enumeration. The PCI Express configuration space also may not be available at reset.

Refresh PCI Devices Dialog Box

The Refresh PCI Devices dialog box lets you specify the starting and ending PCI bus numbers to scan, as well as the base memory address for PCI Express devices.



Refresh PCI Devices Dialog

Open the Refresh PCI Devices dialog box by clicking on the option from the menu or by right clicking in the window. Enter the first bus and last bus to scan. Scanning all PCI buses, from 0 to 255, may take considerable time. It is recommended that you start by scanning buses 0 to 3.

Enter the base memory address for PCI Express devices. This 8-digit hexadecimal number, for example E0000000, indicates the location in target memory where the PCI Express configuration space is located. Since this value is target specific, it cannot be automatically determined.

Click the Refresh button to begin scanning.

PCI Devices Window Menu

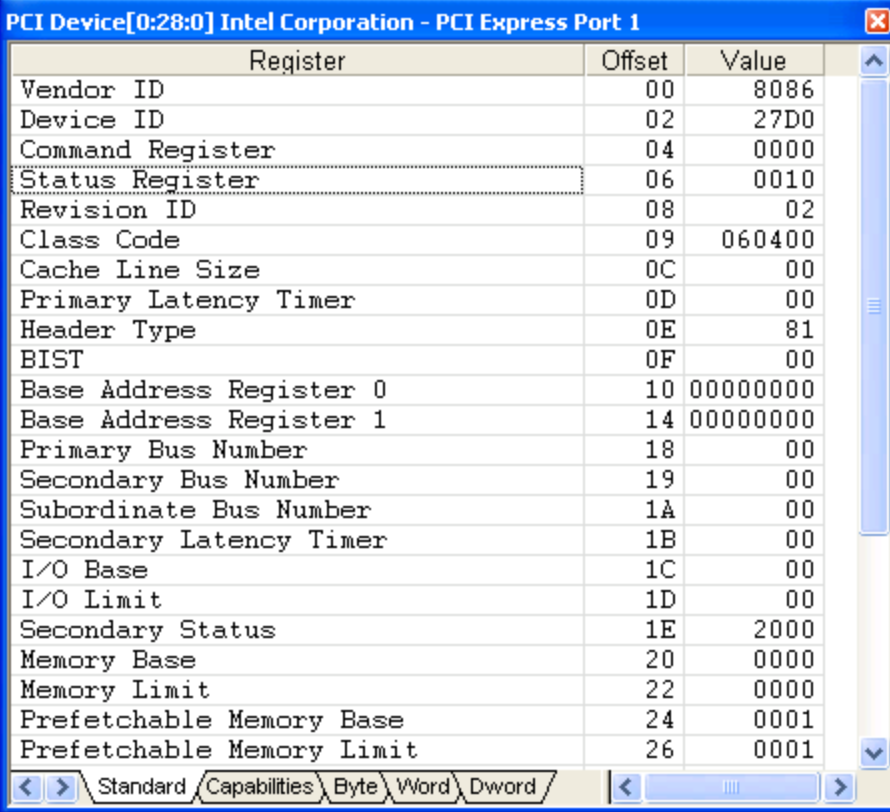
The context menu contains two items, Refresh and View Registers.

Refresh. Opens the Refresh PCI Devices dialog box.

View Registers. Opens the PCI Registers dialog box. The PCI Registers dialog box can also be opened by double-clicking an entry in the PCI Devices window.

PCI Registers Dialog Box

You can view more detailed information about a PCI device by opening the PCI Registers dialog box. Here you find a list of the configuration registers and device capabilities for the currently selected PCI function. You can display multiple PCI Registers dialog boxes for different PCI devices at the same time.



Register	Offset	Value
Vendor ID	00	8086
Device ID	02	27D0
Command Register	04	0000
Status Register	06	0010
Revision ID	08	02
Class Code	09	060400
Cache Line Size	0C	00
Primary Latency Timer	0D	00
Header Type	0E	81
BIST	0F	00
Base Address Register 0	10	00000000
Base Address Register 1	14	00000000
Primary Bus Number	18	00
Secondary Bus Number	19	00
Subordinate Bus Number	1A	00
Secondary Latency Timer	1B	00
I/O Base	1C	00
I/O Limit	1D	00
Secondary Status	1E	2000
Memory Base	20	0000
Memory Limit	22	0000
Prefetchable Memory Base	24	0001
Prefetchable Memory Limit	26	0001

Standard Capabilities Byte Word Dword

PCI Registers Dialog

The Registers column displays the name of the register as a text string. The Offset column denotes the register's location (byte offset) in hexadecimal with respect to the start of the function's configuration data. The Value column displays the value of the register in hexadecimal. Note that register sizes vary.

Click on one of the tabs at the bottom to change the display format. The Standard tab displays the standard PCI-compatible register set. The Capabilities tab displays the PCI Capability and PCI Express Extended Capability register sets. The Byte, Word, and Dword tabs display all the registers in a hexadecimal format, 256 bytes for PCI functions and 4096 bytes for PCI Express functions.

Enable Auto Update to have the Registers dialog box automatically refresh itself each time the target stops.

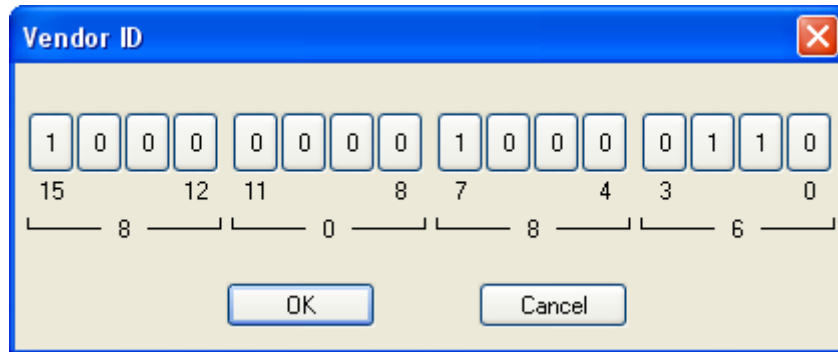
This is a resizable modeless dialog box, meaning that it stays on top of other windows and allows you to switch to other windows while staying active until you close it.

Registers Dialog Box Menu

The context menu contains four items: Edit, Expand, Refresh and Auto Update.

Edit. Puts the currently selected register's Value cell in edit mode for modifying. Hit the <Enter> key when done editing a value. All the registers are read back from the target after editing a register in case the modification affects other register values.

Expand. Opens the Expand dialog box which allows editing of individual bits.



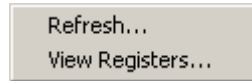
Expand Dialog

Refresh. Reads the register values from the target and updates the grid.

Auto Update. Causes the registers to be refreshed from the target when ever the target stops.

PCI Devices Window Menu

The context menu contains two items, **Refresh** and **View Registers**. Selecting **Refresh** on the context menu opens the **Refresh PCI Devices** dialog box. Selecting **View Registers** opens the **PCI Registers** view.



PCI Devices menu

View Registers menu item. The **PCI Registers** view is opened by double-clicking an entry in the **PCI Devices** window or via the context menu. It displays detailed information for the specific PCI device, including the name and values of all registers. You can change the PCI device shown by selecting a different entry in the **PCI Devices** window while the **PCI Registers** view is open. The name and location of the registers may change, depending on the type of device shown.

PCI Registers[00:1d:00]: Intel Corporation:USB UHCI Controller #1		
Register	Offset	Value
Vendor ID	00	8086
Device ID	02	24D2
Command Register	04	0000
Status Register	06	0000
Revision ID	08	00
Class Code	09	000000
Cache Line Size	0C	00
Latency Timer	0D	00
Header Type	0E	0
BIST	0F	00
Base Address Register 0	10	00000000
Base Address Register 1	14	00000000
Base Address Register 2	18	00000000
Base Address Register 3	1C	00000000
Base Address Register 4	20	00000000
Base Address Register 5	24	00000000
CardBus CIS Pointer	28	00000000
Subsystem Vendor ID	2C	00
Subsystem ID	2E	00
Expansion ROM Base Address	30	00000000
IRQ Line	3C	00

PCI Registers view

How To - PCI Devices Window

How to Open the PCI Registers View From the PCI Devices Window

1. Open the **PCI Registers** view by double-clicking an entry in the **PCI Devices** window, by clicking on **View Registers** from the menu bar, or by right-clicking an entry and selecting **View Registers** from the context menu.

The **PCI Register** view displays, showing the standard set of PCI registers for the currently selected PCI function in the **PCI Devices** window.

2. Change the PCI device shown by selecting a different entry in the **PCI Devices** window while it is open.

The standard registers are shown by default.

3. Use the **PCI Registers** view context menu to choose the display format and control automatic updating.

Standard Registers only displays the standard PCI-compatible registers set. **Byte**, **Word**, and **Dword** items display all the registers in a hexadecimal format. This is 256 bytes for PCI functions and 4096 bytes for PCI Express functions.

4. Enable **Auto Update** to have the **PCI Registers** view refresh itself each time the target stops.

How to Refresh a PCI Devices Dialog Box

1. Open the **Refresh PCI Devices** dialog box by clicking on the option from the menu bar or by right clicking in the window.

The **Refresh PCI Devices** dialog box displays.

2. Enter the first bus and last bus to scan.

Scanning all PCI buses, from 0 to 255, may take considerable time. It is recommended that you start by scanning buses 0 to 3.

3. Enter the base memory address for PCI Express devices.

This 8-digit hexadecimal number, for example E0000000, indicates the location in target memory where the configuration space of PCI Express devices starts. Since this value is target specific, it cannot be automatically determined.

4. Click the **Refresh** button to begin scanning.

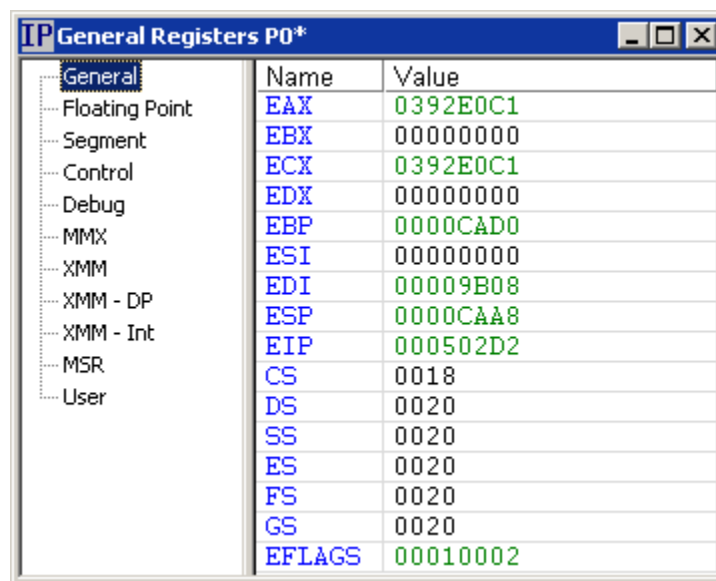
Registers Window

Registers Window Overview

Registers Window Introduction

To open the **Registers** window, select **View|Registers** on the menu bar.

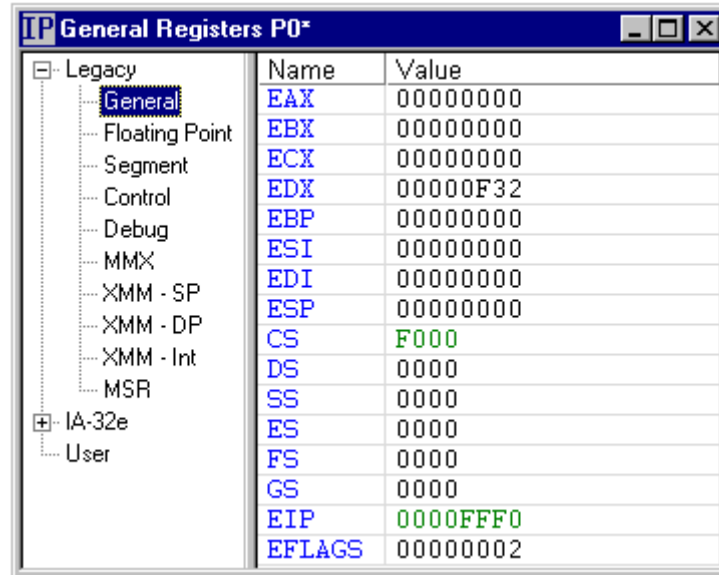
The **Registers** window displays processor registers. It is a splitter window with two panes. The left-hand pane contains a list of register types that are available for viewing. The number of register types varies by processor. (By clicking on **User**, you can create your own custom register set.) The right-hand pane displays the processor registers contained in a selected register-type list. Up to 16 **Registers** windows can be open at the same time.



Name	Value
EAX	0392E0C1
EBX	00000000
ECX	0392E0C1
EDX	00000000
EBP	0000CAD0
ESI	00000000
EDI	00009B08
ESP	0000CAA8
EIP	000502D2
CS	0018
DS	0020
SS	0020
ES	0020
FS	0020
GS	0020
EFLAGS	00010002

*List of **General** registers in the **Registers** dialog box (IA-32 processor)*

Processors with a large number of register lists (e.g., Intel's 64-bit extensions to 32-bit processors) offer register types and sub-types. Groups can be expanded and contracted by clicking on the box with the + or -.



Name	Value
EAX	00000000
EBX	00000000
ECX	00000000
EDX	00000F32
EBP	00000000
ESI	00000000
EDI	00000000
ESP	00000000
CS	F000
DS	0000
SS	0000
ES	0000
FS	0000
GS	0000
EIP	0000FFF0
EFLAGS	00000002

*List of **General** registers in the **Registers** dialog box
(Intel 64-bit extension and AMD64 processors))*

Multi-Processor Systems

If SourcePoint detects more than one target processor, selections are added to the **Register** menu and to the context menu for selecting a processor.

By default, the **Track Viewpoint** menu item in the register menu is selected. This indicates that the **Registers** window are displaying registers from the current viewpoint processor. (For more information on viewpoint, see "[View Menu](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*). If a specific processor is selected, then the registers from that processor are always displayed, regardless of the current viewpoint.

The current processor selection is indicated in the **Registers** window title bar at the bottom of your monitor screen.

Viewing the Registers Window

All register values, except for floating point registers, are displayed in hex. Floating point registers are displayed in scientific notation. Register values may be displayed and edited in binary by opening the **Expand** window. (For more information on how to access the **Expand** window, see below.)

The flyover help for a register value varies depending on the register type. If the register contains sub-fields, then the field names and values are displayed. For instance, the flyover help for EFLAGS displays all the processor flags and their values. The flyover help for floating point registers displays the value in hex. The flyover help for segment registers (CS, SS, etc.) displays the base, limit, and access rights associated with that register.

64-bit register values are displayed with a '#' character separating the upper and lower 32 bits of the value.

Modifying the Registers Window

Color Coding

Modified registers are colored green. This indicates that the register's value has changed as a result of a **go** or **step** operation or that the value has been edited.

Note: Register values are saved prior to a **go** or **step** operation. To save time, only the currently displayed register values are saved. This means that only currently displayed registers are subject to colorization. If you scroll a register into view that was not visible when the last **go** or **step** operation was performed, then its value is colored black.

Read-only register values are displayed in gray rather than black. Write-only register values are displayed as a string of asterisks rather than numbers.

Customizing the Registers Window

All changes to the **Registers** window are saved in the project file. These changes include register list changes, column widths for each register list, and window and pane sizes, along with number and location of windows. The currently selected register list and processor are also saved.

Adding or Reordering Registers

- Registers lists can be reordered. Simply drag a register name to a new position in the current list. Drop the register where you would like to insert it.
- Registers can be added to a list. Either drag the register name to the left-hand pane and drop it on another register type on the list or drag it to the right-hand pane of another **Registers** window.
- Registers can be removed from a list. Select a register, right-click, and then select **Remove Register**.
- Register lists can be restored to their original content and order by right-clicking and selecting **Restore Defaults**.

Resizing the Window

- The size of the left-hand and right-hand panes can be adjusted by clicking on the pane separator and moving it back and forth.
- The **Name** column can be resized by clicking on the column separator (in the heading), and moving the separator back and forth. The **Name** column can be automatically sized to the longest visible name by double-clicking on the column separator (in the heading). The **Value** column automatically re-sizes to fit the right pane.

User Register List - The Ultimate Custom Window

The **User** register window is no different than any other except that it starts out empty. Registers can be added by dragging register names from other **Registers** windows, by dragging a register name to the left-hand pane and dropping it on the **User** entry, or by right-clicking on a register name and selecting **Copy to User Page**.

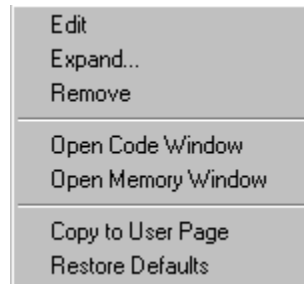
Printing Registers

SourcePoint 7.7.1

A the list of registers found under a register type can be printed by selecting **File|Print** from the menu bar. The entire list is printed.

Register Window Menu

To open the **Registers** window menu, go to **View|Registers** on the menu bar. The menu bar changes to include a **Register** drop down menu. Click on the menu.



Register menu

Edit menu item. To edit a register value, either cursor to the field and begin typing or left-click to select the field. To end editing a register value, press the Enter key or cursor to another field. If a value is being edited when either **go** or **step** is selected, the edit is terminated, the value is written to the processor, and then the **go** or **step** operation is performed.

Read-only register values are displayed in gray. These values may not be edited. Write-only register values are displayed as a string of asterisks. To edit a write-only value, type over the asterisks. When the edit is terminated, the value is written to the processor and the register value changes back to asterisks.

Register values can be copied and pasted by using the **Edit** menu item on the menu bar to copy and paste a value.

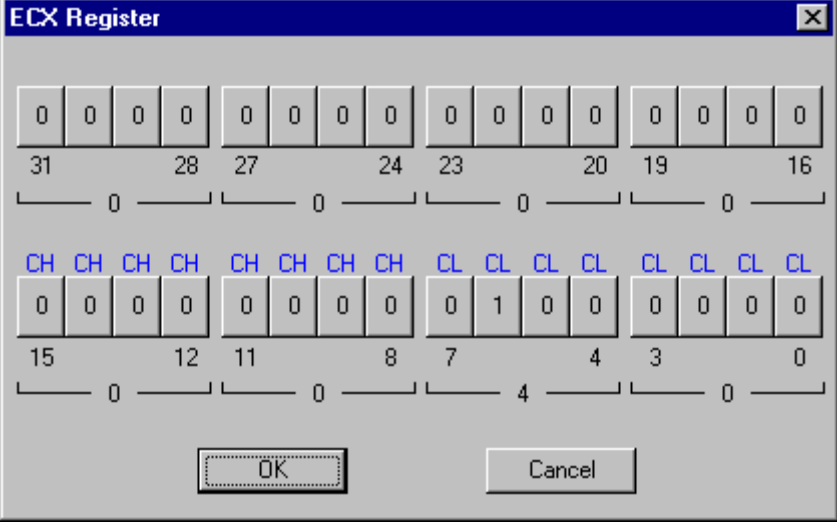
Register values may be displayed and edited in binary by opening the **Expand** window.

Expand menu item. To expand a register to view individual bits, first click on the register name or value in the **Register** window. Go to **Registers|Expand** on the menu bar or right-click and choose **Expand** from the context menu. A window appears with a bit representation of the selected register.

Each bit is displayed as a button with its value displayed in binary. The name of each bit (if any) is displayed above the button. Multi-bit fields have the same name above each button. Bit names are shortened to a maximum of three characters. The flyover help for a bit displays a longer description of that bit.

Bit numbers, along with the value of each nibble in hex, are displayed beneath the buttons.

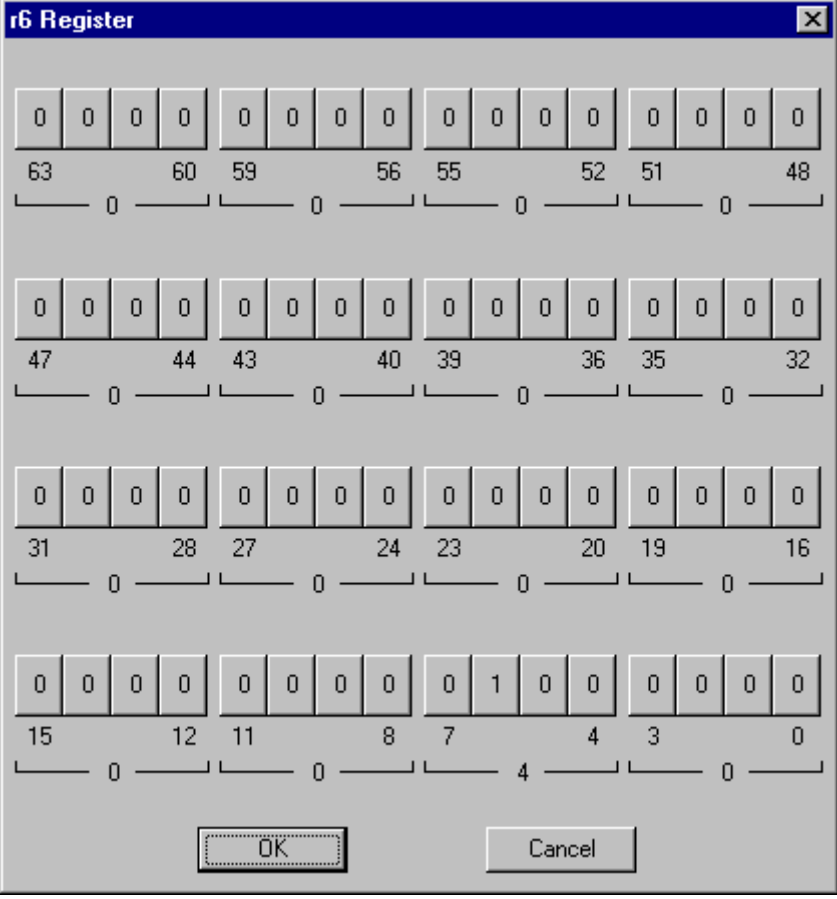
To change the value of a bit, click on the bit, or cursor to a bit and press 0 or 1. To write the value to the processor, press the **OK** button.



The ECX Register dialog box shows a 32-bit register configuration. The top row of bits (31 to 16) is all 0s. The bottom row (15 to 0) has bits 15-12 as 0s, bits 11-8 as 0s, bit 7 as 0, bit 6 as 1, bit 5 as 0, bit 4 as 0, and bits 3-0 as 0s. The labels 'CH' and 'CL' are above the bottom row of bits. The 'OK' button is highlighted.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31			28	27			24	23			20	19			16
0				0				0				0			
CH				CH				CH				CH			
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15			12	11			8	7			4	3			0
0				0				4				0			

Bit representation for the ECX register



The r6 Register dialog box shows a 64-bit register configuration. The top three rows of bits (63 to 16) are all 0s. The bottom row (15 to 0) has bits 15-12 as 0s, bits 11-8 as 0s, bit 7 as 0, bit 6 as 1, bit 5 as 0, bit 4 as 0, and bits 3-0 as 0s. The 'OK' button is highlighted.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
63			60	59			56	55			52	51			48
0				0				0				0			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
47			44	43			40	39			36	35			32
0				0				0				0			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31			28	27			24	23			20	19			16
0				0				0				0			
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15			12	11			8	7			4	3			0
0				0				4				0			

*Bit representation for the r6 register
(Intel 64-bit extension and AMD64 processors)*

Note: If you want to display a register's bits and have them remain viewable while starting/stopping/stepping, open a **Watch** tab in the **Symbols** window and drag the register name into it. Click on the "+" sign to the left of the register name to display the register's bits. the **Watch** tab window operates in real time; if the bit number changes, you will see it in this window. Alternatively, the register/bit can be added individually by manually adding its symbol to the **Watch** tab.

Remove menu item. To remove an item from the **Registers** window, begin by clicking on the item. From the **Registers** drop down menu click on the **Remove** menu item.

Open Code Window menu item. This option opens a Code Window to the address specified by the selected register.

Open Memory Window menu item. This option opens a Memory Window to the address specified by the selected register.

Copy to User Page menu item. To copy a register from the **Registers** window to a **User** page, begin by clicking on the item. From the **Registers** drop down menu click on the **Copy to User Page** menu item. The selected register is copied to the **User Page** in the **Register** window tree. This is useful to customize a register display.

Restore Defaults menu item. If this menu item is enabled, the SourcePoint default registers display is restored.

Viewpoint menu item. This menu item only displays if you are on a multi-processor target. It indicates the status of the processor viewpoint. If you have enabled one of the processor options, that processor is tracked. If you have enabled the **Track Viewpoint** is enabled, the current processor is tracked.

How To - Registers

How to Change Binary Values

The **Expand** window displays a binary graphical representation of the bits in a register value.

1. To open the **Expand** window, select a register, right-click, and select **Expand**.

Each bit is displayed as a button with its value displayed in binary. The name of each bit (if any) is displayed above the button. Multi-bit fields have the same name above each button. Bit names are shortened to a maximum of three characters. The flyover help for a bit displays a longer description of that bit.

Bit numbers, along with the value of each nibble in hex, are displayed beneath the buttons.

2. To change the value of an expanded register, choose the individual bits or cursor to a bit and press 0 or 1.

The bit values toggle as they are clicked on with the left mouse button. The hexadecimal value below each four-bit group changes accordingly.

3. Write the value to the processor.
4. Click the **OK** button.

How to Change Hexadecimal Values

There are two ways to change hexadecimal values.

1. Go to **View|Registers** on the menu bar and choose the desired registers group or specific register.

Note: Entire fields cannot be selected when changing register values. The cursor control keys or the mouse must be used to position the blinking caret immediately before the hexadecimal digits can be changed.

2. Use the mouse or cursor control keys to position the blinking caret immediately before the digit or series of digits to be changed.
3. Enter the new value for each digit, as required. Note that the caret position is overwritten as new values are entered. Use the cursor control keys to skip over unchanged digits.
4. To effect the value changes, press the Enter key or click your mouse on another register field or window.

The color of all digits in the field changes from black to green.

Note: This method replaces the entire register content.

OR

1. Right-click on a register.
2. Select **Expand** from the **Registers** menu or context menu.
3. Edit the value in place.

Symbols Windows

Symbols Window Overview

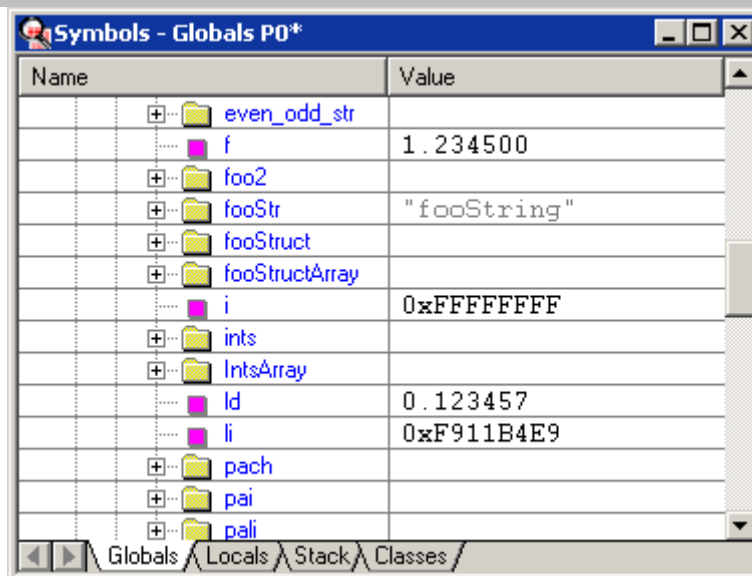
Symbols Window Introduction

To access the **Symbols** window, go to **View|Symbols** on the menu bar or click on the **Symbols** icon on the toolbar. The **Symbols** window displays symbolic debug information. You have access to all symbols and source code via the **Symbols** window.

The **Symbols** window is a tabbed dialog with four tabs (views): **Globals**, **Locals**, **Stack**, and **Classes**. Each can be accessed via a mouse click on the tab of choice, or by tabbing through them to the one you want.

- The **Globals** tab displays a hierarchy of loaded programs. Programs can be expanded to show modules, procedures, and symbols.
- The **Locals** tab shows the variables accessible in the current stack frame.
- The **Stack** tab shows the stack as a list of stack frames.
- The **Classes** tab lists structure and class definitions in a hierarchy similar to that under the **Globals** tab.

Note: If you prefer separate windows for each view rather than using the tabs, open instances of the **Symbols** window, resize each window as desired, then save the settings in a project file. Up to 16 **Symbols** windows can be open at the same time.



*Symbols view under the **Globals** tab*

General Features

Each view contains a multi-column tree control. Listed below are some of the common features found in each view.

Display Base

Values can be displayed in either decimal or hexadecimal. Select **Hexadecimal Display** in the context menu to toggle between the two. Regardless of the display base, addresses are always displayed in hexadecimal. In addition, values larger than 32 bits are always displayed in hexadecimal.

Note: There is no **Symbols** drop down menu on the menu bar. Menu items can be accessed via a context menu only. To open a context menu, right-click on a **Symbols** window.

Editing Values

Variable and register values can be edited by double-clicking the left mouse button or by selecting **Edit** from the context menu. To end editing a value, press the Enter key or select another field. If a value is being edited when either **Go** or **Step** is selected, then the edit is terminated, the value is written, and then the **Go** or **Step** operation is performed. Values can be copied and pasted by using the **Edit** menu, by using Ctrl-C/Ctrl-V, or by using drag and drop. Selecting **Undo** from the **Edit** menu restores a value to its original, unedited value. Selecting **Redo** restores the edited value.

Properties

A **Properties** dialog box can be opened by selecting **Properties** from the context menu. The information displayed varies depending on the type of item selected. Selecting a new item automatically refreshes the information displayed.

ToolTips

Most items have flyover tooltips that display some of the information available in the **Properties** dialog box.

Keyboard Support

The arrow keys provide keyboard support for navigation through the tree:

- The Up Arrow and Down Arrow keys move between items.
- The Left arrow and Right Arrow keys move along a particular branch. Pressing the Right Arrow expands a branch if it is not currently displayed. Pressing the Left Arrow moves to the first item in a branch; pressing it a second time collapses the branch.
- The Home and End keys move to the top or bottom of the tree.
- The Page Up and Page Down keys move a page at a time.
- The + and - keys expand and collapse the current tree node.
- The Enter key alternately expands and collapses the current node.
- The use of the asterisk (Shift and the number 8 on the keyboard) expands all tree nodes beneath the currently selected node.

Shortcuts

Select **Collapse All** from the context menu to collapse all nodes in the tree. Certain views also have an **Expand All** entry in the context menu which expands all nodes in the tree. The use of the asterisk can also be used to expand all nodes in the tree.

Refresh

Values are refreshed automatically when the processor runs or when a value is changed elsewhere in SourcePoint. To force a refresh of all values in a view, select **Refresh** from the context menu.

Printing

To print a view, go to **File|Print** on the menu bar. The entire tree is printed, but only currently expanded nodes are included.

Saving to a File

To save a view, go to **File|Save As** on the menu bar. The **Save As** dialog opens. Specify a file name (or use the default) and then click **OK**. The entire tree is written to the specified file name.

Colors

Colors can be changed via the Colors tab under **Options|Preferences**.







For more information, see "[Options Menu - Preferences Command](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*.

Multi-Processor Environment

In multi-processor systems, register values and stack-relative variables are always associated with the current viewpoint processor.

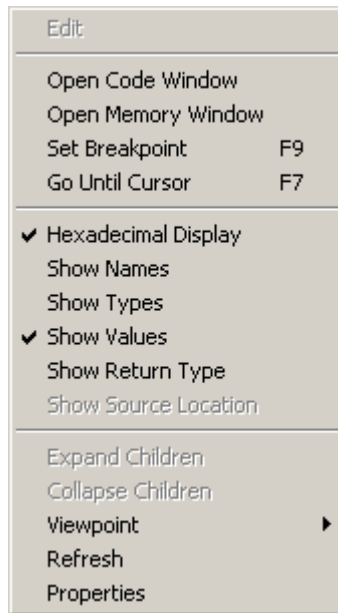
Symbols Window Icon Definitions

The icons you may find in the **Symbols** window are as follows:

	Program
	Module with source information
	Module with no source information
	Simple or terminal variable (structure element, array element, de-referenced pointer)
	Non-terminal variable (structure, array, pointer)
	Procedures within a module/methods within a class

Symbols Window Menus

There are several context menus in the **Symbols** window, depending on the type of symbol. To display the context menu associated with a symbol, click on it, then right click to bring up the menu. A typical menu is described below.



*Typical **Symbols** context menu*

Edit menu item. When a value cell is selected, you can edit the value of the associated symbol.

Open Code Window/Open Memory Window menu items. These menu items allow you to open a **Code** or **Memory** window at the associated address.

Set Breakpoint menu item. This menu item allows you to set a breakpoint in the **Code** window at the address of a selected symbol without having to open the **Breakpoints** window to do so.

Go Until Cursor menu item. Enabling this menu item causes SourcePoint to run from a node to the point where you have placed your cursor.

Hexadecimal Display menu item. This menu item is enabled by default and causes the values displayed to be listed in hexadecimal.

Show Names/ShowTypes/Show Values/Show Return Type menu items. These menu items are enabled if you are in the **Stack** tab. They cause SourcePoint to display names, types, and values of functions in the **Symbols** window..

Show Source Location menu item. This menu item is available only if you are connected to an IDE.

Expand Children/Collapse Children menu items. Enabling one of these menu items expands/collapses selected nodes in the symbol tree.

Viewpoint menu item. This menu item indicates the status of the processor viewpoint. If you have enabled one of the processor options, that processor is tracked. If you have enabled the **Track Viewpoint** option, the current processor is tracked.

Refresh menu item. This menu item refreshes the current view.

Properties menu item. Enabling this menu item opens a message box that contains information such as **Name**, **Mangled Name**, **Type**, **Address**, **Length**, **Scope**, and **Program**. Different symbols have different properties.

Classes Tab

The **Classes** tab lists structure and class definitions

What you see in the **Classes** tab may depend on whether you have left the **Smart Symbol Analysis** option enabled (the default) in the **Program** tab of the **Preferences** dialog box. (See the topic, "[Options Menu - Preferences Menu Item](#)" in "SourcePoint Overview" under *SourcePoint Environment* for more details.) If so, the **Classes** tab shows only those classes that have already been discovered.

You may choose to disable the **Smart Symbol Analysis** option in order to view all symbols. However, it may take considerable time for the symbols to load. Alternatively, if you want to see a particular class or structure, go to the **Globals** tab and expand the module where it is declared. The module usually has the same name as the file.

Note: The tab works properly only for files with Dwarf2 symbols.

Globals Tab

The **Globals** tab displays a hierarchy of loaded programs. Programs can be expanded to show modules, procedures, and symbols.

Columns

There are up to four columns displayed in the Files view: **Name**, **Address**, **Type**, and **Value**, depending on how expanded an entry is. The **Name** column displays program, module, procedure, and symbol names. The **Address** and **Type** columns display symbol addresses and data types. The **Value** column displays variable values. The **Name** and **Value** columns are always displayed, while the **Address** and **Type** columns can be enabled or disabled via the context menu.

Programs

Each program can be expanded in the **Globals** folder to show the modules it contains. A **Code** or **Memory** window, showing the starting point of a program, can be opened by selecting either the **Open Code Window** or **Open Memory Window** menu item from the context menu. Programs can be removed from SourcePoint by selecting either **Remove Program** or **Remove All Programs** from the context menu.

Note: If a module does not contain any data variables, the + disappears from in front of the **Data** folder the first time it is expanded.

Note: Currently, values are not available for program global variables.

Modules

Each module can be expanded to show the procedures it contains. Module bitmaps are colored yellow to indicate that source line information is available. A **Code** or **Memory** window, showing the first procedure in the module, can be opened by selecting either **Open Code Window** or **Open Memory Window** from the context menu. To set a breakpoint at the first procedure in the module, select **Set Breakpoint** from the context menu.

Expanding the **Data** folder of a program displays all the global variables defined in the program. Expanding the **Data** folder for a module displays the variables defined within that module.

Note: To speed program load, symbol information is not completely processed until requested. For very large programs (programs with a lot of symbols), opening the **Data** folder for a program may take a while. Opening the **Data** folder for a particular module is usually faster.

Note: if a module doesn't contain any global variables, the + disappears from in front of the **Data** folder the first time it is expanded.

Procedures

A **Code** or **Memory** window, showing the procedure, can be opened by selecting either **Open Code Window** or **Open Memory Window** from the context menu. To set a breakpoint at the entry point of the procedure, select **Set Breakpoint** from the context menu. Selecting **Go Until Cursor** from the context menu causes the processor to run until the procedure is executed.

Symbols

Symbols include variables and labels. Variables have editable values, while labels do not. Composite variables, including arrays, structures, and unions, are expandable to show their sub-elements. The **Address** and **Type** columns are not visible by default but can be enabled via the context menu. (You must have selected a symbol for these items to be available in the context menu). Alternatively, the address and data type of a symbol can be viewed either via the flyover tooltips or by selecting **Properties** from the context menu. Variable values can be edited by double-clicking the left mouse button or by selecting **Edit** from the context menu. Variable values are normally colored black. If a variable value changes, either by running or stepping the processor or by editing its value directly, then the value is colored green to indicate the change.

Locals Tab

The **Locals** tab shows the local variables accessible in the current stack frame, including procedure arguments and automatics. Composite variables, including arrays, structures, and unions, are expandable to show their sub-elements.

Columns

The **Locals** tab has up to four columns displaying variable names, addresses, data types, and values. The **Name** and **Value** columns are always displayed. The **Address** and **Type** columns are disabled by default but can be enabled via the context menu. Alternatively, the address and data type of a symbol can be viewed via the flyover tooltips or by selecting **Properties** from the context menu.

Editing

Variable values can be edited by double-clicking the left mouse button or by selecting **Edit** from the context menu. Variable values are normally colored black. If a variable value changes, either by running or stepping the processor, or by editing its value directly, then the value is colored green to indicate the change. A variable can be copied to a **Watch** window by selecting **Copy To Watch** from the context menu.

For more information about the Watch window, see "[Watch Window Introduction](#)" part of "Watch Window Overview," found under *Watch Window*.

Sorting

The variables in the **Locals** tab can be sorted by name, address, or data type by left clicking in the appropriate column heading. Click once to sort in ascending order, again to re-sort in descending order.

Multi-processor

In multi-processor systems the **Locals** tab is always associated with the current viewpoint processor.

Stack Tab

The **Stack** tab shows the program's current call stack. Stack frames can be expanded to show local variables, including procedure arguments and automatics. Composite variables, including arrays, structures, and unions, are expandable to show their sub-elements.

Columns

The **Stack** tab has up to four columns displaying names, addresses, data types and values. The **Name** and **Value** columns are always displayed. The **Address** and **Type** columns are disabled by default but can be enabled via the context menu. Alternatively, the address and data type of a variable can be viewed via the flyover tooltips, or by selecting **Properties** from the context menu.

Stack Frames

Each frame shows the name of the procedure called. Argument names, data types, and values can be selectively displayed via the context menu. A **Code** or **Memory** window showing the procedure can be opened by selecting either **Open Code Window** or **Open Memory Window** from the context menu. To set a breakpoint at the entry point of the procedure, select **Set Breakpoint** from the context menu. Selecting **Go Until Cursor** from the context menu causes the processor to run until the procedure is executed.

How To - Symbols Window

How to Change Values in the Symbols Window

Note: Entire fields cannot be selected when changing symbol values. The cursor control keys or the mouse must be used to position the blinking caret immediately before the variable or register digits can be changed.

1. Right-click on the value you want to change (or select **Edit** via the context-sensitive menu).
2. Use the mouse or cursor control keys to position the blinking caret immediately before the digit or series of digits to be changed.
3. Enter the new value for each digit, as required. Use the cursor control keys to skip over unchanged digits.
4. To effect the value changes, press the Enter key or click your mouse on another field or window.

The color of all digits in the field changes from black to green.

Note: This method replaces the entire symbols content.

Trace Window

Trace Window Overview

Trace Window Introduction

The Trace window displays a history of executed instructions.

Intel processors have two methods of recording trace data: last branch record MSRs (LBRs), and the BTS. Both methods record Branch Trace Messages (BTMs). A BTM is recorded whenever the processor executes an instruction that causes a branch, or when an interrupt or exception occurs. SourcePoint uses the BTMs to display disassembled trace data. Both of these methods provide instruction trace only (no data tracing). Neither method supports any kind of timestamp capability.

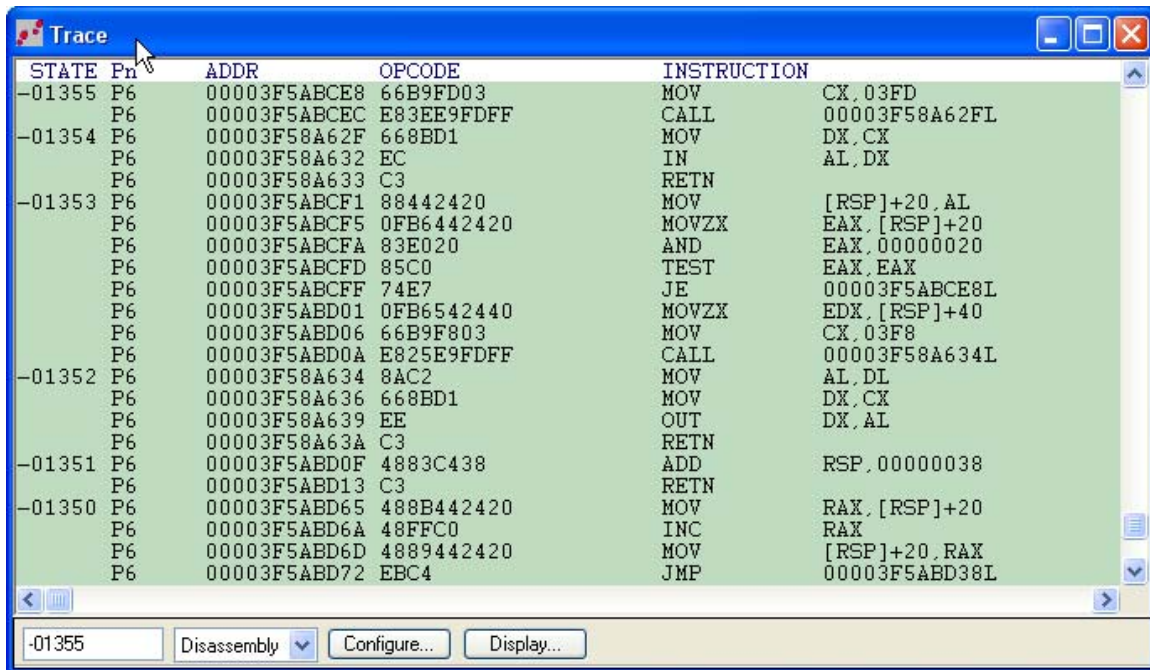
The advantage of LBR trace is it is nonintrusive. The processor can run at full speed when using LBR trace. The disadvantage of LBR trace is the limited number of LBRs available (typically 4 - 16). Each LBR stores a single BTM. If you assume an average of 5 instructions between branches, then roughly the last 80 instructions executed are traced.

The BTS is an area of memory set aside for storing BTMs. The BTS can be much larger, and store many more BTMs. The disadvantage of using the BTS is that writing BTMs to memory takes a lot longer than writing BTMs to LBRs. Each branch results in 12-24 bytes of memory being written. For some applications BTS trace may result in too high a speed penalty to use. Another disadvantage of BTS trace is the inability to trace out of reset (if memory is unavailable).

BTS trace and LBR trace are not mutually exclusive. They can both be enabled at the same time. Since there is no speed penalty to leaving LBR trace enabled all of the time, there is no UI for disabling it.

The Trace Window

To open the Trace window, select View|Trace on the menu bar or click on the Trace icon on the toolbar.



Trace window (Disassembly mode)

There are three display modes: State, Mixed, and Disassembly. The normal display mode is Disassembly. State and Mixed are used primarily for troubleshooting trace capture problems.

Disassembly Mode. This mode shows disassembled instructions. This is the default display mode. Source code information can also be displayed. See the section on the [Trace Display Settings](#) dialog box for a list of information that can be displayed. The following fields are displayed in Disassembly mode:

- **State.** Displays the state number (cycle number) of the instruction. The trigger location is marked as 0. Cycles before the trigger are negative numbers, while cycles after the trigger are positive numbers.
- **Pn.** Displays the trace source (the processor that generated the trace).
- **Addr.** Displays the address of the instruction.
- **Opcode.** Displays the instruction opcode bytes. Up to 10 bytes are shown. If the instruction is longer than 10 bytes, then the tooltip can be used to display all of the instruction bytes.
- **Instruction.** Displays the disassembled instruction (mnemonic and operands).

State Mode. State mode shows raw trace data. This mode is used primarily for troubleshooting hardware or disassembly problems. The following fields are displayed:

- **State.** Displays the state number (cycle number) of the instruction. The trigger location is marked as 0. Cycles before the trigger are negative numbers, while cycles after the trigger are positive numbers.
- **Pn.** Displays the trace source (the processor that generated the trace).
- **From-H.** Upper 32 bits of BTM source address
- **From-L.** Lower 32 bits of BTM source address
- **To-H.** Upper 32 bits of BTM destination address
- **To-L.** Lower 32 bits of BTM destination address

Mixed Mode. Displays both state and disassembly data.

Trace Window Dialog Bar

The dialog bar at the bottom of the Trace window contains shortcuts to often-used features. Display of the dialog bar is optional. It can be turned off via the View menu. All functionality in the dialog bar is also available in the Trace window menu.

Cycle Number

The current cycle number (the location of the caret in the Trace window) is shown at the far left of the dialog bar. The trigger location is always marked as cycle 0. Cycles before the trigger are negative numbers, while cycles after the trigger are positive numbers. Enter a number in this field to jump to a particular location in trace.

Display Mode

This drop-down list lets you choose the display mode. There are three modes: Disassembly, State, and Mixed. For more information, see above.

Configure Button

The **Configure** button opens the [Trace Configuration](#) dialog box. This dialog is used to configure trace capture settings for the BTS.

Display Button

The **Display** button opens the [Trace Display Settings](#) dialog box. This dialog is used to configure the display format of the Trace window.

Trace Window Menu

The Trace menu displays on the menu bar after you have opened a Trace window. It is also displayed when right-clicking in the Trace window.

State, Disassembly, Mixed. Selects the current display mode. Provides the same functionality as the Display Mode drop down list in the dialog bar.

Open Code Window. Opens a tracking Code window showing the instruction at the current caret position. Scrolling through trace causes the tracking window to scroll automatically, so the correct source is always displayed. This is an alternate method for displaying source code. The other method is to display source directly in the **Trace** window.

Open Memory Window. Opens a Memory window at the address indicated by the current caret position.

Set Breakpoint. Sets a code breakpoint at the address indicated by the caret. The default breakpoint type is specified in the Preferences dialog box (Options | Preferences | Breakpoints). The choices are Hardware Breakpoint and Software Breakpoint.

Add/Edit Breakpoint. Opens the Breakpoint window for setting more complex breakpoints.

Goto Cycle. Directly positions the caret at a cycle in trace. This can also be accomplished by entering a new cycle number in the dialog bar.

Configure. Opens the [Trace Configuration](#) dialog box. Provides the same functionality as the Configure button in the dialog bar.

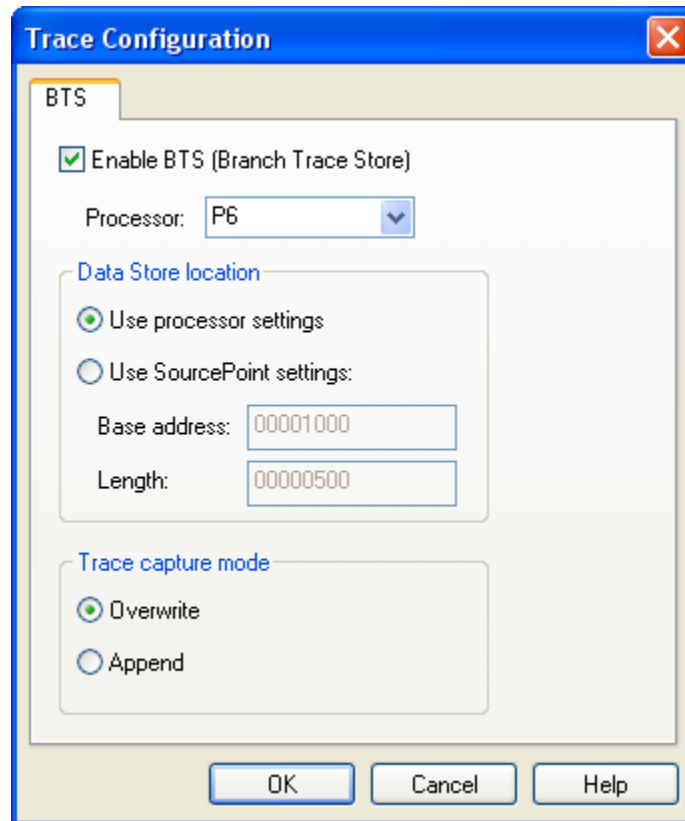
Display. Opens the [Trace Display Settings](#) dialog box. Provides the same functionality as the Display button in the dialog bar.

Disassembly Uses. Indicates whether trace disassembly uses target memory or a cached program. The default is Cached Program, which is much faster. Target memory is used if no program has been loaded or if the program does not contain the required memory for an instruction.

Copy Menu Item. Copies the selected text to the clipboard (CTRL+C).

Trace Configuration

The **Configure** button in the **Trace** window opens the Trace Configuration dialog box. This dialog is used to configure trace capture settings for the BTS. Changes in this dialog take effect on the next trace capture.



Trace Configuration dialog box

Enable BTS. This checkbox enables the BTS. The default setting is disabled.

Processor. This dropdown list selects which processor to trace. The names displayed are the same processor names defined in the Target Configuration dialog.

Data Store Location. When “Use processor settings” is selected, SourcePoint takes care of controlling the BTS (starting and stopping it), but does not define the location of the BTS. This is useful when an operating system (or EFI code) is in charge of allocating space for the BTS.

When “Use SourcePoint settings” is selected, SourcePoint not only takes care of controlling the BTS, but also creates the BTS.

Note: The BTS is actually one component of a larger memory structure called the Data Store.

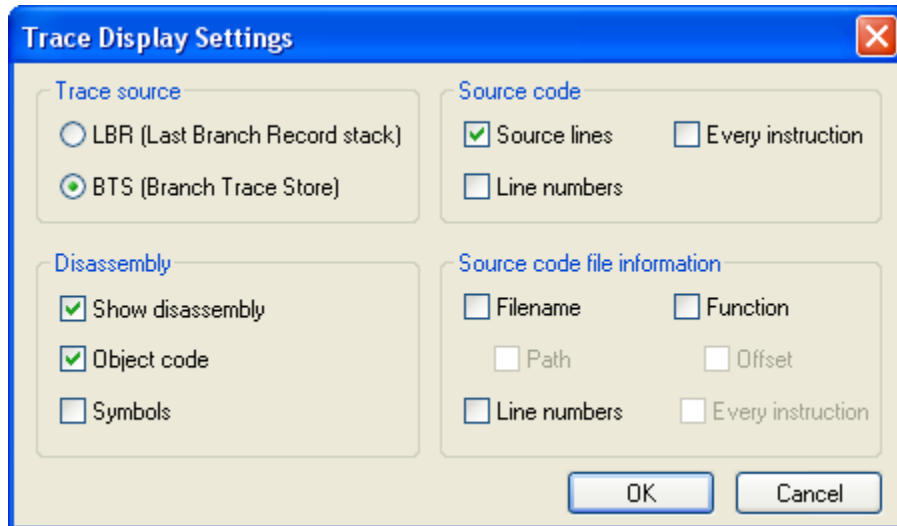
The Base address field specifies the linear base address of the Data Store. This field is enabled when Use SourcePoint settings is enabled.

The Length field specifies the length of the Data Store. This field is enabled when Use SourcePoint settings is enabled. The actual size of the BTS is the length of the Data Store minus 0x60 bytes.

Trace Capture Mode. When “overwrite” is selected, SourcePoint clears the BTS prior to each data capture. When “append” is selected, new trace data is appended to existing trace data. The default setting is overwrite,

Trace Display Settings

The **Display** button in the **Trace** window opens the Trace Display Settings dialog box. This dialog is used to configure the display format of the **Trace** window.



Trace Display Settings dialog box

When LBR is selected, the Trace view displays trace from the LBR MSRs. When BTS is selected, SourcePoint displays trace from the BTS. This selection can be changed without collecting new trace data. The default is LBR trace.

Disassembly section

Show disassembly. When selected, disassembled instructions are shown in the Trace view. Disabling disassembly allows the user to display only Source code. The default is enabled.

Object code. When selected, adds object code bytes prior to the disassembled instruction display. The default is disabled.

Symbols. When selected, replaces numeric address operands with symbolic addresses in the disassembled instruction display. The default is disabled.

Source Code section

These checkboxes control the display of source lines in the Trace view.

Source lines. When selected, enables the display of source code in the Trace view. The default is disabled.

Line numbers. When selected, the line number of the source line is also displayed. The default is disabled.

Every instruction. When selected, source lines are displayed for every disassembled instruction rather than the first instruction of a group (the first instruction after a branch). The default is disabled.

Source Code File Information section

These controls allow the display of source file information corresponding to the disassembled trace data.

Filename. When selected, the source filename is displayed. The default is disabled.

Path. When selected, the full path of the source filename is displayed. The default is disabled.

Line numbers. When selected, line numbers are displayed. The default is disabled.

Function. When selected, function names are displayed. The default is disabled.

Offset. When selected, function offsets are displayed. The default is disabled.

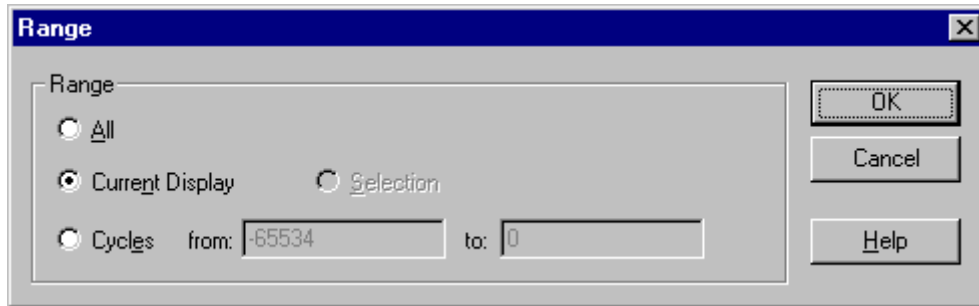
Every instruction. When selected, source file information is displayed for every line of disassembly. The default is disabled.

How To - Trace Window

How to Print Trace

1. Go to **File|Print** on the menu bar to print the current data from the **Trace** window.

The **Range** dialog box displays.



*Print **Range** dialog box*

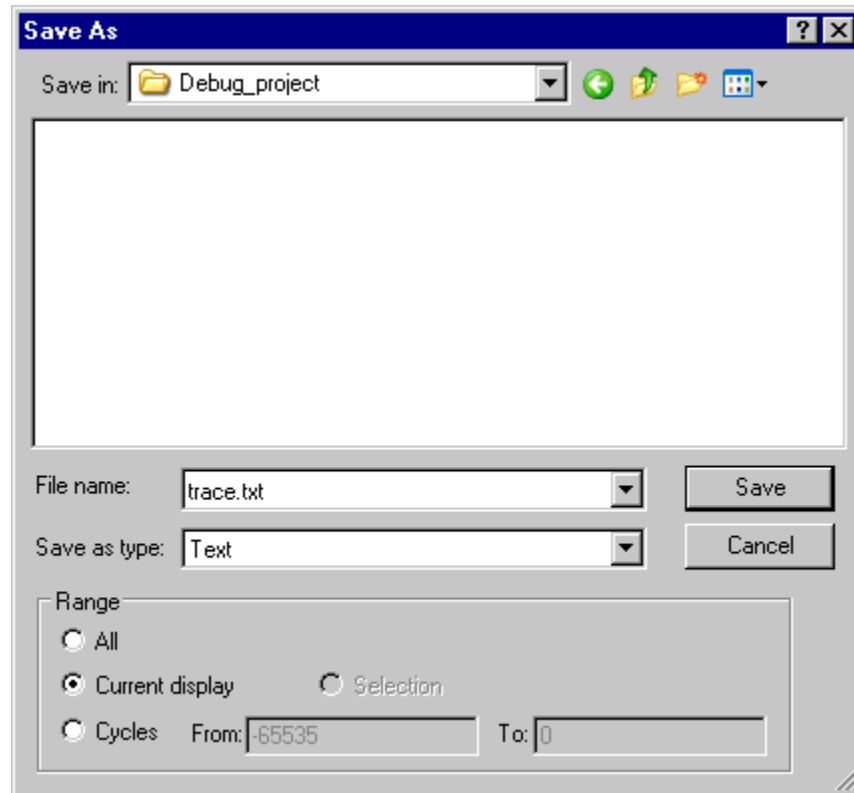
2. Choose the range or cycles to print.
3. Click the **OK** button.

Note: Selecting **All** saves the entire trace buffer. This can take several minutes.

How to Save Trace

1. With the **Trace** window active, select **File|Save As** on the menu bar.

The **Save As** dialog box displays.



Save As dialog box

2. Choose a file name and location to save to disk.
3. In the **Range** section, select the range to save.

You can choose to save the current display, a range of cycles, a highlighted selection, or the entire trace buffer (this can take several minutes).

4. Click on the **Save** button.

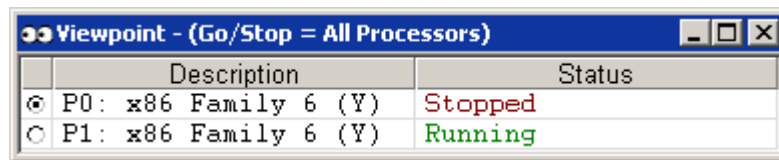
Viewpoint Window

Viewpoint Window Overview

Viewpoint Window Introduction

The **Viewpoint** window is available when the emulator is connected to a target with multiple processors. It allows you to view each processor and select the viewpoint processor, the processor whose state is reflected in SourcePoint. On targets with AMD processors, you can also **Go**, **Stop**, or **Step Into** an individual processor or all processors as a whole.

The **Viewpoint** window expands and contracts as the window is resized, making the window easy to dock. Columns resize automatically as the window is resized. Processor entries are dynamically added and removed from this window as the operating environment changes. If the window is opened before the target has been acquired, it is empty. Once the number of target processors is determined, then a row is added for each one. The rows are always sorted by description.



	Description	Status
<input checked="" type="radio"/>	P0: x86 Family 6 (Y)	Stopped
<input type="radio"/>	P1: x86 Family 6 (Y)	Running

Viewpoint window with individual processor control

First column. The first column contains radio buttons to select the current viewpoint. If a processor is unavailable for selection, the entire row, including the radio button, is grayed out.

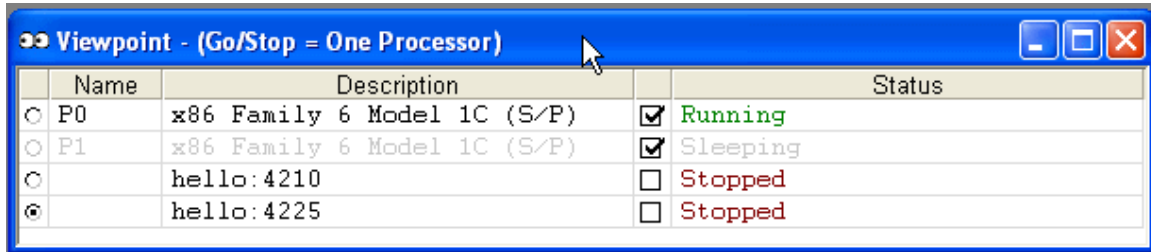
Description column. The second column displays a description of the processor. The description format is of the form: "Pn: ProcDescr", where 'n' is the viewpoint value and 'ProcDescr' is a string describing the processor type.

Status column. The third column displays processor status. The options are Running, Stopped or Sleeping. If a processor is currently stopped due to hitting a breakpoint, then its status is appended with "(hit breakpoint)".

Go, Stop, Step Into columns. These columns contain the buttons to control the running state of each processor.

Linux Task Debugging

If you are using SourcePoint for Linux task debugging, the Viewpoint window displays all current Linux tasks in addition to processors.



Name	Description	Status
<input type="radio"/> P0	x86 Family 6 Model 1C (S/P)	<input checked="" type="checkbox"/> Running
<input type="radio"/> P1	x86 Family 6 Model 1C (S/P)	<input checked="" type="checkbox"/> Sleeping
<input type="radio"/> hello:4210		<input type="checkbox"/> Stopped
<input checked="" type="radio"/> hello:4225		<input type="checkbox"/> Stopped

Viewpoint window with Linux tasks

For Linux tasks, the Description column displays the task name and process ID separated by a colon. Only the task filename is displayed. The full path to the task is available via a tool tip.

When a Linux task is started or attached to an entry for that task, it is automatically added to the Viewpoint window. After the task exits, the entry is automatically removed. If the current viewpoint is a task and that task terminates, then another stopped viewpoint is selected automatically, choosing an available task in priority over a processor.

Note: It is not always possible to change the viewpoint to any of those in the window. For example, if you are Linux debugging and switch to a processor viewpoint and stop it, you are not able to switch back to the task viewpoint. This is because task viewpoints are not available when the processor viewpoint that controls them is stopped. In these cases, the unavailable viewpoints are grayed out and the Set Viewpoint menu item is disabled.

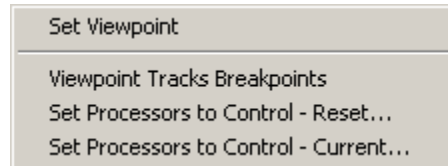
Although processors can be controlled individually, or controlled as a whole, tasks are always independently controlled.

Viewpoint Window Menu

SourcePoint displays one of two menus, depending on whether you are connected to an Intel processor or an AMD processor. The context menu allows you to access all controls and settings relevant to the **Viewpoint** window. To bring up the **Viewpoint** window context menu, right click on the window.

Menu for Intel Processors

When you are connected to an Intel-processor target, two menu items are available.



Viewpoint context menu (Intel processors)

Set Viewpoint menu item. This menu item is used to set the current viewpoint processor. It performs the same function as clicking on the radio button in the first column.

Viewpoint Tracks Breakpoints menu item. When this menu item is enabled, the viewpoint processor switches automatically to whichever processor has hit a breakpoint.

Set Processors to Control - Reset menu item. This menu item pops up the **Emulator Configuration** dialog box and allows you to select the processors you want the emulator to control at target reset.

Note: For additional information, see the topic, "[Disabling Processors](#)," part of "Viewpoint Window Overview," found under *Viewpoint Window*.

Set Processors to Control - Current menu item. This menu item pops up a dialog that allows you to select the processors you want the emulator to control now.

Note: The **Go** and **Stop** buttons in a given row control only that row's processor if **Go/Stop Controls One Processor** is checked on the context menu. You can go or stop any individual processor, even if it is not the current viewpoint processor. However, if **Go/Stop Controls All Processors** is checked, clicking on a **Go** or **Stop** button in any row will cause all processors to go or stop. These two settings are mutually exclusive and only affect the **Go** and **Stop** buttons. For convenience, the **Viewpoint** window's title bar always displays the current state of this setting.

Note: The **Go**, **Stop**, and **Step Into** buttons of a given row control only that row's processor if **Go/Stop Control One Processor** is enabled on the context menu. You can run, step, or stop any individual processor, even if it is not the viewpoint processor. However, if **Go/Stop Control All Processors** is enabled, clicking on a **Go**, **Stop**, or **Step Into** button in any row will cause all processors to run, stop, or step. These two settings are mutually exclusive.

Menu for AMD Processors

When you are connected to an AMD-processor target, this menu displays.

Viewpoint context menu (AMD processors)

Set Viewpoint menu item. This menu item is used to set the current viewpoint processor. It performs the same function as clicking on the radio button in the first column.

Go menu item. This menu item is used to cause a processor to run. It performs the same function as clicking on the **Go** button on the right side of the **Viewpoint** window.

Stop menu item. This menu item is used to cause a processor to stop. It performs the same function as clicking on the **Stop** button on the right side of the **Viewpoint** window.

Step Into menu item. This menu item can be used to step into a processor. It performs the same function as clicking on the **Step Into** button on the right side of the **Viewpoint** window.

Go/Stop Control All Processors menu item. When this menu item is enabled, clicking the **Go**, **Stop**, or **Step Into** button of any processor causes all processors to go, stop, or step into.

Go/Stop Control One Processor menu item. When this menu item is enabled, clicking the **Go**, **Stop**, or **Step Into** button of any processor causes only that processor to **Go**, **Stop**, or **Step Into**.

Viewpoint Tracks Breakpoints menu item. When this menu item is enabled, the viewpoint processor switches automatically to whichever processor has hit a breakpoint.

Set Processors to Control - Reset menu item. This menu item pops up the **Emulator Configuration** dialog box and allows you to select the processors you want the emulator to control at target reset.

Note: For additional information, see the topic, "[Disabling Processors](#)," part of "Viewpoint Window Overview," found under *Viewpoint Window*.

Set Processors to Control - Current menu item. This menu item pops up a dialog that allows you to select the processors you want the emulator to control now.

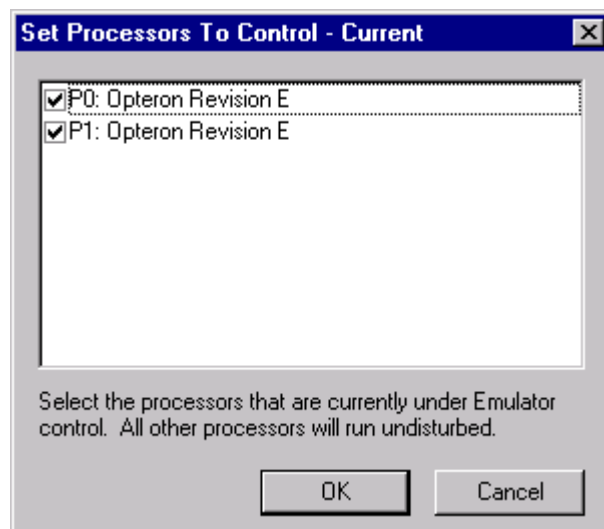
Note: The **Go** and **Stop** buttons in a given row control only that row's processor if **Go/Stop Controls One Processor** is checked on the context menu. You can go or stop any individual processor, even if it is not the current viewpoint processor. However, if **Go/Stop Controls All Processors** is checked, clicking on a **Go** or **Stop** button in any row will cause all processors to go or stop. These two settings are mutually exclusive and only affect the **Go** and **Stop** buttons. For convenience, the **Viewpoint** window's title bar always displays the current state of this setting.

Note: The **Go**, **Stop**, and **Step Into** buttons of a given row control only that row's processor if **Go/Stop Control One Processor** is enabled on the context menu. You can run, step, or stop any individual processor, even if it is not the viewpoint processor. However, if **Go/Stop Control All Processors** is enabled, clicking on a **Go**, **Stop**, or **Step Into** button in any row will cause all processors to run, stop, or step. These two settings are mutually exclusive.

Disabling Processors

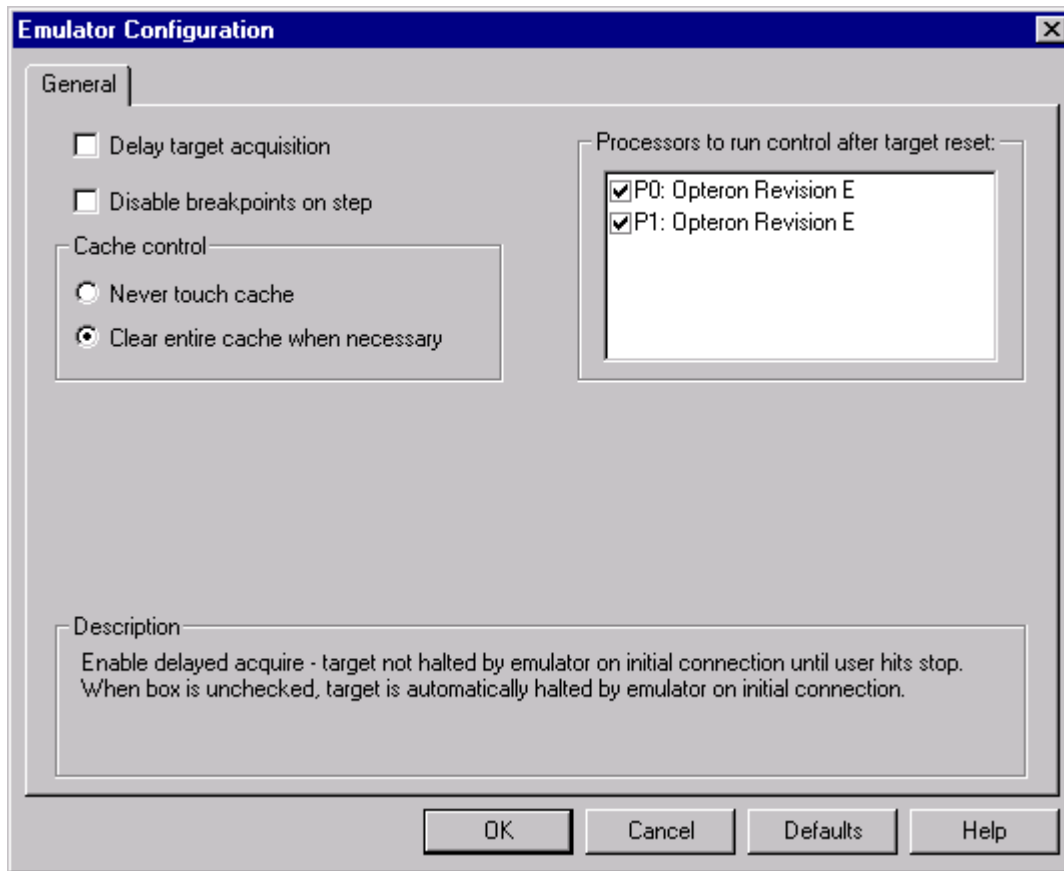
Processors can be "masked" such that they are not under emulator control. You may select the processors that the emulator currently controls using the **ProcessorControl** control variable bit mask in the **Command** window. A processor that is not available because of the ProcessorControl mask has its status appended with "(not controlled)" and is grayed out in the **Viewpoint** window. See the [ProcessorControl](#) command help topic for further information.

Alternatively, there is a dialog box that lets you easily set the ProcessorControl mask. To specify the processors that the emulator currently controls, right click on the **Viewpoint** window to open the context menu. Click on the **Set Processors to Control - Current** menu item. This brings up the **Processors to Control** dialog box, which lists all processors. Place a check mark in the box to the left of each processor that the emulator should currently control. Processors that are unchecked here will be shown as "not controlled" and grayed out in the **Viewpoint** window.



Set Processor To Control - Current dialog box

In multiprocessor systems, it is often necessary to not allow the emulator to control some processors at reset and leave them untouched until code has run to sufficiently configure the target hardware. After that point, you need to allow the emulator to control these processors and perform your debugging. The dialog box described below allows you to accomplish this.



Emulator Configuration - General Tab (as seen if opened via **Viewpoint** context menu)

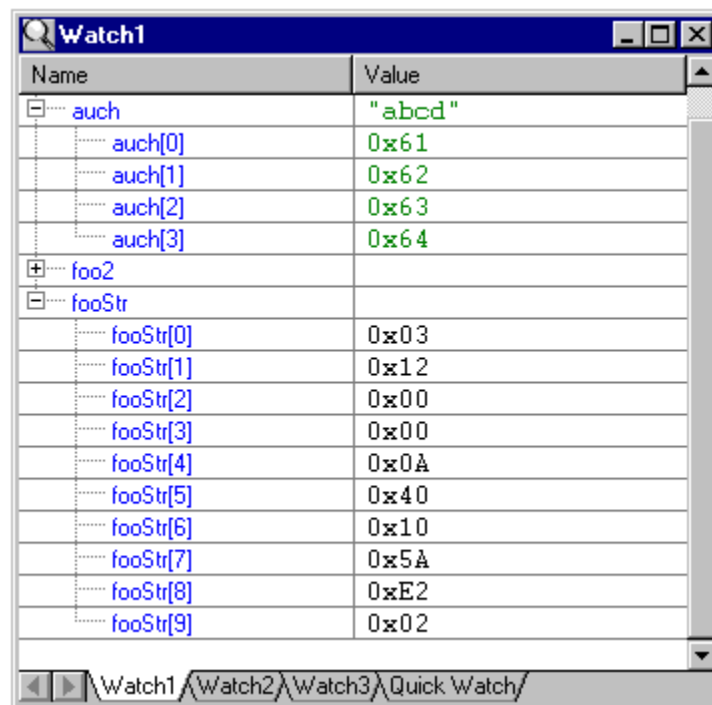
You include and exclude processors from emulator control at target reset in the **General** tab on the **Emulator Configuration** dialog box. To select the processors the emulator controls at target reset, right click on the **Viewpoint** window to open the context menu. **Click on the Set Processors to Control - Reset** menu item to bring up the **General** tab on the **Emulator Configuration** dialog box. The list box on the right side lists all the available processors. To select a processor to be under emulator control at target reset, place a check mark in the box to the left of the processor name. Processors that are unchecked here are not acquired by the emulator when the target resets and are shown as "not controlled" and grayed out in the **Viewpoint** window. These settings are applied to the ProcessorControl mask at each target reset.

Watch Window

Watch Window Overview

Watch Window Introduction

The **Watch** window is designed to allow you to more easily "watch" those variables, registers, and expressions you want to view often, especially as they change value, by copying them into a **Watch** or **Quick Watch** tab in the **Watch** window. This saves you from having to scroll through a **Symbols** window tab to view them. Composite variables, including arrays, structures, and unions, are expandable to show their sub-elements.



*Watch window showing data in the **Quick Watch** tab view*

Watch Tabs

The **Watch** tabs are designed to hold user-specified variables, registers, and expressions whose values are re-evaluated each time the processor stops or is stepped.

Note: In Monitor mode, the target is read without being stopped. In this mode, values do not update automatically. Use the **Refresh** menu item in the context menu to update values in Monitor mode.

Quick Watch Tab

The **Quick Watch** tab is identical to the other **Watch** tabs except that its settings are not saved and its contents are cleared on every halt or stop event. This is of use if you want to see a value

only once and it is a complex value (simple variables can be seen in the flyover). **Quick Watch** entries do not clutter your **Watch** tabs with extra variables.

General Features

Columns

The **Watch** or **Quick Watch** view displays up to four columns showing names, addresses, data types, and values. The **Name** and **Value** columns are displayed by default. The **Address** and **Type** columns can be enabled via the context menu. Alternatively, the address and data type of a symbol can be viewed via the flyover tooltips, or by selecting **Properties** from the context menu.

Values

Variable and register values normally are colored black. If a value changes, either by running or stepping the processor or by editing its value directly, then the value is colored green to indicate the change. In addition, register values are colored gray to indicate a read-only register or are displayed as asterisks to indicate write-only registers.

Values can be displayed in either decimal or hexadecimal. Select **Hexadecimal** in the context menu to toggle between the two. Regardless of the display base, addresses are always displayed in hexadecimal. In addition, values larger than 32 bits are always displayed in hexadecimal.

Adding Watches

Watches can be added to a **Watch** or **Quick Watch** view in several ways. To directly enter a variable name, register name, or expression, select **Add Watch** from the context menu. Drag and drop can also be used to add watches. Names and/or expressions can be dragged in from a **Symbols** window or from the **Code**, **Trace**, or **Command** windows. Register names can be dragged in from a **Register** window. The variables in a **Watch** or **Quick Watch** view can be reordered by using drag and drop to reposition a watch in the list.

Editing Values

Variable and register values can be edited by double-clicking the left mouse button or by selecting **Edit** from the context menu. To end editing a value, press the Enter key or select another field. If a value is being edited when either **Go** or **Step** is selected, the edit is terminated, the value is written, and then the **Go** or **Step** operation is performed. Values can be copied and pasted by using the **Edit** menu menu items, by using Ctrl-C/Ctrl-V, or by using drag and drop.

ToolTips

Most items have flyover tooltips that display some of the information available in the **Properties** dialog box, available via the context menu.

Keyboard Support

The arrow keys provide keyboard support for navigation through a tree:

- The Up Arrow and Down Arrow keys move between items.

- The Left arrow and Right arrow keys move along a particular branch. Pressing the Right arrow expands a branch if it is not currently displayed. Pressing the Left arrow moves the cursor to the first item in a branch; pressing it a second time collapses the branch.
- The Home and End keys move to the top or bottom of the tree.
- The Page Up and Page Down keys move a page at a time.
- The + and - keys expand and collapse the current tree node.
- The Enter key alternately expands and collapses the current note.
- The use of the asterisk (Shift and the number 8 on the keyboard) expands all tree nodes beneath the currently selected note.

Printing

To print a view, select **File|Print** on the menu bar. The entire tree is printed, but only currently expanded nodes are included.

Colors

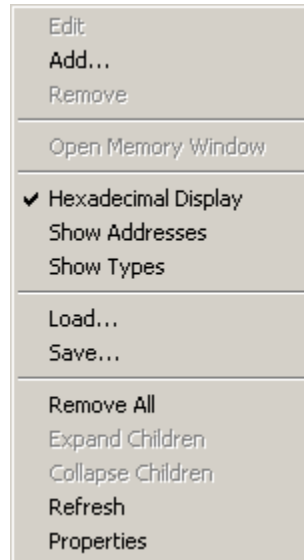
Colors can be changed via the **Colors** tab under **Options|Preferences**.

Multi-processor

In multi-processor systems, register values and stack-relative variables are always associated with the current viewpoint processor.

Watch Window Menu

A context menu can be accessed by right-clicking on a variable.



Watch/Quick Watch window context menu

Edit menu item. The **Edit** menu item lets you edit values. Variable and register values can be edited by double-clicking the left mouse button or by selecting **Edit** from the context menu. Expression values are not editable. Watch names can also be edited.

Add menu item. The **Add** menu item opens an **Add Watch** dialog box into which you can put a the name of a variable, expression, or register or browse for one. Once the dialog closes, the name displays automatically in a **Watch** or **Quick Watch** view (depending on which tab you have chosen), along with its value. If the **Address** and **Type** fields are enabled, data display in those columns, too.

Remove menu item. This menu item removes a highlighted line.

Open Memory Window menu item. Enabling this menu item causes the **Memory** window to open at the specified address listed in the **Watch** or **Quick Watch** tab.

Hexadecimal Display menu item. Select **Hexadecimal** to toggle between decimal and hexadecimal. Regardless of the display base, addresses are always displayed in hexadecimal. In addition, values larger than 32 bits are always displayed in hexadecimal.

Show Addresses menu item. The **Show Addresses** menu item displays the **Addresses** column with the variable address in it.

Show Types menu item. The **Show Types** menu item displays the **Type** column with the variable type listed.

Load menu item. You can load a watch or a group of watches you have saved in a ".brk" or ".prj" file by clicking on the **Load** menu item.

Note: Using this command replaces the watches that currently exist in the **Watches** tabs.

Save menu item. Clicking on the **Save** menu item opens a **Save As** dialog box. From there you can create and save the current watch or group of watches in a ".brk" file.

Remove All menu item. This menu item removes all data from a **Watch** or **Quick Watch** tab.

Expand Children menu item. The **Expand Children** menu item expands all composite variables, displaying their sub-elements.

Collapse Children menu item. This menu item causes the window to collapse all composite variables that have been expanded to show their sub-elements..

Refresh menu item. Use this menu item if you are running in Monitor mode and want to refresh your values.

Properties menu item. When this menu item is selected, a **Properties** dialog displays. The information varies depending on the type of item selected.

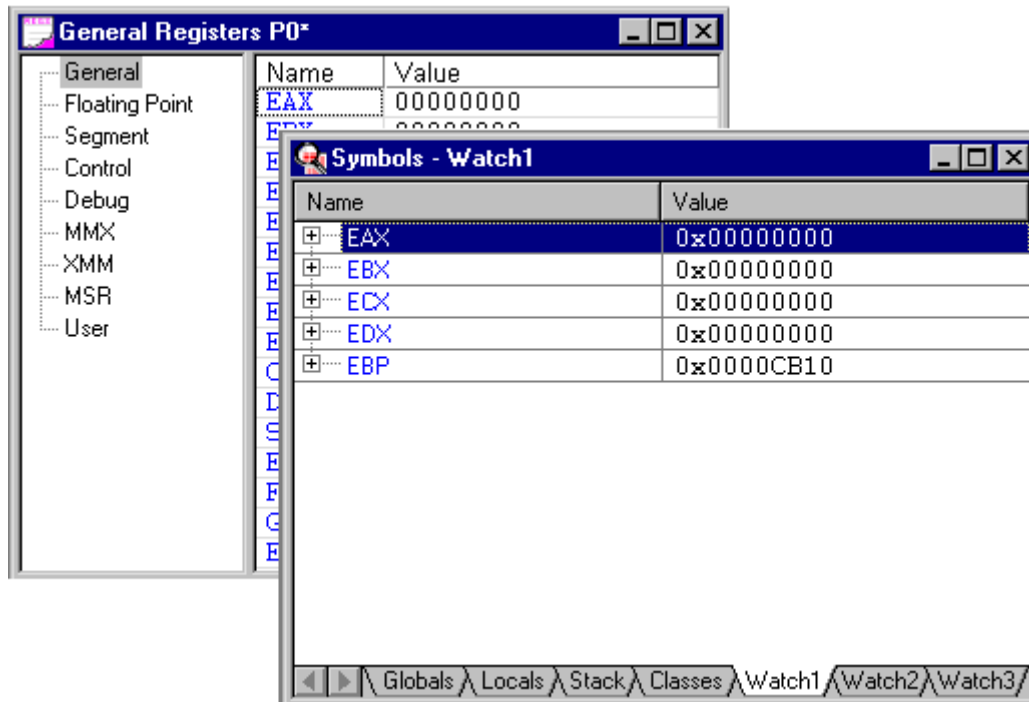
How To - Watch Window

How to Add and Expand Registers in a Watch View

Adding Registers to a Watch View

1. Open the **Symbols** window.
2. Select an empty **Watch** tab.
3. Open the **Registers** window.
4. Move the windows so that you can easily see both views.
5. In the left-hand pane of the **Registers** window, click on the type of register group you want to view. From the right-hand pane, select the register you want to move into the **Watch** view.
6. Click and drag that register into the **Watch** view.

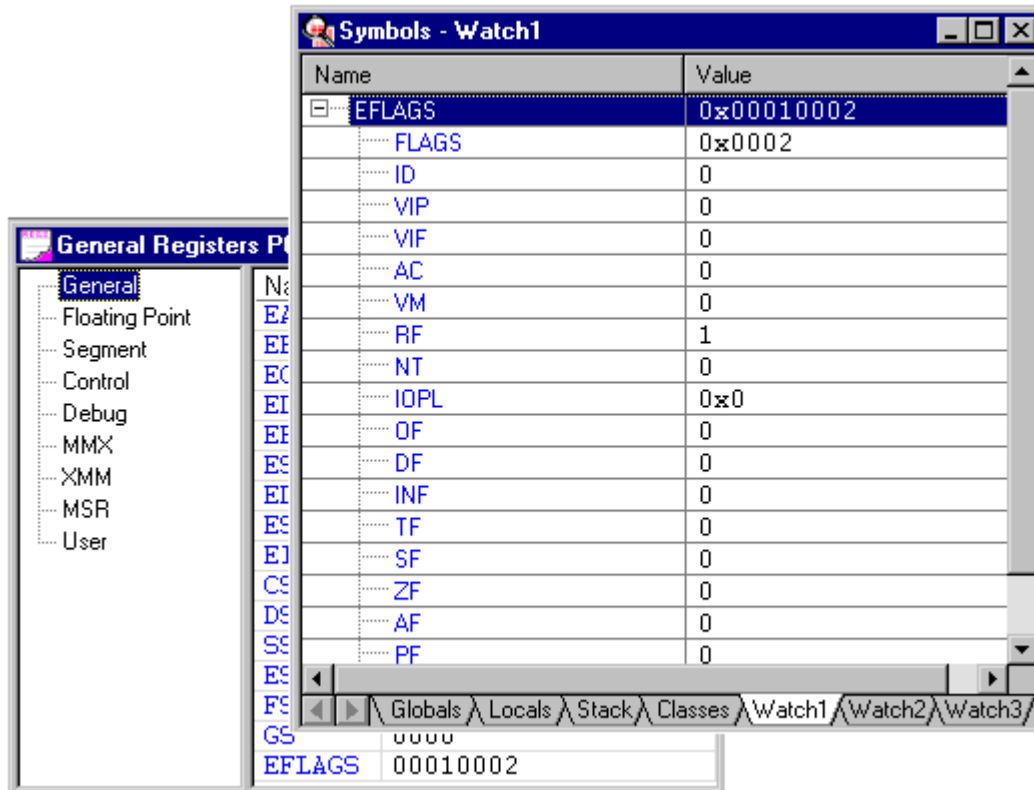
The register and its value move together. The register is fully editable in the **Watch** view. Changes made in the **Watch** view are automatically updated in the **Registers** window.



*Example of some registers dropped into **Watch** view in **Symbols** window*

Expanding Registers in a Watch View

To expand a register in the **Watch** view, double-click on the register.



*EFLAGS register drag into **Watch** view of **Symbols** window and expanded*

How to Add Symbols to a Watch or Quick Watch View

Select **Edit|Find Symbol** from the menu bar.

For more information how to use this dialog box, see, "[Edit Menu](#)," part of "SourcePoint Overview," found under *SourcePoint Environment*. Scroll down to the **Find Symbol** command.

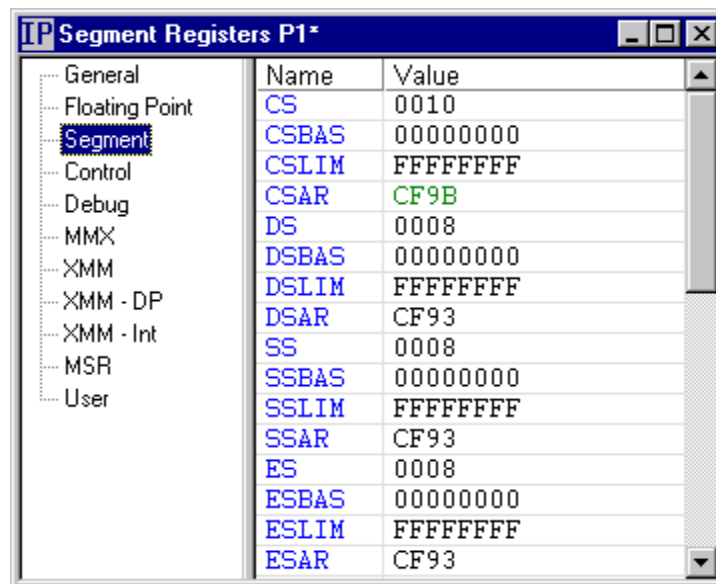
Technical Notes

Descriptor Cache: Revealing Hidden Registers

Many developers are unfamiliar with a very important set of registers that play a crucial role in memory access. These registers are sometimes referred to as "descriptor cache" or "hidden" registers. They are accessible and modifiable only when using an in-circuit emulator such as those produced by Arium. The information below explains how these registers are the true basis for forming linear addresses rather than the segment registers, even in Real mode.

When code execution causes a descriptor table lookup, the processor goes into the descriptor table once to access the descriptor's base, limit, and access rights. A group of three hidden registers linked to each segment register retains this information. The processor does not need to access this table entry again until a segment change is made.

The following figure shows an example of a Segment Registers window from SourcePoint. Note that it displays the descriptor cache for all segment registers. We will discuss the code descriptor cache (i.e., CSBAS, CSLIM, and CSAR), but this information generally applies to all descriptor caches.



	Name	Value
General	CS	0010
Floating Point		
Segment	CSBAS	00000000
	CSLIM	FFFFFFFF
Control	CSAR	CF9B
Debug	DS	0008
MMX	DSBAS	00000000
XMM	DSLIM	FFFFFFFF
XMM - DP	DSAR	CF93
XMM - Int	SS	0008
MSR	SSBAS	00000000
User	SSLIM	FFFFFFFF
	SSAR	CF93
	ES	0008
	ESBAS	00000000
	ESLIM	FFFFFFFF
	ESAR	CF93

Segment Registers window (IA-32 processor)

The linear address where code is accessed is determined by the CSBAS register. CS simply serves to convey the information into CSBAS. For example, if a Real mode program executes a far call that loads a value of F800 into CS, a value of 000F8000 is loaded into CSBAS. The linear address is derived by adding CSBAS to EIP.

These descriptor cache registers also explain why the reset vector is FFFFFFFF0 even though CS is F000 and EIP is 0000FFF0. The reset vector is produced by adding CSBAS (FFFF0000) to EIP (0000FFF0). Since the address is derived in this manner, the reset value in CS has no effect. The CS register is initialized to F000 at reset solely for software compatibility with legacy processors.

When entering Protected mode, system software must perform a far jump that loads CS to reference the appropriate descriptor in the GDT. This causes the processor to access the code descriptor and cache the base, limit, and access rights in CSBAS, CSLIM, and CSAR, respectively. The values remain in these hidden registers until execution changes context by loading another code descriptor.

Modifying a segment register (i.e., a segment selector) manually does not have the same effect as when it is modified by program execution. For instance, CSBAS, CSLIM, and CSAR are not automatically changed when CS is modified using an emulator. In most cases, all of these registers will need to be changed to produce the desired effect.

UEFI Framework Debugging

Overview

The Intel® Platform Innovation Framework for Unified Extensible Firmware Interface (UEFI), commonly known as the UEFI Framework, is a new firmware architecture standard that defines a set of software interfaces and replaces the legacy BIOS found on traditional PC computers. This framework provides the kind of modularity, flexibility, and extensibility that were formerly unavailable with traditional BIOS. With UEFI, BIOS developers can now write all their code in 'C', rather than assembly language. See Intel's website at <http://www.intel.com/technology/framework> or <http://www.tianocore.org> for more information on UEFI Framework.

Along with this new firmware architecture and the 'C' code that implements it comes the need for source-level debugging. Arium's debugger, SourcePoint™ (versions 7.0 and later) for Intel and AMD processors offers native debug support for UEFI Framework platforms. Users can set breakpoints, single step, view variables, see the call stack, and access all of the feature-rich functionality SourcePoint normally provides. This includes source-level debugging during the PEI, DXE, and OS Boot phases of UEFI. Below is a set of instructions for setting up SourcePoint to debug the UEFI Framework.

UEFI Macros

Note: The macros described below are installed in the Macro\UEFI sub-folder of the SourcePoint install path. Several of the UEFI macro files contain directory paths to other macro files. If you move the macro files or change the current working directory in SourcePoint (via the 'cd' command), you will need to update the macros files with the new locations.

EFI.mac

After installing SourcePoint, run the EFI.mac macro file located in the Macro\UEFI directory. This creates six custom toolbar buttons and associates each with a corresponding UEFI proc.

- The StartPEI icon resets the target, then runs to PeiMain and loads the PEI symbols.
- The PEIMs (Pre-UEFI Initialization Modules) icon loads the symbol files for the PEI modules found in target memory.
- The DXEs (Driver Execution Environments) icon loads the symbol files for the DXE modules found in target memory.
- The HOBs (Hand-Off Blocks) icon displays a list of UEFI HOBs found in target memory.
- The SysConfigTable icon displays the contents of the UEFI system configuration table.
- The DumpMemMap icon displays the UEFI Memory Map.



EFI.mac toolbar buttons

PEI Debugging

The PEI environment requires a specialized configuration of SourcePoint. PEI gets control shortly after target reset. PEI modules are dispatched and executed after cache RAM is mapped into system memory and the stack is initialized. Having a stack this early allows 'C' language code to execute, but a special memory map must be configured to take advantage of it.

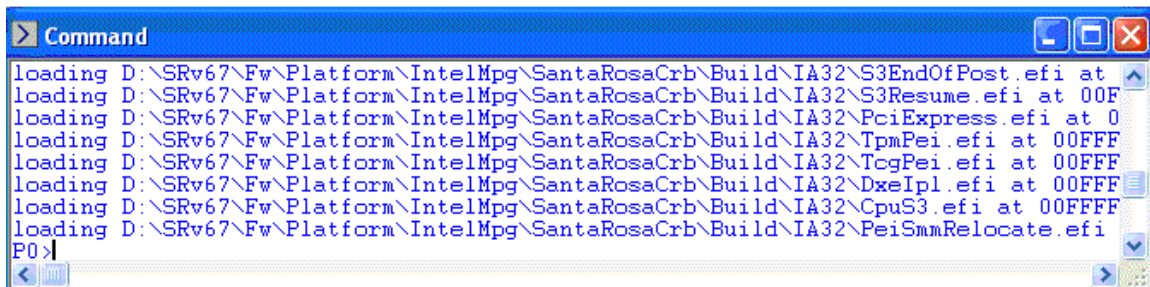
To configure SourcePoint for source-level debugging of PEI code, follow these steps.

1. Optional: Select **Options|Target Configuration|Memory Map** from the menu, and set it similar to the following (your system may vary depending on your memory map):

Start	End	Type
00000000P	000FFFFFFP	DRAM
FEF00000P	FFEFFFFF	SRAM
FFF00000P	FFFFFFFFP	FLASH

The first entry in the table designates the first 1MB of system memory. The middle entry designates the location of the cache RAM mapped into system memory. The third entry designates the firmware ROM.

2. The **StartPEI** button will reset the target and step one instruction at a time until the processor enters protected mode. It then will load the PEI module symbols and run until PeiMain.
3. Alternatively, you can use the **PEIs** macro button at any time when the processor is in protected mode.

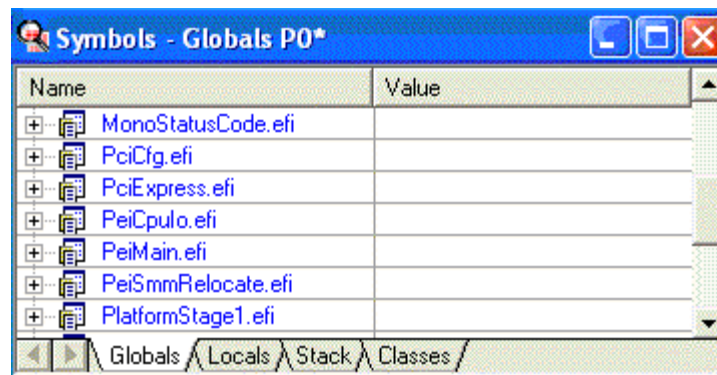


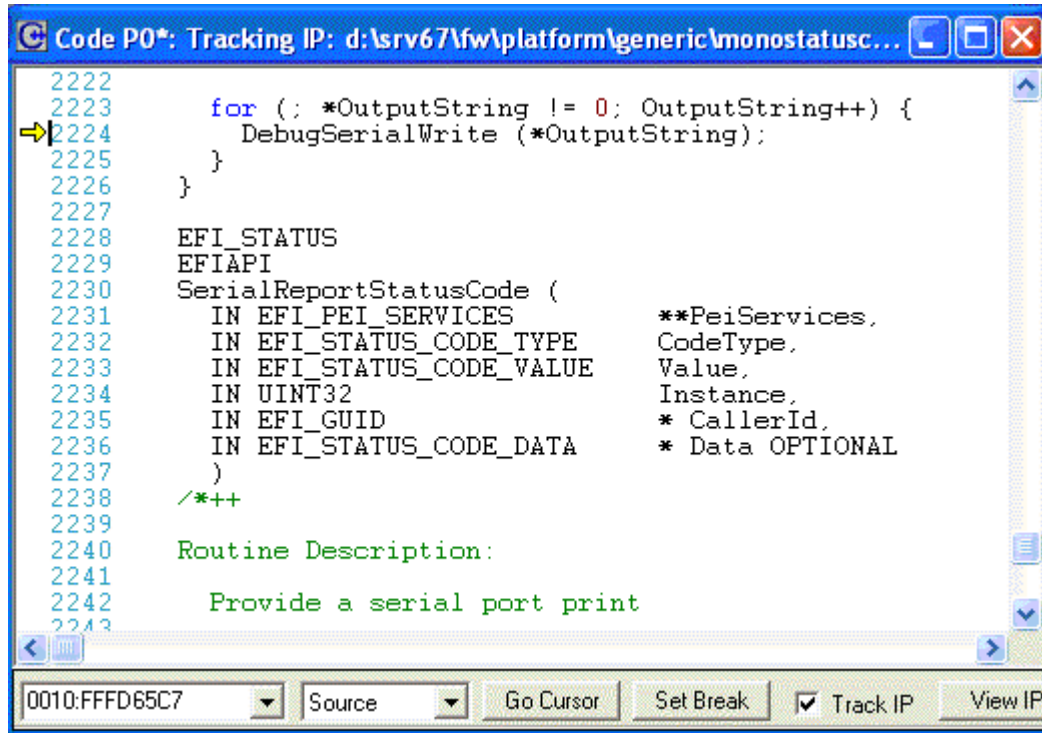
```

Command
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\S3EndOfPost.efi at
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\S3Resume.efi at 00F
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\PciExpress.efi at 0
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\TpmPei.efi at 00FFF
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\TcgPei.efi at 00FFF
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\DxeIpl.efi at 00FFF
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\CpuS3.efi at 00FFF
loading D:\SRv67\Fw\Platform\IntelMpg\SantaRosaCrb\Build\IA32\PeiSmmRelocate.efi
P0>

```

Command window after running PEIMs macro function



Symbols window after loading PEIM modules*Code window after loading PEIM modules***DXE Debugging**

Once system RAM is initialized and the PEI phase completes, the DXE environment is entered. This is less specialized than PEI; nevertheless, it requires a few SourcePoint parameters to be set.

To configure SourcePoint for source-level debugging of DXE code, follow these steps:

1. Run the target to the UEFI shell, or as far as it will go in DXE.
2. Stop the target.
3. Click the DXEs toolbar icon to load the DXE symbols.
4. Browse the source code files using the **Symbols** window and set breakpoints in your code.
5. Reset the target and go until you hit a breakpoint.

```

0010:1EE8E535 POP      ECX
0010:1EE8E536 JNE      short ptr CpuIoServiceRead+9f
318
319      break;
0010:1EE8E538 JMP      short ptr CpuIoServiceRead+d1
309      case EfiCpuIoWidthUint8:
310      for (; Count > 0; Count--, Buffer.buf += OutS
0010:1EE8E53A MOV      EBX,dword ptr [EBP]+18
0010:1EE8E53D TEST     EBX,EBX
0010:1EE8E53F JBE      short ptr CpuIoServiceRead+d1
311      *Buffer.ui8 = CpuIoRead8 ((UINT16) Address:
0010:1EE8E541 PUSH     dword ptr [EBP]+14
0010:1EE8E544 CALL     near32 ptr CpuIoRead8
→ 0010:1EE8E549 ADD      dword ptr [EBP]+14,EDI
0010:1EE8E54C MOV      byte ptr [ESI],AL
0010:1EE8E54E ADD      ESI,dword ptr [EBP]+1c
0010:1EE8E551 DEC      EBX
0010:1EE8E552 POP      ECX
0010:1EE8E553 JNE      short ptr CpuIoServiceRead+bd
329
330
331      return EFI_SUCCESS;
0010:1EE8E555 XOR      EAX,EAX
0010:1EE8E557 POP      EDI
0010:1EE8E558 POP      ESI
0010:1EE8E559 POP      EBX
332
0010:1EE8E55A POP      EBP
0010:1EE8E55B RETN
  
```

DXE Code window

HOBs

Open the **Command** window, and then click the HOBs toolbar icon to display the hand-off blocks on the target.

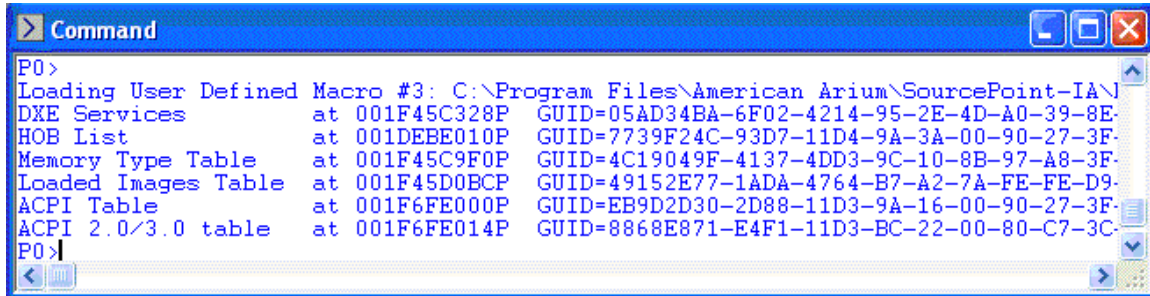
```

> Command
HOB Resource descriptor at 001DEBEC08P
Resource type      0x0 (system memory)
Attributes         0x3C03
                  Present
                  Initialized
                  Uncacheable
                  Write-combinable
                  Write-through cacheable
                  Write-back cacheable
Base address       0x0000000000000000
Length             0x000000000000A000
HOB Resource descriptor at 001DEBEC38P
Resource type      0x5 (reserved memory)
Attributes         0x0
  
```

Example of HOB display

System Configuration Table

Open the **Command** window, and then click the **SysConfigTable** toolbar icon to display the contents of the UEFI system configuration table on the target.



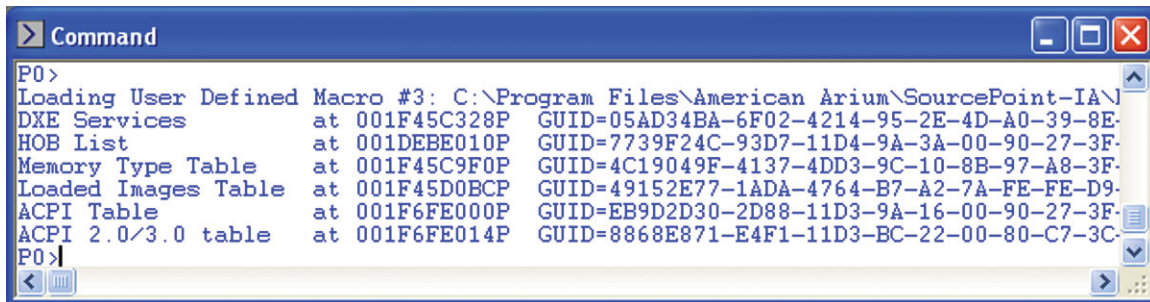
```

P0>
Loading User Defined Macro #3: C:\Program Files\American Arium\SourcePoint-IA\
DXE Services      at 001F45C328P  GUID=05AD34BA-6F02-4214-95-2E-4D-A0-39-8E-
HOB List          at 001DEBE010P  GUID=7739F24C-93D7-11D4-9A-3A-00-90-27-3F-
Memory Type Table at 001F45C9F0P  GUID=4C19049F-4137-4DD3-9C-10-8B-97-A8-3F-
Loaded Images Table at 001F45D0BCP  GUID=49152E77-1ADA-4764-B7-A2-7A-FE-FE-D9-
ACPI Table        at 001F6FE000P  GUID=EB9D2D30-2D88-11D3-9A-16-00-90-27-3F-
ACPI 2.0/3.0 table at 001F6FE014P  GUID=8868E871-E4F1-11D3-BC-22-00-80-C7-3C-
P0>
  
```

Example of System Configuration Table

UEFI System Memory Map

Open the **Command** window, and then click the **DumpMemMap** toolbar icon to display the contents of the UEFI memory map.



```

P0>
Loading User Defined Macro #3: C:\Program Files\American Arium\SourcePoint-IA\
DXE Services      at 001F45C328P  GUID=05AD34BA-6F02-4214-95-2E-4D-A0-39-8E-
HOB List          at 001DEBE010P  GUID=7739F24C-93D7-11D4-9A-3A-00-90-27-3F-
Memory Type Table at 001F45C9F0P  GUID=4C19049F-4137-4DD3-9C-10-8B-97-A8-3F-
Loaded Images Table at 001F45D0BCP  GUID=49152E77-1ADA-4764-B7-A2-7A-FE-FE-D9-
ACPI Table        at 001F6FE000P  GUID=EB9D2D30-2D88-11D3-9A-16-00-90-27-3F-
ACPI 2.0/3.0 table at 001F6FE014P  GUID=8868E871-E4F1-11D3-BC-22-00-80-C7-3C-
P0>
  
```

Example of UEFI System Memory Map

Notes

1. DXE Debugging Tip

To stop the target and load symbols just before a DXE module is dispatched, open the **Symbols** window, choose the **Globals** tab, and drill down to:

```

program: DXEMAIN.efi
module:  image (image.c)
function: CoreStartImage()
  
```

Right click on CoreStartImage and select **Open Code Window** from the pop-up menu.

Set a processor breakpoint in **CoreStartImage()** where **Image->EntryPoint()** is called. This hits before each DXE module is dispatched, but afterwards its entry is placed in tables. Each time you hit this breakpoint, click the **DXEs** toolbar icon to load the DXE symbols.

Now you can load symbols just before your DXE module runs instead of running to the UEFI shell, then loading symbols, then resetting the target, then running to your breakpoint.

2. Watchdog Timer on Intel Platforms

Some motherboards with Intel processors have a TCO timer that will assert RESET independent of the emulator. See the Arium application note titled, "Disabling the TCO Timer in an Intel I/O Controller Hub" for details. Resetting the target from SourcePoint can cause a **Target state undefined** error message to appear because the timer asserts RESET and confuses the emulator. The solution to this problem is to configure the **ICH_TCO_Timer_Disable.mac** macro to run at every target reset.

3. The UEFI firmware on the target contains strings that hold the paths to the program symbol files on your hard drive. SourcePoint macros read target memory, find these strings, then load the symbol files specified in these paths. The symbol files must be located in the path specified in the UEFI firmware.

For example, one path might look like this:

```
"Z:\Platform\IntelSsg\D845GRG\Build\IA32\DxeMain.efi"
```

This architecture, defined by Intel, presents a requirement for UEFI debugging. You must have the UEFI symbol files on the host computer in the same directories as specified in the firmware on the target. This should not be a problem if you build the UEFI firmware on the same host from which you run SourcePoint.

If the drive letter or path doesn't match exactly, you can use the `'subst'` command from the Windows command prompt to map a drive letter to a desired path (example: `'subst d: c:\working\EFI'`).

Memory Casting

C/C++ developers can declare a SourcePoint debug variable to be a pointer to a specific type of data where the type is defined in the user's loaded program symbols. When casted to a new type, the debug variable pointer acts just like a program variable pointer of that type. You can display a block of memory as a data structure or use elements of the structure in expressions. You can also display blocks of target memory as a specific symbolic type without defining a debug variable as a pointer to that block.

Defining Debug Variables of a Symbol Type as Defined in a Loaded Program

This is used to define debug variables for future use in expressions or for display. The basic syntax is:

```
define symbol [variable_name] = ( [type_name] ) [address]
```

Example

```
define symbol myVar = (myStruct) 0x1234
```

Debug variables defined in this manner are not available if the program that defines the variable's type does not have symbols loaded.

Casting Blocks of Target Memory as a Symbol Type as Defined in a Loaded Program

This is used to simply display the memory using the format of the data type. This is commonly used in **Watch** window expressions. The basic syntax is:

```
( [type_name] ) [address]
```

Example

```
(myStruct) 0x5678
```


Microsoft® PE Format Support in SourcePoint

Overview

Definition of PE

PE32/PE32+ defines the SP32 Portable Executable File Format. PE is a load time relocatable file format that can contain multiple sections/segments inside of a single file. The Extensible Firmware Interface (EFI) also utilizes the PE format for EFI applications and device drivers. For details of the format, see the Microsoft PE32/COFF File Format Specification.

Definition of PDB

The .PDB extension stands for "program database." It holds the format for storing debugging information that was introduced in Visual C++ version 1.0. One of the most important motivations for the change in format was to allow incremental linking of debug versions of programs, a change first introduced in Visual C++ version 2.0.

While earlier, 16-bit versions of Visual C++ used .PDB files, the debugging information stored in them was appended to the end of the .EXE or .DLL file by the linker. In the versions of Visual C++ mentioned above, both the linker and the integrated debugger were modified to allow .PDB files to be used directly during the debugging process, thereby eliminating substantial amounts of work for the linker and also bypassing the cumbersome CVPACK limit of 64K types.

By default, when you build projects generated by the Visual Workbench, the compiler switch /Fd is used to rename the .PDB file to <project>.PDB. Therefore, you will have only one .PDB file for the entire project.

When you run makefiles that were not generated by the Visual Workbench, and the /Fd is not used with /Zi, you will end up with two .PDB files:

- VCx0.PDB (where "x" refers to the major version of the corresponding Visual C++, either "2" or "4"), which stores all debugging information for the individual .OBJ files. It resides in the directory where the project makefile resides.
- <project>.PDB, which stores all debugging information for the resulting .EXE file. It resides in the \WINDEBUG subdirectory.

Why two files? When the compiler is run, it doesn't know the name of the .EXE file into which the .OBJ files will be linked, so the compiler can't put the information into <project>.PDB. The two files store different information. Each time you compile an .OBJ file, the compiler merges the debugging information into VCX0.PDB. It does not put in symbol information such as function definitions. It only puts in information concerning types. One benefit of this is that when every source file includes common header files such as <windows.h>, all the typedefs from these headers are only stored once, rather than in every .OBJ file.

When you run the linker, it creates <project>.PDB, which holds the debugging information for the project's .EXE file. All debugging information, including function prototypes and everything else, is placed into <project>.PDB, not just the type information found in VCX0.PDB. The two kinds of .PDB files share the same extension because they are architecturally similar; they both allow incremental updates. Nevertheless, they actually store different information.

The new Visual C++ debugger uses the <project>.PDB file created by the linker directly, and embeds the absolute path to the .PDB in the .EXE or .DLL file. If the debugger can't find the .PDB

file at that location or if the path is invalid (if, for example, the project was moved to another computer), the debugger looks for it in the current directory.

FAQs

What tool-chains has the SourcePoint PE Loader been validated against?

May 2002 microsoft platform SDK

compiler(cl) version 13.00.9500.7 (for IA64)

linker (link) version 7.00.9500.7

Microsoft Visual Studio.Net (Visual C++ Version 7) (for 32bit)

compiler(cl) version 13.00.9466

linker (link) version 7.00.9466

Microsoft Window server 2003 DDK

compiler(cl) version 13.10.2240.8 for IA64

linker (link) version 7.10.2240.8 for IA64

compiler(cl) version 13.10.2207 for AMD64

linker (link) version 7.10.2207 for AMD64

PE supports several symbol formats. Which format is supported by SourcePoint?

SourcePoint supports Codeview both in .PDB format and "non-PDB" format. SourcePoint does not support COFF symbols within a PE file.

Why are COFF symbols not supported?

COFF symbols were used with early versions of Microsoft® Windows® and with MASM. Although the COFF format has line number information, the latest MS Linker does not generate line number information when COFF is used. COFF symbols in a PE format file would not support the display of source code in the **Code** window.

Does SourcePoint support C++ with PE format?

Not in the current version of SourcePoint. PE/PDB support in SourcePoint is for C language level support is primarily intended for EFI debugging. However, SourcePoint can load the symbols of a C++ application. Some of the symbols will be readable while others will be in a mangled format. The user can differentiate "classes" from "structures," but the class properties and methods are not directly associated.

Does SourcePoint support debugging 32-bit PE applications?

SourcePoint supports 32-bit (PE32) versions of the PE format.

What linker switch is used to create CodeView/PDB?

/DEBUG /DEBUGTYPE:CV for PDB

/DEBUG /DEBUGTYPE:CV /PDB:NONE for CodeView without PDB

/DEBUG /DEBUGTYPE:COFF for COFF

Does SourcePoint support PE/PDB generated by SEPTYPE option?

No, SourcePoint does not support PDB generated by PDBTYPE:SEPTYPE. When the PDBTYPE:SEPTYPE switch is used, type information is put into separate files other than the PDB file. SourcePoint does not read these files. In Visual C++, the Separate Types button in Category: "Debug" in Link page of Project|Settings must be unchecked to generate symbols compatible with SourcePoint.

Does SourcePoint support .DBG format?

No. SourcePoint does not support the .DBG format.

Does SourcePoint support PE files containing multiple code sections (segments)?

Yes.

Known restrictions of PE/PDB support in SourcePoint

- Separate Typefile. Separate type information file is not supported as described above.
- Demand Loading. For PDB, demand loading is not supported in SourcePoint.
- Module Range. Since PDB does not provide an accurate module range, SourcePoint guesses at the last address of module from last line number of the module. Codeview (without PDB) provides the exact size of a module, and SourcePoint can have accurate module range.
- SourcePoint does not support PDB formats generated by MS Linker ver 5.xx and older.

Multi-Clustering

Arium currently supports multi-clustering for Intel IA-32, Intel 64-bit extensions to 32-bit, and AMD64 processors. Using Arium's ECM-50, you can connect to as many as four multi-clusters in a target at one time.

Note: One emulator is required for each multi-cluster.

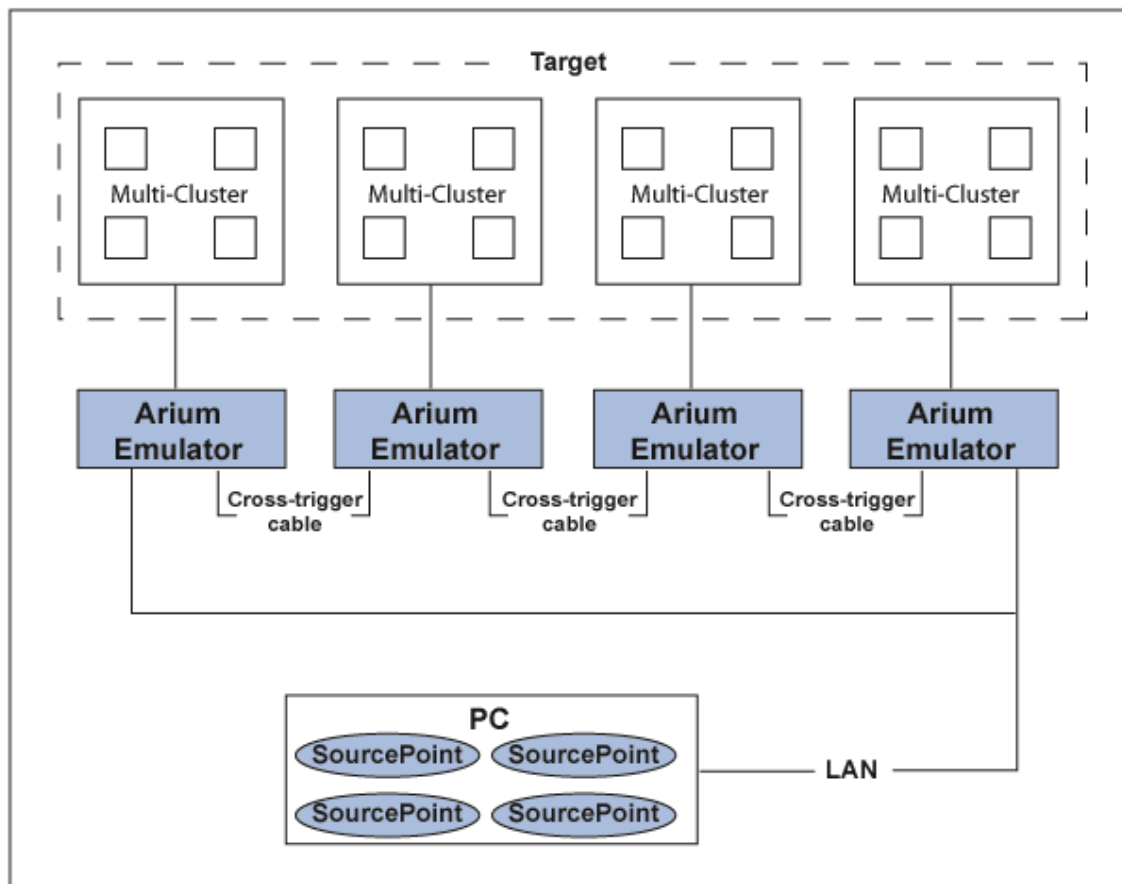
Note: The cable(s) for multi-cluster debug support must be ordered separately. Such cables are not part of the regular emulator package.

Hardware Setup

The current methodology for supporting multi-cluster consists of connecting an ECM to each cluster's debug port via the appropriate PBD, and interconnecting these ECMs by way of "cross-trigger" cables. This cable, approximately three feet long, is used to connect two ECMs to one another.

To connect emulators to a multi-cluster target:

1. Connect a cross-trigger cable to the mini-DIN socket labeled "TO NEXT" on one emulator and to the mini-DIN socket labeled "FROM PREV" on the other.
2. To connect a third and/or fourth emulator, follow the instructions above.



Sample connection

Caution: Do not connect the emulators into a ring.

To connect the emulator to the target and a host (or a network supporting a host), see the *Getting Started* installation guide that came with the emulator.

Software Setup

SourcePoint support for multi-cluster requires that a separate copy of SourcePoint be running for each emulator. The separate SourcePoint programs communicate with each other, however, so that when a **Go** command is issued within one SourcePoint, the other SourcePoint programs automatically issue their own **Go** commands.

Option #1 - Multiple Installs of SourcePoint

Install SourcePoint multiple times, once for each cluster.

Option #2 - Single Install of SourcePoint

1. Install a single copy of SourcePoint.
2. Create an “ini” file for each copy of SourcePoint. The standard “ini” file is “sp.ini” (located in the directory where SourcePoint is installed). Name the new ini files “sp2.ini”, “sp3.ini”, etc. (Do not use quotation marks around the file names.)
3. Create separate icons for each copy of SourcePoint.
4. Associate each SourcePoint icon with the correct “ini” file. For each icon, right click, then select Properties, then Shortcut, and add “-ini iniFileName” at the end of the target field (e.g., “c:\Program Files\American Arium\SourcePoint\sp.exe”-ini sp2.ini).
5. Start SourcePoint for the first cluster. Specify the connection for the emulator controlling the first cluster. (Select **Options|Emulator Connection** from the menu bar.)
6. Select **File|Project|Save As** to create a project file with the current emulator connection. Choose different project file names for each cluster.
7. Repeat steps 2 and 3 for the other emulators. The goal is to create a project file for each copy of SourcePoint that will be running. Each project file will have a different emulator connection depending on which emulator is being controlled.

Running in Multi-Cluster Mode

To run in multi-cluster mode:

1. Double-click on each SourcePoint icon to start as many copies of SourcePoint as there are emulators.

SourcePoint checks for other active SourcePoint tasks and automatically registers with them.

2. To enable multi-cluster support, select **Options|General** on the SourcePoint menu.

The **Preferences** dialog box appears.

3. Go to the **General** tab.
4. Enable **Multiclustert support enabled**.

5. Set up breakpoints in each SourcePoint and press **Go** in any SourcePoint.

The cross-trigger bus (established by the cross-trigger cables) holds the combined system stopped until the last cluster is started.

Note: When **Go** or **Stop** commands are issued in one SourcePoint (either from the menu, toolbar, or command line), that copy of SourcePoint sends messages to all the other SourcePoints to generate the same command. There is no need to select **Go** or **Stop** in each copy of SourcePoint. Likewise, when one SourcePoint hits a breakpoint and triggers, all processors in all clusters stop (see triggering latency below).

Timing

The emulators are synchronized to one another via the cross-trigger bus, but some finite differential in time exists between the "start" signal arriving at the various clusters being controlled. This time differential is caused by circuit delays and by the method required for starting the cluster type. The Intel Pentium 4 and some other soon-to-be-released processor systems require JTAG operations to start the processors, while Intel P6-class processor systems require only the negation of a PREQ signal. The time differential (or slip) between starting individual clusters in a multi-cluster system is thus dependent upon the type of system being controlled.

- Maximum slip between starting multiple Intel P6-class processor clusters is 50 ns; typical slip is 10 ns.
- All multi-clusters use a similar mechanism for stopping the processors, and thus only one set of stop "slip" times needs to be specified. Maximum slip between stopping multiple clusters is 50 ns; typical slip is 10 ns.
- In addition to the signals monitored by the emulators from their connected clusters, the external "BNC-IN" connector can be used to signal a stop to the multiple emulators connected by the cross-trigger cables. When signaling a stop to the multi-cluster environment, the maximum delay from "BNC-IN" to assertion of the stop on the cross-trigger bus is 40 ns; typical slip is 21 ns.
- So long as the emulator is configured via SourcePoint to "listen" to the "BNC-IN" connector, the "BNC-OUT" connector will reflect an assertion at this "BNC-IN" connector. The maximum time from "BNC-IN" assertion to "BNC-OUT" assertion is 78 ns; typical slip is 44 ns.

Registers Keyword Table

Data Registers	EAX	extended accumulator register, bits 0-31	ord4
	AX	accumulator register, bits 0-15	ord2
	AH	accumulator register, bits 8-15	ord1
	AL	accumulator register, bits 0-7	ord1
	EBX	extended BX register, bits 0-31	ord4
	BX	BX register, bits 0-15	ord2
	BH	BX register, bits 8-15	ord 1
	BL	BX register, bits 0-7	ord1
	ECX	extended CX register, bits 0-31	ord4
	CX	CX register, bits 0-15	ord2
	CH	CX register, bits 8-15	ord1
	CL	CX register, bits 0-7	ord1
	EDX	extended DX register, bits 0-31	ord4
	DX	DX register, bits 0-15	ord2
	DH	DX register, bits 8-15	ord1
	DL	DX register, bits 0-7	ord1
Pointer/Index Registers	EBP	extended base pointer	ord4
	BP	base pointer	ord2
	ESP	extended stack pointer	ord4
	SP	stack pointer	ord2
	EDI	extended destination index	ord4
	DI	destination index	ord2
	ESI	extended source index	ord4
	SI	source index	ord2
Code Segment Registers	CS	code segment register	ord2
	CSBAS	code segment register, base	ord4
	CSLIM	code segment register, limit	ord4
	CSAR	code segment register, access rights	ord1
Data Segment Registers	DS	data segment register	ord2
	DSBAS	data segment register, base	ord4
	DSLIM	data segment register, limit	ord4
	DSAR	data segment register, access rights	ord1
Extra Segment Registers	ES	extra segment register	ord2

	ESBAS	extra segment register, base	ord4
	ESLIM	extra segment register, limit	ord4
	ESAR	extra segment register, access rights	ord1
F Segment Registers	FS	F segment register	ord2
	FSBAS	F segment register, base	ord4
	FSLIM	F segment register, limit	ord4
	FSAR	F segment register, access rights	ord1
G Segment Registers	GS	G segment register	ord2
	GSBAS	G segment register, base	ord4
	GSLIM	G segment register, limit	ord4
	GSAR	G segment register, access rights	ord1
Stack Segment Registers	SS	stack segment register	ord2
	SSBAS	stack segment register, base	ord4
	SSLIM	stack segment register, limit	ord4
	SSAR	stack segment register, access rights	byte
Instruction Pointer and Flags Registers	EIP	extended instruction pointer	ord4
	IP	instruction pointer	ord2
	EFLAGS	extended flags register	ord4
	FLAGS	flags register	ord2
Control Registers	CR0	machine status register	ord4
	CR1	Intel reserved	ord4
	CR2	page fault linear address register	ord4
	CR3	page directory base register	ord4
	CR4	processor extensions register	ord4
System Address Registers	GDTBAS	global descriptor table, base	ord4
	GDTLIM	global descriptor table, limit	ord2
	LDTR	local descriptor table register	ord2
	LDTBAS	local descriptor table, base	ord4
	LDTLIM	local descriptor table, limit	ord2
	LDTAR	local descriptor table, access rights	ord1
	IDTBAS	interrupt descriptor table, base	ord4
	IDTLIM	interrupt descriptor table, limit	ord2
	TR	task state segment register	ord2
	TSSBAS	task state segment, base	ord4
	TSSLIM	task state segment, limit	ord4
	TSSAR	task state segment, access rights	ord1
Debug Registers	DR0	debug register	ord4
	DR1	debug register	ord4

	DR2	debug register	ord4
	DR3	debug register	ord4
	DR4	debug register	ord4
	DR5	debug register	ord4
	DR6	debug register	ord4
	DR7	debug register	ord4

Stepping

There are three commands (menu items, toolbar buttons, commands) for stepping: step into, step over, and step out of. These commands are described in detail below.

Step Into command. This single-steps the next instruction in the program and enters each function call that is encountered. This is useful for detailed analysis of all execution paths in a program.

Step Over command. This single-steps the next instruction in the program and runs through each function call that is encountered without showing the steps in the function. This is useful for analysis of the current routine while skipping the details of called routines.

Step Out Of command. This single-steps the next instruction in the program, and runs through the end of an existing function context. This is useful for a quick way to get back to the parent routine.

These commands may be interpreted in two ways, depending on whether source is available for the current execution location. If source debugging information is available for the current execution location, then it is possible to do a source-level step (**step into** or **step over**). A source-level step differs from a low-level or machine-level step by the range of addresses involved. In source-level stepping, the unit of interest is the source line (with its associated address range). In low-level stepping, the unit of interest is the machine instruction. For assembly code, source-level and low-level steps may be the same.

By using the stepping instructions in conjunction with the go/stop and breakpoint capabilities, a user may effectively track through the execution of programs.

Strategies for Source Level Stepping

Most compilers output debugging information at the level of the source line. This means that SourcePoint (or any other debugger for that matter) can source level step only a line at a time. Many languages allow the construction of multiple source statements on a single line. While this will not cause any difficulty in SourcePoint, for the purposes of stepping, it is a good idea to separate out as much functionality as possible onto separate lines.

For instance, the following C language source fragment will source level step as one statement:

Command input:

```
for ( i = 1, i < function1( 100 ), i++ ) { j = function2( i ); k += j; }
```

The intricacies of the internal execution of function1 and function2 will be missed. You could always step through the machine language generated by the compiler, but this is often quite time consuming and potentially confusing. The above C code might be rewritten as follows in order to step through the execution of function1 and function2 and the parts of the *for* block at the source level:

Command input:

```
for ( i = 1,  
i < function1( 100 ),  
i++ )
```

```
{
function2( i );
k += j;
}
```

This rewriting of the code will not affect the execution performance or effect, but will enable more effective debugging and perhaps a cleaner coding style. The rewriting is entirely optional. You might consider selective rewrites on certain parts of code that are to be debugged.

The writing of code compressed onto single lines applies to all source languages. The use of macros in assembly language, multiple statements on a line or defines in the C language, and similar constructions in other languages present the same difficulties in stepping. The debugger source level steps one source line at a time.

Stepping at source level or machine level

You can control whether stepping takes place at the source level or machine level via the **Code** window. If a single **Code** window is open, then the display mode of that window controls how stepping is performed. If the display mode is **Source**, stepping will take place at the source level. If the display mode is **Mixed** or **Disassembly**, stepping will take place at the machine level.

If multiple **Code** windows are open, the rules are more complex. The general rule is the **Code** window that is tracking the instruction pointer (has **Track IP** checked) and has the focus (contains the flashing cursor and has a highlighted title bar) determines the method of stepping. Situations where the **Code** window that has the focus is not tracking the instruction pointer may not conform to these rules. If the method of stepping is not as expected, switch focus to a **Code** window that is tracking the instruction pointer and select the desired display mode.

Step Into

The **step into** ability of the debugger can be invoked via a command, a toolbar button, or a menu item. This single-steps the next instruction in the program, and enters each function call that is encountered. This is useful for executing every path in a program.

Source Level Step Into

When the debugger performs a source level **Step Into**, machine instructions within the range of addresses defined by the source statement at the current point of execution are repeatedly executed until the point of execution lands outside the range. Upon execution of the source level **step into** function, the debugger first remembers the range of addresses for the source statement that contains the current execution point. Then, a machine-level step into is executed. The new execution point is determined. If the execution point is still contained within the range described by the source statement, then another machine-level **Step Into** is executed. This is repeated until either execution falls outside the range of the source statement or 255 steps have been executed. Note that 255 is the maximum number of steps allowed by the step command.

Machine Level Step Into

When the debugger performs a machine level **Step Into**, a machine instruction is executed. If executed via the menu or the toolbar, only one step is performed. If executed as a command (or macro), an optional repeat count is accepted. The repeat count can be between 1 and 255, inclusive. This causes the requested number of steps to be executed.

Step Over

The "step over" ability of the debugger can be invoked via a command, a toolbar button, or a menu item. This single-steps through instructions in the program. If this command is used when you reach a function call, the function is executed without stepping through the function instructions. This is useful for skipping over the execution of a subroutine and continuing with the execution of the current routine.

The step over capability may require the use of one of the four debug registers. It is always a good idea not to use all of the debug registers in breakpoints if you intend on using the **Step Over** command. If all of the debug registers are in use, a **Step Over** command will execute up to the point where the use of the debug register is required, and then it will stop with an error message dialog box.

Source Level Step Over

When the debugger performs a source level "step over," machine instructions within the range of addresses defined by the source statement at the current point of execution are repeatedly executed until the point of execution lands outside the range or a call is encountered. Upon execution of the source level **Step Over** command, the debugger first remembers the range of addresses for the source statement that contains the current execution point. Then, a machine-level **Step Over** command is executed. The new execution point is determined. If the execution point is still contained within the range described by the source statement, then another machine level **Step Over** command is executed. This is repeated until either execution falls outside the range of the source statement or 255 steps have been executed. Note that 255 is the maximum number of steps allowed by the **Step** command.

Machine Level Step Over

When the debugger performs a machine level "step over," one of two operations is performed. If the instruction at the current execution point is a call, a breakpoint is set after the call, and the target machine is given a **Go** command. All of the instructions in the subroutine called and any instructions recursively called will execute. The setting of a breakpoint requires the use of one of the four debug registers. If a debug register is not available, an error message will be displayed. If the instruction at the current execution point is not a call, a machine level "step into" is performed (see above). If executed via the menu or the toolbar, only one step is performed. If executed as a command (or macro), an optional repeat count is accepted. The repeat count can be between 1 and 255 inclusive. This causes the requested number of steps to be executed.

Step Out Of

Select the **Step Out Of** command to stop program execution at the next location after the return from the current function. This command places a breakpoint on the instruction immediately following the call instruction for the current routine. This is useful to skip over the rest of the current function and all calls made by the function. In the process of debugging, when you have determined that the current function does not contain the problem you're looking for, this provides a rapid method of proceeding with debugging after the current function.

The **Step Out Of** command requires one hardware debug register for the breakpoint. If the resource is unavailable, this routine does not change anything and produces a beep.

Symbolic Text Format (Textsym)

This file format is a simple text file to specify symbolic debug information.

File Format

Field Separator

Each field is separated by the vertical bar ('|') character. White space around the bar is optional. All leading and trailing white spaces between fields are ignored.

Signature

The first line of this text file contains a signature and version information. The "TEXTSYM format" string and version number must be as shown. White space will be ignored. If a valid signature is not found, the load will abort.

```
TEXTSYM format | V1.1 <eol>
```

Debug Information

Debug information for each symbol is specified on a separate line as specified below:

GLOBAL/LOCAL	Offset Value	CODE/DATA	Symbol Name	Object_____Size
--------------	--------------	-----------	-------------	-----------------

Where:

GLOBAL/LOCAL	Usage is tool dependent. If symbol is specified as GLOBAL, then it must be unique within this module - no duplication is allowed. Some tools may ignore symbols marked as LOCAL.
Offset Value	64-bit hex value treated as an unsigned number. The offset value is added to the address where the symbol file is loaded.
CODE/DATA	A required keyword. When the debug tool forms an IA-64 symbolic address, this field is used to determine whether the resulting symbol has a data address or an execution address. This field is ignored when reading IA-32 symbols.
Symbol Name	A contiguous ASCII string of characters that are legal to identify a variable/function name in C/C++. Symbol names are case sensitive. Length is not restricted but limited by the debug tool that consumes it.
Object Size	The size of a data object in bytes. This field is optional. It is in bytes for code and data symbols. *This field is not allowed in version 1.0 and is optional in version 1.1. Both versions 1.0 and 1.1 are currently supported.

High-level source display is not possible in the absence of line numbers. All the symbols are treated as if they are public symbols. The file name will be used as the module name to associate the symbol. De-referencing of symbols is not supported.

Example

TEXTSYM format | V1.0 <eol>

GLOBAL	0000000c00000000	CODE	ENTER_RESET <eol>	
GLOBAL	0000000000000430	DATA	OSTypeFound <eol>	
LOCAL	0000000000001234	CODE	BAR <eol>	
GLOBAL	0000000000001238	DATA	FOO	4 <eol>

Using Bookmarks

Bookmarks are temporary placeholders that allow you to mark locations in the data. They are supported in line view-based windows (e.g., **Code**, **Memory**, **Trace**, **Log**, **Command**, and **Linux Console** windows). Bookmark options can be manipulated via the **Edit** menu in on the main menu bar or via icons on the icon toolbar. The following outline provides brief information on how to use bookmarks in SourcePoint.

Adding/Removing Bookmarks

1. Bookmarks can be set on any type of line-view line (state, disassembly, source, data, etc.).
2. If display settings are changed such that a bookmarked line is no longer displayed, then the bookmark is set invalid and ignored. If display settings are changed such that the bookmarked line is displayed again, then the bookmark is marked valid and can be used again.
3. Use Ctrl+F2 to toggle a bookmark.

Navigating Bookmarks

1. F2 moves you forward to the next bookmark.
2. Shift+F2 moves backwards to the previous bookmark.

Clearing Bookmarks

1. Bookmarks are cleared automatically when a view is closed.
2. Bookmarks are cleared automatically when SourcePoint is closed; they are not saved in the project file.
3. For the **Trace** window only:
 - Bookmarks are cleared automatically when new trace data are captured.
 - Bookmarks are cleared automatically when you switch between displaying a binary trace file and emulator trace.
4. For the **Command**, **Log**, and **Linux Console** windows only:
 - Bookmarks are cleared automatically when you clear one of these windows.
 - If enough lines are added to these windows, then it is possible for lines at the beginning of the view to be discarded. If one of these lines is bookmarked, the bookmarks is cleared.
5. Ctrl+Shift_F2 clears all bookmarks in all windows.

Bookmark Indications

Bookmarks are indicated by a changed background color in the line that is marked. The background color is light blue unless you change it via the **Color** tab under **Options|Preferences**.

Which Processor Is Which

Introduction

SourcePoint orders processors from last to first on the JTAG chain. This follows the order in which the JTAG device ID is shifted out. That is, since the last processor on the JTAG chain outputs its data first, it is considered the first, or P0 processor. The next-to-last processor shifts out its data next and is considered the P1 processor, and so on.

What Does "Last on the Chain, First on the Chain" Mean?

The JTAG chain is a serial data flow from the emulator, through each processor, then back to the emulator (See Figure 1, below). As data is shifted out of the emulator, existing data that are in the processors are shifted back to the emulator. When the data goes back into the emulator, it goes into a buffer, filling the buffer from top to bottom.

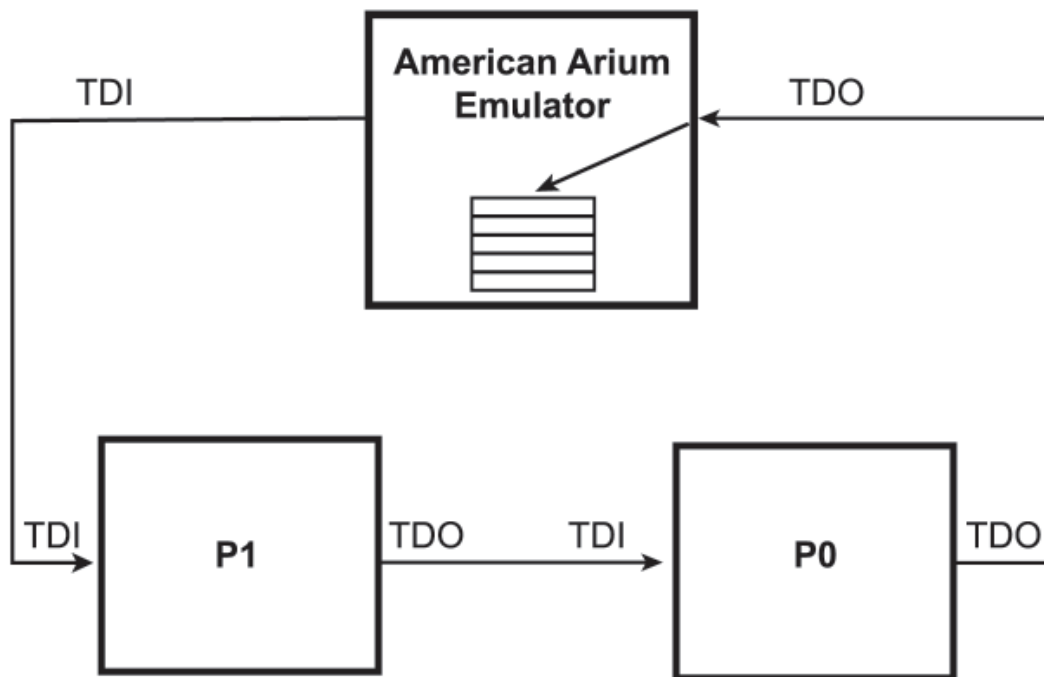


Figure 1

As an example, one of the first operations the emulator performs is getting device IDs from all the processors in the JTAG chain (the serial circuit created by connecting all processor together as in the diagram). Nearly all ARM processors have the capability to return a device ID. In their initial state, processors have a 32-bit register that contains the device ID and is attached between TDI and TDO. By shifting the data through the circuit, the device ID for the last processor (P0 in the diagram) is shifted out first and onto the top of the buffer inside the emulator. (The first device ID has been shifted out from the last processor in the circuit, the device ID for the next-to-last processor has been shifted into the last processor and more shifting needs to be done to shift it through and into the emulator into the next available space in the buffer.) At that point, the

emulator have the device ID for the last processor first, followed by the next-to-last processor. That is why SourcePoint orders the processors from last to first on the chain.

How Is This Related to the PROCESSORCONTROL Variable in SourcePoint?

The PROCESSORCONTROL variable contains a mask of which processors SourcePoint should control. The mask is actually a bit pattern representing the processors that are on the JTAG chain, the least significant bit representing P0, the next significant bit representing P1, and so on. If a particular bit is 1, then SourcePoint is to control that processor. If 0, then SourcePoint is to ignore that processor. By default, PROCESSORCONTROL has "on" all the bits that correspond to the number of processors. That is to say, if there are two processors in the chain, similar to the diagram above, then PROCESSORCONTROL is 0x03 by default. If there were four processors in the chain, then PROCESSORCONTROL would be 0x0f by default.

By setting off the bits for the corresponding processor, you can make SourcePoint ignore certain processors. For example, in the diagram above, if you only want to control P1, then you can set PROCESSORCONTROL=0x02. Likewise, if you only want to control P0, then you can set PROCESSORCONTROL=0x01.

Using another example where four processor are on the JTAG chain, they are labeled P0, P1, P2 and P3. Similar to the previous two processor examples above, P0 is still be the last one in the chain, P1 is the next to last, P2 the next to next-to-last (or the second) and P3 would be the first in the chain. Then, for example, if you wanted only to control P3, you would set PROCESSORCONTROL=0x08. Using the mask, you can control any combination of processors. In the case of, say, four processors in the chain, then to control P0 and P2, you could set PROCESSORCONTROL=0x05.

What Does It Mean to Control More Than One Processor?

When you click on the **Go** button (or use the **Go** command) in SourcePoint, all processors that SourcePoint is to control are started. When you click on the **Stop** button (or use the **Halt** command), all processors that SourcePoint is to control are stopped. However, the single step command will only single step a single processor.

SourcePoint Commands

Command-Related Topics

Array Data Type

The [define](#) command is used to create and array debug variables. Use a bracketed expression suffixed to the debug variable name to create an array. The value of the expression determines the size of the array. The definition type will determine the type of the array elements. See the example below which illustrates how to define an array named ValueArray with 32 elements of type ord4.

Command input:

```
define ord4 ValueArray[0x20]
```

You can then use a for loop to assign values to this array:

Command input:

```
define ord1 cnt = 0
define ord4 value = 0x0f
for (cnt=0; cnt < 32; cnt++)
{
    ValueArray[cnt] = value
    value = value * 0x0f
}
```

Arrays can also be initialized at the time they are defined, such as:

Command input:

```
define nstring StrList[10] = "empty"
```

Notes on Defining Arrays

- Unless otherwise specified, all array elements are initialized to 0, or the type specific equivalent.
- Arrays are global or local in scope under the same conditions as non-composite debug-variable types.
- Attempting to access an array element using an invalid index value results in an error.
- Arrays can be passed as arguments to procs and also returned as the return value of a proc. If a data type is specified, it should be followed with brackets, but without a specified array size.

Array Elements

Each element of an array is a fully functional debug variable of the specified type. The individual elements of the array behave the same as a regular debug variable in every respect. Array elements are referenced by a bracketed zero-based index. For an array of n elements, the valid indexes are 0.. $n-1$.

Arrays as Variables

Arrays are limited as to how they may be used as entities. Using an array name without an element reference (no bracketed value) refers to the complete array. The only expression operator is the binary assignment operator. There are no unary operators for arrays. The assignment operator is restricted in the following ways:

- Arrays can only be assigned the value of other arrays.
- Array-to-array assignment is only valid when the types are identical.
- If the arrays differ in size, then the destination array is resized to that of the source array.

Array Type with Debug Object Commands

Arrays are also limited as to how they are managed with debug object commands as follows:

- The **show** or **remove** commands operate on complete arrays but attempting to use these commands to manage a single array element results in an error.
- The **show** command shows the type and size of the array but not the array element values.
- The **eval** command accepts array elements but causes an error if you attempt to evaluate an array as an entity.

Character Strings

A character string is a sequence of one or more ASCII printing characters enclosed in double quotation marks. You can include special or non-printing characters in a string by preceding each with a backslash (\). If the backslash precedes a character that has no special meaning, the two-character sequence is equivalent to the single character following the backslash. You can continue strings across line boundaries only by means of the backslash (\) followed by pressing the Enter key.

The following are examples of character strings:

Command inputs:

"ABCD"

"This is a test."

"The total is 534.876."

Control Register Bit Maps and Names

1. CR0 - Machine Status Register

Bit	Mnemonic	Description
31	PG	paging bit
30	CD	cache disable bit
29	NW	not write-through bit
18	AM	alignment mask bit
16	WP	write protect bit
5	NE	numeric error bit
4	ET	extension type bit
3	TS	task switched bit
2	EM	emulation bit
1	MP	math present bit
0	PE	protection enable bit

Note: Unspecified bits are Intel reserved.

2. CR1 - Intel Reserved
3. CR2 - Page Fault Linear Address Register
4. CR3 - Page Directory Base Register

Bit	Mnemonic	Description
4	PCD	page-level cache disable bit
3	PWT	page-level writes transparent bit

Note: Unspecified bits are Intel reserved.

5. CR4 - Processor Extensions Register

Bit	Mnemonic	Description
8	PCE	enable RDPMC instruction (Intel® Pentium® Pro only)
7	PGE	page global enable bit (Intel Pentium Pro only)
6	MCE	machine check enable bit
5	PAE	physical address extension bit (Intel Pentium Pro only)
4	PSE	page size extensions bit
3	DE	debugging extensions bit

2	TSD	time stamp disable bit
1	PVI	protected-mode virtual interrupts bit
0	VME	virtual-8086 mode extensions bits

Note: Unspecified bits are Intel reserved.

The following examples illustrate how to display and, as applicable, change values in various situations.

1. The following example shows how to display and then change the value of the EBX register.

Command input:

ebx

Result:

000035F0H

Command input:

ebx = 45f0h ; ebx

Result:

000045F0H

2. The following example shows how to display and then change the value of the PG bit (bit 31 of CR0), which affects page translation.

Command input:

pg

Result:

TRUE

Command input:

pg= 0; pg

Result:

FALSE

3. The following example shows how to display the value of the accessed bit in the CS selector.

Command input:

csar.a

Result:

TRUE

Control Constructs

Control constructs are used to group and execute emulator commands.

Use control construct commands to group and execute emulator commands in loops, alternately, and as blocks. The emulator control constructs are as follows:

`break`

`continue`

`do {commands} while (expr)`

`for (comand1 ; expr ; command2) {commands}`

`if (expr) {commands1} [else {commands2}][. . .]`

```
switch (expr)
{
case label-expr: [ commands; ][ . . . ]
[ default: commands; ]
}
```

`while (expr) {commands}`

Each control construct is a separate entry in this help file.

Control Variables

Control variables are built-in debug variables defined by SourcePoint. Use the assignment operator (=) to assign a value to control variables. If you enter a control variable at the command line prompt without assigning a value, the value of the control variable is displayed.

Control variables may be used in expressions. For more information see the individual entries for each control variable.

Control variable	Type	Read/Write	Options	Default
asmmode	int2	rw	use16, use32	use16
base	int2	rw	bin, dec, oct, & hex	hex
breakall	bool	rw	true, false	false
cachememory	bool	rw	true, false	false
defaultpath	nstring	rw	n/a	n/a
displayflag	bool	rw	true, false	false
editor	nstring	rw	editor filepath	"notepad.exe"
execution point (\$)	ptr	rw	(address)	n/a
first_jtag_device	ord4	r	n/a	n/a
homepath	nstring	r	n/a	n/a
isem64t	bool	r	true, false	n/a
isrunning	bool	r	true, false	n/a
itpcompatible	bool	rw	true, false	false
last_jtag_device	ord4	r	n/a	n/a
macropath	nstring	r	n/a	n/a
num_devices	ord4	r	n/a	n/a
num_jtag_devices	ord4	r	n/a	n/a
num_processors	ord4	r	n/a	n/a
processorcontrol	int2	rw	$0 - 2^n - 1$	$2^n - 1$
processors	int2	r	1...n	n/a
projectpath	nstring	r	n/a	n/a
safemode	bool	rw	true, false	false
tabs	int2	rw	1-8	4
targpower	bool	r	true, false	n/a
targstatus	nstring	r	n/a	n/a
tck	nstring	rw	varies	varies
use	int2	rw	use16, use32	use16
verify	bool	rw	true, false	false

viewpoint (view)	int2	rw	p0...pn	p0
vpalias	nstring	rw	n/a	n/a
yieldflag	bool	rw	true, false	false

n = number of processors in system

Note: If you are reading through the online Help guide, you can click on any of the underscored words to be taken directly to a detailed description of the control variable. If you are reading our printed documentation, please refer to Appendix A, *Command Line Commands*.

Data Types

The built-in data types supported by SourcePoint

Discussion:

Data types are used when defining debug variables and when accessing target memory.

Type	Description
ord1 (byte)	unsigned 8-bit quantity (byte is an alias)
ord2 (word)	unsigned 16-bit quantity (word is an alias)
ord4 (uint, dword, offset)	unsigned 32-bit quantity (uint, dword and offset are aliases)
ord8 (qword)	unsigned 64-bit quantity (qword is an alias)
ord12	unsigned 96-bit quantity (supported but ord16 is used)
ord16	unsigned 128-bit quantity (Not available for memory access)
char	ASCII character
nstring (string)	a string object (similar to CString)
int1	signed 8-bit quantity
int2	signed 16-bit quantity
int4 (int)	signed 32-bit quantity (int is an alias)
int8	signed 64-bit quantity
int16	signed 128-bit quantity (not available for memory access)
real4 (float)	signed 32-bit floating point value (float is an alias)
real8 (double)	signed 64-bit floating point value (double is an alias)
real10	supported, but real8 is used
pointer	represents an address in target memory
boolean (bool)	true (non-zero value) or false (zero value)
Array	Array of elements of any valid debug data type with the exception of pointers. (Not available for memory access.)

Example 1

To define a debug variable called o4Val and assign it a value of 5:

Command input:

```
define ord4 o4Val = 5
o4Val
```

Result:

5

Example 2

To display 20 bytes of memory at address 1000 as 16 bit quantities:

Command input:

SourcePoint 7.7.1

ord2 1000 len 10

Result:

00001000 0080 8D01 42B9 D00A F8B4 03B8 EB04 1000
00001010 F500 712E F3C1 018F F8A0 12A8 E008 F8B4

Allowable Operation by Type

The data type of an operand dictates what type of operation you perform on it. This implicit control is illustrated in the first of two tables that follow. Column 1 represents the type of operator you must use for the data types the emulator recognizes. The operator groups are defined in the second table.

Operator Groups / Operand Types

	Ord	Int	Bool	Real	Str	Char
assignment	X	X	X	X	X	X
bitwise	X	X	-	-	-	X
additive	X	X	-	X	X	X
multiplicative	X	X	-	X	-	X
relational	X	X	X	X	X	X
logical	X	X	X	-	-	X
& (address of)	X	X	X	X	X	X

Note: The modulus operators (% and %=) are not supported for real data types (+ and += are supported).

Operators by Group

```

assignment  =
bitwise     & | ^ <<>> ~ &=
           |= ^= <<>>
additive    + - ++-- += -=
multiplicative * / % *= /= %=
relational  == != < > <= >= <>
logical     ! && || ^^
& (address of) & .

```

Debug Procedures

User-defined debug procedures.

Syntax

```
define proc [ data-type ] proc-name
```

```
[ ( argument-name [ ... ] ) [ define-spec ] ]
```

```
{ [ forward-spec ] [ commands | return argument-name ] [...] }
```

define-spec is the following:

```
[ [ define ] argument-type argument-name ] [...]
```

forward-spec is the following:

```
forward { proc [ data-type ] proc-name | data-type var-name }
```

Where:

define	Signals creation of a user-defined procedure or procedure argument.
proc	Specifies a user-defined procedure.
data-type	Specifies the data type to be returned.
proc-name	Specifies the name of a debug procedure.
argument-name	Specifies the name of an argument that is used in the procedure. Separates the names of arguments with commas.
argument-type	Specifies the data type of the argument.
commands	Any emulator commands (except for include), including the argvector and argcount predefined local variables.
forward	Specifies a forward-referenced or recursive debug object. Forward references any debug objects that have not yet been defined.
var-name	Specifies a user-defined name for the debug object that is being forward referenced (that is, not yet defined).
data-type	Specifies an emulator data type.
return	Specifies an argument name whose value is returned upon completion of proc execution.

Discussion

Use debug procedures (procs) to define custom functions. Create a proc with the **define** command. You can use any text editor to initially create and edit a proc. You can also enter a proc at the command line. A proc is executed when it is called by name, just as a built-in function is executed.

You can define debug procedures that accept arguments. If the argument-type argument is not specified, the caller data type is used as the default. When executing a proc, an error message is displayed if the proc requires arguments that have not been passed to it.

To define debug procedures that accept a variable number of arguments, use two predefined local variables, **argvector** and **argcount**. The **argcount** variable tracks the number of arguments supplied when the function is called. The **argvector** variable (array) stores the actual arguments passed when a function is called.

Recursive or reentrant debug procedures are supported to the extent of available host memory. Debug procedures can also call other debug procedures that have been previously defined. Use the **forward** option to reference debug objects (including other debug procedures) that have not yet been defined. To define recursive debug procedures, the **forward** option must be used.

Debug variables defined inside the proc are local to the proc unless declared as global. Debug variables inside the proc not declared as global are automatically removed after the execution of the proc.

Use the **return** command to return values from a proc. If the **return** command is not used or executed, the proc returns a null value. Use the return reserved word only inside procedure definitions; it is not a separate command. If the return data type does not match the calling data type, then an explicit data type conversion occurs. The default data type of the return value is `ord4`.

If a proc executes an emulation command (such as **go** or **istep**), the statements after the emulation command are executed immediately unless followed by the **wait** command. The **wait** command prevents the emulator from executing any more commands until a breakpoint is reached.

Note: You can use debug procedures and macro files to create a library of frequently used commands. The emulator displays a syntax error when a proc processes an undefined proc symbol or variable. Define all program symbols before referencing.

Examples

1. The following example shows how to define a proc to execute a series of commands to display the flags bits in binary.

Command input:

```
define proc flag_bits
{
  define byte savebase = base
  base = 2t
  flags
  base = savebase
}
flag_bits
```

Result:

```
000000000100000110
```

The following example shows how to define and then execute a procedure named `avg` that accepts three parameters and returns their average:

Command input:

```
define proc avg (a,b,c)
{
  return ((a + b + c) /3)
}
base = 10t
avg ( 4,6,3)
```

Result:

4T

- The following example shows how to use the forward option to refer to undefined debug procedures.

Command input:

```
define proc int8 calc (a,b,c)
define int8 a
define int8 b
define int8 c
{
  forward proc int8 min /* Forward references procs */
  forward proc int8 max /* min and max. */
  if ((a > 0) && (b > 0))
    return (max (a,b) * c) \
  else if ((a < 0) && ( b < 0))
    return (min(a,b) * c) \
  else return (0)
} // End of calc proc
define proc int8 min (x,y) // Define min proc
{
  return ((x < y) ? (x): (y))
}
define proc int8 max (x,y) /* Define max proc. */
{
  return ((x > y) ? (x): (y))
}
base = 10t
calc (2,4,6) /* Execute calc proc. */
```

Result:

0T#24T

- The following example shows how to use the forward option to create a recursive procedure.

Command input:

```
define proc ord8 factorial (n)
define ord8 n
{
  forward proc ord8 factorial // Recursive proc
  // forward reference
  if (n == 0)
    return (1) else
    return (n * factorial (n - 1))
}
```



```
base = 10t  
factorial (3)
```

Result:
0T#720T

Command input:
factorial (4) // Execute the proc

Result:
0T#24T

Command input:
factorial (5)

Result:
0T#120T

Command input:
factorial (6)

Result:
0T#720T

Debug Pointer Types

A pointer can replace a hard address (e.g., if you define address C0000:0000FEEF to be "abc", then the pointer-reserved word is "abc"). The pointer-reserved word can define a pointer debug object. Pointer debug variables can be used anywhere a memory address is required.

Debug Variables

Variables store values that can change during execution or be changed by user command. Variables are used as debug variables created during the debugging session.

A debug variable is represented in an expression with a user-defined name.

Expressions

Expressions are made up of one or more operands combined with operators.

Syntax

op [binary-operator op] [. . .]

op is the following:

{ [left-unary operator] operand
| operand [right-unary operator] }

Where:

left-unary-operator Acts on a single operand.

operand Specifies one of the following:

- A number.
- A quoted string (e.g., "string").
- The name of a user-defined debug object.
- A built-in or user defined function. For more information see the **proc** command.
- A memory access operation using data types. For more information see the **memory access** commands. Some operands are user-defined; others are system-defined.

right-unary-operator Acts on a single operand.

binary-operator Acts on two operands. The result is a single operand.

Discussion

An expression is a single value or a combination of operands (one or more constants, debug variables, or functions) separated by operators. Evaluating an expression applies the operators to the operands until a single result is obtained.

An expression entered as a command (not as a part of a command) is evaluated directly and the result is displayed in the current base.

You can use the contents of a variable or a memory location in an expression.

Example

Command input*:

ax=4+1
ax+1

Result:

00000006H

Command input:*ax***Result:***0005H*

Note: Command inputs are cumulative; you must start at the first input to get the results shown from subsequent inputs.

Operands

The three classes of operands are constants, functions, and the contents of debug variables and memory locations. Within each class, some operands are user-defined and others are built-in. An expression can be a single operand without any operators.

Constants

Constants do not change value during execution. Constants can be any supported emulator data type. User-defined constants include ordinals, integers, real numbers, and strings. You can also use the boolean values true (1) and false (0) as constants.

Note: To avoid confusion with debug objects and symbolic references, hexadecimal numbers must not begin with a letter. Begin hexadecimal numbers with a leading zero. For example, 0ah is valid, whereas ah is considered to be the AH register.

Ordinal numbers contain one or more valid digits and an optional suffix character (y, q, t, h) or prefix character (0y, 0q, 0n, 0x) indicating the number base. If you omit the prefix or suffix character, the digits are interpreted according to the current setting of the base control variable.

Integer numbers contain one or more valid digits and an optional suffix character (y, q, t, h) or prefix character (0y, 0q, 0n, 0x) indicating the number base. If you omit the prefix or suffix character, the digits are interpreted according to the current setting of the base control variable.

Integers of the form nK (Kilobyte) and **nM** (Megabyte) are valid constants, where n is an unsigned decimal integer, K is 1024 decimal, and M is 1048576 decimal.

Real Numbers use the syntax [sign] numerals [e [sign] [numerals]]. Real numbers are always decimal. You can place the decimal point anywhere in the sequence of numerals (e.g., 0.1, 12345.6789). The optional exponent (e) element of a real number forms scientific notation (e.g., 43.337e4, -1.0445e-5).

Note: The decimal Point is required in real numbers to distinguish those from hexadecimal integers of the form nnenn. Numerals are required on the left and the right of the decimal point.

A **string** has up to 254 characters enclosed in quotes ("). The value of a string is its ASCII representation, with a byte for each corresponding character. You can enclose single-character strings (char) in apostrophes ('). You can use single-character strings as operands for arithmetic operations.

The following example shows how to use strings:

Command input:

```
define nstring var2 = "This is a string."  
var2
```

Result:

"This is a string."

Command input:

```
'a'
```

Result:

01100001Y

Command input:

```
R // A char used in an arithmetic operation  
'a' + 5t
```

Result:

00000066H

Functions

Built-in functions and user defined procedures.

Syntax

[name =] function-name [(expr [,...])]

Where:

name	Specifies the debug object to which the function return value is assigned.
function-name	Specifies the reserved word for a built-in function or the name of a predefined debug procedure.
expr	Specifies one or more numbers or expressions separated by commas. Parentheses are required to delimit expr.

Discussion

The emulator functions are divided into several categories, as outlined in the following table. Most of the commands are explained in more detail in their own entries in this chapter.

Character Functions

isalnum	isalpha	isascii	isctrl
isdigit	islower	isprint	ispunct
isspace	isupper	isxdigita	
toascii	tolower	toupper	

File I/O Functions

fopen	fclose	fgetc	fgets
fputc	fputs	fprintf	

I/O Functions

printf	putchar	puts
------------------------	-------------------------	----------------------

Math Functions

abs	acos	asin	atan
---------------------	----------------------	----------------------	----------------------

atan2	cos	exp	log10
loge	Pow	sin	sqrt
tan			

String Functions

strcat	strcmp	strcpy	strlen
strncat	strncmp	strncpy	strtod
strtol	strtoul		

Miscellaneous Functions

ctime	flist	rand	srand
sleep	time		

Macros and Procs

A macro is a batch-like file that, once loaded, automatically executes a series of commands. A proc is a C-like subroutine that can be called at any time to execute a series of commands. Macros and procs are valuable tools that eliminate redundant steps. Instead of manual initialization of a target, a macro or proc may reset the target, modify specific target chipsets, set registers to known values, and start the target bootstrap sequence.

Procs may also be created in the **Command** window. The benefit of creating a command line proc is that it then may be repetitively called; however, there is no way to save the proc to a file, so it will be lost when you leave the current debug session.

Procs that have been loaded into memory cannot be altered. You must delete them, modify the macro file, and re-load the file.

Differences Between Macros and Procs

There are three major differences between macros and procs:

- Macros are immediately executed; procs are simply loaded in memory.
- After a proc is loaded, it can be re-run any time; the only way to re-run a macro is to re-load it.
- Procs can be removed from memory; macros are never "stored" in memory (they execute upon load), so they may never be "removed."

Minor differences between macros and procs include:

- Arguments may not be passed to macros
- There is a "structure" to a proc; macros are unstructured

Sample Proc

Command input:

```
Reset define proc init_target()
cs = 2000h
{
  csbase = 2000h reset
  eip = 10 cs = 2000h
  go csbase = 2000h
  eip = 10
  go
}
```

Note: A macro file may contain any number of procs but, at most, one macro. Upon loading the macro file, the procs are loaded into memory (ready for immediate execution), and the series of commands that comprise the macro begins executing. The macro may also call any of the previously loaded procs.

Macro and Proc Limitations

The maximum number of lines contained within a single proc cannot exceed 600 lines.

Procs that have been loaded into memory cannot be altered. You must delete the proc, modify the macro file and re-load it.

Memory Access

Display and modify memory.

Syntax

To display memory:

```
[[px]] data-type addr-spec [display-base]
```

To modify memory:

```
[[px]] data-type addr-spec = {expr[,...] | data-type addr-spec |  
debug-var-array}
```

To fill memory:

```
[[px]] data-type destination-range = expr
```

To copy memory:

```
[[px]] data-type destination-range = data-type source-range
```

Where:

[px]	is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
data-type	specifies the data type used to access memory (e.g., ord1, ord2, ord4, etc.). For more information, see Data Types .
expr	specifies a number or an expression. You can enter more than one expression by using a comma as a separator.
addr-spec	{addr addr-range}
addr	specifies an address. For more information, see Memory Access: Addresses, found later in this topic.
addr-range	is an address range. There are two ways to specify a range: addr1 to addr2 or addr length expr.
destination-range	is a range of memory to write.
source-range	is a range of memory to read.
addr1 to addr2	specifies a range of memory beginning with address addr1 and including address addr2. Addr2 must be greater than addr1.
addr length expr	specifies a range of memory beginning with address addr1. The range includes a number of items (specified by expr).
expr	specifies a number or an expression. You can enter more than one expression by using a comma as a separator.
debug-var-array	is an array of debug variables to write to memory (e.g., ord4 data[10]).
display-base	specifies a temporary override of the current display base (bin oct dec

hex).

Discussion

For memory read commands, the requested data is displayed in the current base (specified by the base control variable), unless an override is specified. Addresses are always displayed in hexadecimal. If the data-type is ord1 (or byte), the ASCII representation of the data is shown on the right-hand side of the screen with non-printing characters displayed as a period.

Memory is read using the viewpoint processor unless a processor override is specified.

For a memory copy command, the source and destination ranges may not overlap, and the destination range must be equal to or greater than the source range. If the destination range is larger, the source data are repeated to fill the destination range of memory.

The data-type size is the resolution used for copy or fill. Only complete data items are written to the destination, and the source and destination data-types must match.

You can also use memory access commands in an expression. For example, define ord4 var1 = byte 100hp takes the value at location 100hp, translates it to an ord4, and puts in a debug variable name var1.

When a memory access operation is part of an expression, ranges of addresses are not allowed.

Note: If verify=true, the emulator reads back what is written.

Example 1

To display a byte of memory:

Command input:

```
int1 20000h
```

Result:

```
42H
```

Example 2

To write 32 bits of memory at address 100:

Command input:

```
ord4 100 = 12345678  
ord4 100
```

Result:

```
12345678H
```

Example 3

To set a debug variable from 4 bytes of memory at addr 1000p:

Command input:

```
define ord4 myData = ord4 1000p
```

Example 4

To fill a range of memory with a single value and then display the range:

Command input:

```
ord1 100h length 20h = 30h
ord1 100h length 20h
```

Result:

```
00000100 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
"000000000000000000"
00000110 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
"000000000000000000"
```

Example 5

To copy a range of memory:

Command input:

```
ord1 200h length 20h = 42
ord1 100h length 10h = ord1 200h length 10h
ord1 100h length 10h
```

Result:

```
00000100 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
"BBBBBBBBBBBBBBBBBB"
```

Example 6

To write a repeating sequence of values and display the new values:

Command input:

```
ord2 700h length 5t = 1,2,3
ord2 700h length 5t
```

Result:

```
00000700 0001 0002 0003 0001 0002
```

Example 7

To copy a value from one memory location to another and read the new value:

Command input:

```
ordl 200hp = ordl 100hp
ordl 200hp
```

Result:

```
42H "B"
```

Example 8

To copy the contents of a file to target memory at address 0:

Command input:

```
define ord4 file1
define ord4 nItemsRead
define ordl buf[1000]
define ptr pMem = 0

file1 = fopen("test.dat", "r")
while (feof(file1) == 0)
{
    nItemsRead = fread(buf, file1)
    ordl pMem length nItemsRead = buf
    pMem += nItemsRead
}
fclose(file1)
```

Example 9

To copy the first 50 bytes of an array to target memory at address 0:

Command input:

```
define ordl buf[1000]
define ptr pMem = 0
ordl pMem length 50 = buf           // copy first 50 bytes
```

Example 10

To copy target memory beginning at address 1000h into an nstring variable (note that memory is read until a terminating null character is found, or until 1000 characters have been read):

Command input:

```
define nstring filename = nstring 1000h
filename
```

Result:

```
"c:\doc\test.txt"
```

Memory Access: Addresses

This section describes addresses used for memory access commands.

Syntax

```
expr [ p ]
```

Where:

expr	specifies a number or an expression that will evaluate to a virtual address .
p	causes an address to be interpreted as a physical address.

Discussion

Use memory access commands to access memory in the target system. When the <addr> option appears in the syntax guide, enter an appropriate address, pointer debug variable, or an expression that evaluates to an address.

The emulator supports physical and virtual addressing. It assumes that numeric addresses are virtual unless overridden by a "p" (without quotation marks) suffix for physical address.

Virtual Address

A virtual address is the default emulator address type. The Memory Management Unit allows an address to be mapped to a different physical address. This is frequently used to manage physical memory allocation, as in the case where memory allocation of multiple processes with potentially conflicting address mappings is needed.

Physical Address

A physical address is the address used as an index into physical memory.

Operand Character Delimiters

The emulator recognizes operand delimiters as shown in the following table.

Character	Function
space	logical blank
tab	logical blank
carriage return	line terminator
back slash (\)	command continuation indicator
semicolon (;)	command separator
quote (")	string delimiter
apostrophe (')	single-character delimiter
period (.)	symbolic delimiter
colon (:)	module name prefix
comma (,)	list element separator
/* ... */ (comment)	logical blank
// (comment)	logical blank

Operators

The following table lists the emulator operators in order of precedence and describe how evaluation occurs. Precedence determination parallels the C programming language. Expressions containing the logical operators &&, ||, and ^^ evaluate left to right and terminate as soon as a result is determined. This limit means that under some circumstances, the right side of the operation may not be evaluated. For example, the left side of an && operation is evaluated; if the result is zero, the right side is not evaluated.

Emulator Operators in Order of Precedence			
Category	Symbol	Associate	Function
primary	()	left	group expressions
	[]	left	index into ustring
right-unary	++	left	post-increment
	--	left	post-decrement
left-unary	*	left	indirection
	-	left	unary minus
	!	left	logical NOT
	~	left	bitwise NOT
	++	left	pre-increment
	--	left	pre-decrement
binary	*	left	multiplication
	/	left	division
	%	left	modulus
	+	left	addition
	-	left	subtraction
	<<	left	shift left
	>>	left	shift right
	<	left	less than
	>	left	greater than
	<=	left	less than or equal
	>=	left	greater than or equal
	==	left	equivalence
	!=	left	non-equivalence
	<>	left	non-equivalence
	&	left	bitwise AND
		left	bitwise OR
	^	left	bitwise XOR
	&&	left	logical AND
	^^	left	logical XOR

		left	logical OR ternary
	?:	right	three-element conditional expression (for example: (a>b)?(a):(b) displays the greater value, a or b)
assignment	=	right	simple assignment
	+=	right	implied operand addition
	-=	right	implied operand subtraction
	*=	right	implied operand multiplication
	/=	right	implied operand division
	%=	right	implied operand modulus
	>>=	right	implied operand right shift
	<<=	right	implied operand left shift
	&=	right	implied operand bitwise AND
	^=	right	implied operand bitwise XOR
	=	right	implied operand OR

Real Numbers

Syntax

[sign] numerals.numerals [e [sign] [numerals]]

Discussion

Real numbers are always decimal. You can place the decimal point anywhere in the sequence of numerals. The following example shows how to enter real numbers:

0.1
12345.6789

The optional exponent (e) element of a real number forms scientific notation. The following example shows how to use scientific notation:

43.337e4
-1.0445e-5

Note: The decimal point is required in real numbers to distinguish those from hexadecimal integers of the form *nnenn*. Numerals are required on the left and the right of the decimal point.

Register Group Commands

These commands are used to display entire groups of registers. The availability of each command depends on the processor type.

Syntax

`regs // displays the general registers`

`cregs // displays the control registers`

`dregs // displays the debug registers`

`fregs // displays the floating point registers`

`sregs // displays the segment registers`

`mmx // displays the mmx registers`

Example

Command input:

mmx // display mmx registers in command window

Sub-Expressions

Use parentheses to create sub-expressions with expressions. When you nest parentheses, the sub-expression in the innermost pair of parentheses is evaluated first.

1. The following example shows how to use sub-expressions.

Command input:

```
base = 10t // Set number base to decimal
2 * (10 / (5 - 3))
```

Result:

```
10T
```

2. The following example shows how to display the results of expressions.

Command input:

```
base = 10t // Set number base to decimal
5 * 5 // Binary operation
```

Result:

```
25T
```

Command input:

```
5 * (6/2) // Binary and primary operations
```

Result:

```
15T
```

Command input:

```
(3>6) ? (3): (6) // Ternary operation
```

Result:

```
6T
```

Command input:

```
base = 2t // Set number base to binary
define byte i = 1010y
i = i >> 1 // Binary operation (shift right)
i
```

Result:

```
00000101Y
```

Command input:

```
i++ // Right unary operation (Post-increment)
```

Result:

```
00000101Y // Displayed value did not change
```

Command input:

```
i // Post-increment operations display
```

Result:

00000110Y

Note: Using the ++ operator results in display of the value being incremented. When you want to suppress the display of the value, use the `i = i + 1` operation.

Symbolic References

References to program addresses and variables

Syntax

label

procedure

variable

variable[array-expr]

composite-variable.member[.member]

compound-variable

variable-ref=expr

*ptr-variable

&variable

Where:

label	program label
procedure	procedure name
variable	variable name
composite-variable	structure or union name
member	structure or union member name
compound-variable	a combination of other variable types
ptr-variable	pointer variable name
[array-expr]	specifies a number or expression identifying an element in an array
variable-ref	specifies a variable, an array variable, a composite variable or a compound-variable
expr	specifies a number or expression

Discussion

A program symbol table contains the names of all objects in the program, including the type and (for some objects) the length of each object. A symbolic reference identifies an object by name. When you use a symbolic reference in a command or expression, the emulator returns the value corresponding to the object. The value returned depends on the object type. This section reviews the kinds of symbolic references and the value represented. It also discusses special operators used with symbolic references, the address of operator (&) and the indirection operator (*), the direct-selection operator (.) and the indirect selection operator (->).

Symbol Table

The **load** command reads information about the program symbols from the object file named in the command. This information is stored internally in SourcePoint's symbol table. Symbol information is available in the **Symbols** window. Symbols can be used in place of addresses in the **Memory**, **Code** and **Breakpoints** windows. In addition, symbolic references can be used from within the **Command** window.

Names

All symbolic references involve the names of objects. Symbol names are case-sensitive. The legal characters in a name are defined by the language used in generating the object file. If a name conflicts with a reserved keyword, an emulator control variable, a debug variable, or a processor register name, then preface the name with the reserved keyword override operator (\).

Labels and Procedures

When you specify a label name or procedure name, the address associated with that object is returned. The address of operator (&) is ignored when used with a label or procedure name.

Variables

When you specify a variable name, the value associated with that object is returned.

Command

input:

```
usi           // usi is an unsigned short int
```

Result:

```
0001H
```

Command

input:

```
f           // f is a float
```

Result:

```
1.234500
```

Command

input:

```
f=14.67
```

```
f
```

Result:

```
14.670000
```

Array Variables

An array consists of elements of a given type. To read or write an individual element, specify the index of the element. The address of operator (&) can be used to return the address of an array element.

Command input:

```
ai           // ai is an array of integers
```

Result:

```
ai[0]: -1
ai[1]: -2
ai[2]: -3
ai[3]: -4
ai[4]: -5
ai[5]: -6
ai[6]: -7
ai[7]: -8
ai[8]: -9
ai[9]: -10
```

Composite Variables

A composite variable contains a collection of different objects called members. In "C" these include structures and unions. The direct-selection operator (.) is used to access individual members of a composite variable. If a pointer to a composite variable is specified, then the indirect-selection operator (->) is used to access members. The address of operator (&) can be used to return the address of a member of a composite variable.

Command input:

```
ints           // ints is a structure with 3 members: a, b and c
```

Result:

```
a: 0
b: 0
c: 0
```

Command input:

```
ints.b=5       // change one member
ints
```

Result:

```
a: 0
b: 5
c: 0
```

Compound Variables

The program can contain compound forms such as arrays of arrays, arrays of structures, structures of arrays, and structures of structures. The rules for references to these compound forms are a combination of the previously discussed rules for variables.

Command input:

```
IntsArray           // IntsArray is an array of structures of type int (see Composite
                      variables)
```

Result:

IntsArray [0]:

a: -1

b: -2

c: -3

IntsArray [1]:

a: 1

b: 2

c: 3

Command input:

```
IntsArray [1] .b=-5    // change one member of one element
IntsArray [1]
```

Result:

a: 1

b: -5

c: 3

Pointer Variables

Pointer variables contain addresses that reference program variables. When a program variable is defined as a pointer to another program variable that has a specific data type, use the indirection operator (*) to obtain the value of the variable.

Command input:

```
ints                // ints is a structure with 3 members: a, b, and c
```

Result:

a: 0

b: 5

c: 0

Command input:

```
&ints               // display address of ints
```

Result:

0020:00009AD0

Command**input:**

plnts // plnts points to int

Result:

0020:00009AD0

Command**input:**

**plnts // display ints through the pointer plnts*

Result:

a: 0

b: 5

c: 0

Command**input:**

plnts->b=0 // change a member of ints through plnts

Result:

a: 0

b: 0

c: 0

Changing the Value of a Variable

Variables can be assigned new values from within the **Command** window or the **Symbols** window. To change the value of a variable, the variable must be active (be in the current or global scope). You cannot change the address corresponding to a procedure or label. The value assigned is converted to the variable type.

Type Conversions

Type conversions occur automatically. If the two operands associated with a binary operator are of different types, an implicit type conversion is done to make the two the same type. Before a conversion takes place, however, the object to be converted is expanded to its maximum precision. An error message is generated if the conversion is not allowed.

Types and Type Classes

The following table lists the emulator data types by category with their basic definitions.

Emulation Processor Data Type Descriptions		
Category	Data Type	Description
ordinals	Byte	(same as ord1)
	ord1, uchar	1-byte unsigned value
	ord2, uint, word	2-byte unsigned value
	ord4, ulong, dword	4-byte unsigned value
	Offset	(same as ord4)
	Ord8	8-byte unsigned value
integers	int1	1-byte signed value
	int2, int	2-byte signed value
	int4, long	4-byte signed value
	int8	8-byte signed value
boolean	boolean, bool	1-byte boolean value
real	Real4, float	4-byte real number
	Real8, double	8-byte real number
character	Char	1-byte ASCII character
string	Nstring, String	variable length, null-terminated character string
pointer	pointer	pointer

Commands

#define Command

The **#define** command is used to create debug aliases.

Syntax

#define alias-name commands

Where:

alias-name identifier that serves as an alias for the given command string

commands command or commands that are referenced by the alias name

Discussion

Use the **#define** command to define a debug alias. A debug alias is a new string or alias for a command line string that can be one or more commands.

The debug alias can be defined inside or outside a control construct, a compound statement, or a debug procedure and is always global.

Example

This example defines an alias for a frequently used command:

Command input:

```
#define ld load c:\src\targdev  
ld
```

#undef Command

Use the **#undef** command to remove a debug alias definition.

Syntax

```
#undef alias-name
```

Discussion

The **#undef** command is used to remove alias definitions that were created with either the **#define** or literal **define** commands.

Example

Command input:

```
#define ld include \\test\\test1.cmd  
ld  
#undef ld
```

csr

Read/write control bus registers for Uncore devices.

Syntax

[variable=] csr([DID,]reg) [= value]

Where:

reg	is an expression that evaluates to a valid control bus register number.
value	is an expression that evaluates to a 32 bit value to load in reg.
variable	is a debug variable to hold the return value.
DID	is the device ID of the target device of this command.

Discussion

This function allows access to the Control Status Registers for Uncore devices. The csr function will also perform a CSRACCESSREAD on read operations in order to scan out the control register contents. During writes, the csr function only executes a CSRACCESS IR/DR scan. The csr function returns a 32 bit value. If a core device is selected, the function will automatically perform the operation on the core device's related Uncore device. If the device selected is not a core device, the function will check and make sure it is; otherwise an error message will be displayed.

Example 1

Command input:

```
csr(0x1001) = 0x12340000  
csr(0x1001)
```

Result:

```
12340000H
```

Example 2

Command input:

```
csr(0, 0x1001) = 0x12345678  
csr(0, 0x1001)
```

Result:

```
12345678H
```

Example 3

Command input:


```
define ord4 ord4val = csr(0x1001)
```

Example 4

Command input:

```
define ord4 ord4val = csr(2,0x1001)
```

Aadump command

The **aadump** command displays the current configuration of SourcePoint and the emulator.

Syntax

String aadump ([filename])

Where:

filename A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks.

Example

```
define nstring foo-          // save output to an nstring variable
aadump()
aadump()                    // save output to Command window
aadump ("windb")            // save output to a file called "windb"
```

Abort Command

The **abort** command aborts command file processing.

Syntax

abort

Discussion

The **abort** command aborts current command file processing. If command file execution is nested (nested including commands), all command files are terminated.

Abs Command

The **abs** command returns the absolute value of an expression.

Syntax

```
[ name = ] abs (expr)
```

Where:

- | | |
|--------|---|
| name | Specifies a debug object to which the function return value is assigned. If <i>name</i> is not specified, the return value is displayed on the next line of the screen. |
| (expr) | Specifies a number or an expression that evaluates to an integer or real number. The parentheses are required. |

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Acos Command

The **acos** command returns the arc cosine of an expression. The return value is in the range 0 to pi. If *expr* is greater than 1 or less than -1, acos returns the value 0 (zero).

Syntax

```
[ name = ] acos (expr)
```

Where:

name	Specifies a debug object of type real8 to which the function return value is assigned. If <i>name</i> is not specified, the return value is displayed on the next line of the screen.
(expr)	Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Asin Command

The **asin** command returns the arc sine of an expression. The return value is in the range $-\pi/2$ to $\pi/2$. If *expr* is greater than 1 or less than -1, **asin** returns the value 0 (zero).

Syntax

[name =] asin (expr)

Where:

name	Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
(expr)	Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Asm Command

The **asm** command serves two purposes. It displays memory as disassembled instructions and assembles instructions in-line.

Syntax

Disassembler:

asm addr-spec

Assembler.

asm addr-spec = "statement" [, "statement" ...]

asm addr =

addr-spec is one the following: { addr | addr to addr | addr length expr }

statement is one the following: { instruction | directive }

Where:

asm	Displays memory as assembler instructions or invokes the in-line assembler.
addr	Specifies an address for display or assembly. If you enter an offset only address, the default segment is the current code segment.
length	Specifies the number of instructions to be displayed.
expr	Specifies a number or an expression that indicates the number of instructions to be displayed.
instruction	Intel assembly language instruction.
directive	Microsoft MASM 6.11 assembly language directive or comment.

Discussion

Use the **asm** command to display memory as disassembled instructions or to in-line assemble instructions into memory. The output from the **asm** command is of a fixed format suitable for parsing where consistency is required. For more control of disassembler output, the code window should be used. The in-line assembler is provided to enable quick patches in active memory. The assembler uses a single pass and, therefore, cannot use labels that haven't been defined yet. This is not an insurmountable problem because the command language offers the ability to define labels, and the assembler offers the org directive for out-of-order assembly.

Disassembler

Displays memory as disassembled instructions. When disassembling, the displayed instructions will be of the form:

addr codebytes mnemonic [[operand,] operand]

Where:

addr Address of the start of the instruction.

codebytes Bytes of memory used to codify the instruction displayed as a concatenated string of hexadecimal digits.

mnemonic Instruction mnemonic.

operand Instruction operand. The number of operands is instruction specific. There may be zero or more.

Disassembler Examples

1. The following example displays a single instruction at offset 0ah in the current code segment.

Command input:

asm 0ah

Result:

00000000AH 0000 ADD [BX+SI],AL

2. The following example shows how to display three instructions beginning at the current execution point.

Command input:

asm cs:ip length 3

Result:

*00C3:00000000H 55 PUSH EBP
00C3:00000001H 8BEC MOVE EBP,ESP
00C3:00000003H 81EC04000000 SUB ESP,4H*

3. The following example shows how to display two instructions beginning at offset 0 in the code segment with selector 25 referenced through the LDT with selector 98.

Command input:

asm 098:025:0000H length 2

Result:

*0098:0025:00000000H 9A00000000D100 CALL 0D1:0H
0098:0025:00000007H 66CF IRET*

4. The following example shows how to display instructions from linear address 11426 to 11430.

Command input:

asm 11426L to 11430L

Result:

*00011426L 0000 ADD [BX+SI],AL
00011428L 0000 ADD [BX+SI],AL*


```
0001142AL 0000 ADD [BX+SI] ,AL
0001142CL 0000 ADD [BX+SI] ,AL
0001142EL 0000 ADD [BX+SI] ,AL
```

Assembler

Assembles instructions in line using the current processor modes and settings and places the instruction code bytes in memory. The current focus processor settings are used by default, but they may be overridden by the command language or assembler directives.

The assembler has been designed to accept all of the assembler forms output by the SourcePoint disassembler as well as the common forms found in Intelâ and Microsoftâ assemblers. For instance, all of the following forms are acceptable and produce the same code.

mov	ax, table[bx][di]
mov	ax, table[di][bx]
mov	ax,table[bx+di]
mov	ax,[table+bx+di]
mov	ax,[bx][di]+table

Operand size can be explicitly represented with the PTR operator. The PTR keyword should be preceded immediately by one of the size keywords.

byte ptr	unsigned byte (8-bit)
sbyte ptr	signed byte (8-bit)
word ptr	unsigned word (16-bit)
sword ptr	signed word (16-bit)
dword ptr	doubleword (32-bit)
sdword ptr	signed doubleword (32-bit)
qword ptr	quadword (64-bit)
tbyte ptr	ten-byte (80-bits)

For instance:

mov	eax, byte ptr foo
mov	eax, word ptr bar

Jump size can be explicitly represented with one of the distance operators.

short	jump short, relative
near	jump near, relative

near16	jump near 16-bit, relative
near32	jump near 32-bit, relative
far	jump far, absolute
far16	jump far 16-bit, absolute
far32	jump far 32-bit, absolute

For instance:

jmp	near ptr target
jmp	far32 ptr thing+0x122

Numbers entered into the assembler have the default radix specified by the command language. The default can be changed via the **base** command.

Radix overrides are allowed as either prefix or suffix operators. These are case insensitive.

prefix	suffix	operation
0y	y	binary
0o 0q	o q	octal
0n 0t	n t	decimal
0x \$	h	hexadecimal

For instance,

$$0y1111 = 1111y = 15t = 0n15 = 0xF = 0000Fh$$

Some shorthand suffix operators are available. These are also case insensitive.

suffix	operation
k	times 1,024
m	times 1,048,576

For instance:

$$1k = 1024t$$

The \$ symbol may be used as a shorthand key for the current assembly address. The \$ can be used alone or inside an expression (e.g., \$+10). This makes it easy to enter loops. For example, the following generates an infinite loop at the current location given by cs:eip.

asm \$ = "jmp \$"

Numeric expressions can be formed wherever a number or address is required. The following expression operators are recognized.

[]	expr [base]	used to generate based and/or indexed addressing modes
:	seg:expr	used for segment overrides
/	n / m	integer divide
*	n * m	integer multiply
+	+n	unary plus
+	n + m	integer addition
-	-n	unary minus
-	n - m	integer subtraction
()	(n * m) - p	precedence grouping

Operator precedence is the same as in MASMÒ 6.11.

When assembling, the instructions can be entered in one of two different forms: batch and interactive.

Batch Assembly

The first form allows multiple instructions to be entered on a single line. The instructions must be enclosed in quotation marks and separated by commas. Multiple lines can be used. Simply end the line with a comma to indicate continuation. The instructions are parsed and placed in memory after the entire form has been completed. Any errors encountered will be reported at the end of the form. This form is well suited to inclusion in macros or procs. This is also the form to use to fill a region of memory with an instruction sequence.

```
asm addr-spec = "statement" [, "statement" ...]
```

addr-spec is one the following:

```
{ addr | addr to addr | addr length expr }
```

statement is one the following:

```
{ instruction | directive }
```

Where:

asm invokes the in-line assembler.

addr specifies an address or an expression that evaluates to an address. If you enter an offset only address, the default segment is the current code segment.

instruction Intel assembler language instruction.

directive Microsoft MASM 6.11 assembler language compatible directive or comment.

If a range of memory is given as the addr-spec using either the addr to addr or addr length expr form, then that range of memory will be filled with the code bytes generated by the statements given. This is very useful for rapidly filling memory with single or repeated instruction sequences.

Command input:

asm 4000p length 100t = "nop"

Note: Code bytes must fit evenly in the range or an error will be generated.

The following example shows what happens when an attempt is made to fit a two-byte instruction into a 17-byte address range.

Command input

asm 4000p to 4010p = "in al,61h"

Result:

The code generated (2 bytes) won't fit evenly into the given memory (17 bytes).

Batch Assembly Examples

1. The following example shows how to patch ten NOP instructions starting at linear address 5000

asm 5000L len 10 = "nop"

2. The following example shows how to fill a region of memory starting at physical address 4000 with twenty repetitions of a repeating sequence of instructions

asm 4000p len 20 = "add bx,ax", "add cx,ax"

Interactive Assembly

The second form provides an interactive interface. The first line gives the asm command, the starting address, and the equals sign. Each successive line gives a single instruction. To end assembly, the special command ENDASM can be used. Alternatively, if the form is being typed interactively by the user, a blank line can be used to terminate assembly. When used in an include file, the ENDASM command is required.

Note: Interactive assembly is not available within procedures.

asm addr =

[processor @ address > [label:] statement]

statement is one the following:

{ instruction | directive | endasm }

Where:

asm	invokes the in-line assembler.
addr	specifies an address or an expression that evaluates to an address. If you enter an offset only address, the default segment is the current code segment.
instruction	Intel assembler language instruction.
directive	Microsoft MASM 6.11 assembler language compatible directive.
label	any unused string of characters suitable for a pointer address. The label must be followed by a colon. This performs the same function as the command <code>redefine pointer</code> .
endasm	special directive used to terminate assembly. When used interactively, a blank line performs the same function.

Unlike the batch assembly mode, the interactive mode performs memory updates as the statements are entered. The user can follow the progress by noting that the address prompt will advance to the next address. If an error is encountered during interactive assembly, a message is output, the address doesn't advance, and the user is given another chance to enter the statement.

If a range of memory is given as the `addr-spec` using either the `addr to addr` or `addr length expr` form, then that range of memory will be filled with the code bytes generated by the statements given. This is very useful for rapidly filling memory with single or repeated instruction sequences.

Interactive Assembly Examples

1. The following example shows how to patch code starting at linear address 4000

```
asm 4000L =
P0 @00004000L>mov bx,ax
P0 @00004002L>mov cx,ax
P0 @00004004L>
```

2. The following example shows how to patch in an entire program. The example is designed to be entered via the `include` command. It sets up an environment and defines some symbolic addresses. It then uses the in-line assembler to enter the program.

Please note that the comments use the `//` form at the beginning to comment the command language lines and then the comments switch to the `;` form to comment the assembly language.

<i>// Set up for execution</i>	
<i>cr0.pe=0</i>	<i>/* turn off protected mode */</i>
<i>cr0.pg=0</i>	<i>/* turn off paging *</i>
<i>csar.d=0</i>	<i>/* set to 16 bit mode *</i>
<i>cs=400h</i>	
<i>ds=600h</i>	
<i>ss=800h</i>	
<i>esp=FFEh</i>	
<i>eip=00000000h</i>	

<i>//Install the program</i>	
<i>redefine pointer</i>	<i>FRED = \$</i>
<i>redefine pointer</i>	<i>MYTEST = \$ + 19h</i>

<i>ASM FRED =</i>			
	<i>JMP</i>	<i>MYTEST</i>	
<i>DATA1</i>	<i>DB</i>	<i>0</i>	<i>; DATA1</i>
	<i>DB</i>	<i>0</i>	
<i>JOE:</i>	<i>MOV</i>	<i>AL,0</i>	<i>; AL=0</i>
	<i>DEC</i>	<i>AL</i>	<i>; AL-FF</i>
	<i>MOV</i>	<i>byte ptr [DATA1],AL</i>	<i>; write FF byte</i>
	<i>MOV</i>	<i>AL, byte ptr [DATA1]</i>	<i>; read FF byte</i>
	<i>RETN</i>		<i>; return</i>
<i>BILL:</i>	<i>MOV</i>	<i>AL,BL</i>	<i>; AL=FF</i>
	<i>OUT</i>	<i>80h,AL</i>	<i>; write FF to I/O port 80</i>
	<i>IN</i>	<i>AL,80h</i>	<i>; read I/O port 80 into AL</i>
	<i>RETN</i>		<i>; return</i>
	<i>ORG</i>	<i>MYTEST</i>	
<i>MYTEST:</i>	<i>MOV</i>	<i>AL,0</i>	<i>; AL=0</i>
	<i>MOV</i>	<i>BL,FF</i>	<i>; BL=-1</i>
	<i>MOV</i>	<i>CL,15t</i>	<i>; CL=15</i>
<i>LOOP1:</i>	<i>INC</i>	<i>AL</i>	<i>; LOOP1: AL=1,5,10</i>
	<i>INC</i>	<i>AL</i>	<i>; AL=2,6,11</i>
	<i>INC</i>	<i>AL</i>	<i>; AL=3,7,12</i>
	<i>INC</i>	<i>AL</i>	<i>; AL=4,8,13</i>
	<i>PUSH</i>	<i>AX</i>	<i>; save</i>
	<i>CALL</i>	<i>JOE</i>	<i>; call JOE</i>
	<i>POP</i>	<i>AX</i>	<i>; restore</i>
	<i>INC</i>	<i>AL</i>	<i>; AL=5,10,15</i>
	<i>PUSH</i>	<i>AX</i>	<i>; save</i>
	<i>CALL</i>	<i>BILL</i>	<i>; call BILL</i>
	<i>POP</i>	<i>AX</i>	<i>; restore</i>

	<i>JMP</i>	<i>LOOP1</i>	<i>; go to LOOP1</i>
<i>ENDASM</i>			

Assembler Directives

A number of assembly directives are supported. Most of the directives are meant for interactive use, but all are available in batch.

Address Mode Directive. { use16 | use32 }

Where:

use16 Temporarily overrides the code segment address register d-bit. This allows you to input 16-bit code while the processor is in 32-bit mode. *

use32 Temporarily overrides the code segment address register d-bit. This allows you to input 32-bit code while the processor is in 16-bit mode. *

* The command language **use** command may have already overridden the d-bit. If so, then the assembler use16 and use32 directives temporarily override the mode set by the command language override and not the current processor default.

Address Directive

org addr-expr

Where:

addr-
expr numeric and/or symbolic expression that evaluates to an address.

Data Directive

data-op data-value

data-value is one the following:

{ data | count dup ([data,] data) }

data is one the following (depending or data-op):

{ integer | float | string }

data-op is one the following:

db or byte	Defines unsigned bytes of data (8-bit) Defines unsigned numbers from 0 to 255 Also used for strings
sbyte	Defines signed bytes of data (8-bit) Defines signed numbers from -128 to +127
dw or word	Defines unsigned words of data (16-bit) Defines unsigned numbers from 0 to 65, 535 (64K)
sword	Defines signed words of data (16-bit) Defines signed numbers from -32,768 to +32,767
dd or dword	Defines doublewords of data (32-bit) Defines unsigned numbers from 0 to 4,294,967,295 (4M)
sdword	Define signed doublewords of data (32-bit) Defines signed numbers from -2,147,483,648 to +2,147,483,647
df or dfword	Defines farword data (48-bit) Defines pointer variables.
dq or dqword	Defines quadwords of data (64-bit) Defines 8-byte integers used with floating-point instructions
dt or tbyte	Defines ten-byte data (80-bits) Defines 10-byte integers used with floating-point instructions
ddq or dqword	Defines 16-byte data (128-bit)
real4	Defines short real data (32-bits) 1.18×10^{-38} to 3.40×10^{38}
real8	Defines long real data (64-bits) 2.23×10^{-308} to 1.79×10^{308}
real10	Defines ten-byte real data (80-bits) 3.37×10^{-4932} to 1.18×10^{4932}
real16	Defines 16-byte real data (128-bit)

Assembler Directives Examples

1. The following example shows how to enter a string as data at linear address 1000

```
asm 1000L =
P0@00001000L>byte "This is a test string."
P0@00001016L>
```

2. The following example shows how to clear 10 bytes to zero starting at linear address 5432

```
asm 5432L =
P0@00005432L>db 10 dup ( 0 )
P0@0000543CL>
```

3. The following example shows how to enter a few floating pointer numbers starting at physical address 4000

```
asm 4000p=
```



```
P0@00004000P>real4 12.34  
P0@00004004P>real8 5.234  
P0@0000400CP>real16 543.34  
P0@0000401CP>
```

4. The following example shows how to enter data at one location (linear 1000) and then place instructions to use the data at another location (linear 2000)

```
asm 1000l=  
P0@00001000L>dw 1  
P0@00001002L>org 2000l  
P0@00002000L>mov eax, word ptr [1000]  
P0@00002004L>
```

Asmmode Control Variable

This control variable sets the default address size used by the **asm** command.

Syntax

`asmmode = {expr | use16 | use32 | }`

Where:

`use16` Indicates 16-bit addressing.

`use32` Indicates 32-bit addressing.

`expr` Specifies a number or expression that must evaluate to 16 or 32 decimal.
The default is determined by the current mode of the processor.

Discussion

Use the **asmmode** control variable to set the default address size used by the **asm** command. Entering the control variable without selecting an option displays the current setting.

When set to `use16` (the default) the debug tool interprets assembler addresses as 16-bit. When set to `use32`, the debug tool interprets assembler addresses as 32-bit.

The **asmmode** control variable is identical in function to the **use** control variable.

Example

To set the **asm** control variable to interpret addresses as 32-bit:

Command input:
asmmode = use32

Atan Command

The **atan** command returns the arc tangent of an expression. The return value is in the range $-\pi/2$ to $\pi/2$.

Syntax

[name =] atan (expr)

Where:

name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

(expr) Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Atan2 Command

The **atan2** command returns the second arc tangent of expr2 divided by expr1. The return value is in the range -pi to pi.

Syntax

```
[ name = ] atan2 (expr1, epxr2)
```

Where:

name	Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
(expr1, expr2)	Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Base Control Variable

This control variable displays or changes the default number base.

Syntax

```
base [= {expr | bin | oct | dec | hex}]
```

Where:

expr Specifies a number or an expression that evaluates to one of the number base prefixes. If any other value is entered, an error message is displayed. The default is hexadecimal.

bin Sets the default number base to binary.

oct Sets the default number base to octal.

dec Sets the default number base to decimal.

hex Sets the default number base to hexadecimal.

Discussion

Use the **base** control variable to display or change the number base. All input is interpreted according to the current base except in the presence of a base suffix or prefix. All numeric output displays in the current base except for some special cases (e.g., real numbers always display in decimal). If you enter the **base** control variable without options, the current value displays.

You can also use **base** as an expression within other commands and as a variable (e.g., variable = base). The **base** control variable is type ord2.

The override prefixes and suffixes are shown in the following table.

RADIX Prefixes and Suffixes

Prefix	Base	Suffix
0y	Binary	y
0o	Octal	q,o
0n	Decimal	n,t
0x	Hexadecimal	h

Note: Use a base suffix when setting the base to ensure correct results. For example, base = 10 is ineffective and results in no change to the base regardless of the current base.

Examples

1. The following example shows how to display the current number base.

Command input:

base

Result:

0010H

2. The following example shows how to set the current number base to decimal.

Command input:

base = 10t

base

Result:

10T

3. The following example shows how to set the current number base to hexadecimal.

Command input:

Base=hex

base

0010H

4. The following example shows how to set the processor FLAGS register by using the FLAGS register command with a base suffix to override the default number base.

Command input:

flags = 0010011000001001y

5. The following example shows how to save and then restore the current display base.

Command input:

define ord2 svBase = base

base = hex

base = svBase

Bell (Beep) Command

This command causes an alert to sound.

Syntax

bell

beep

Examples

1. This example uses the bell command.

Command input:

if (eax != 0) bell

2. This example uses the beep command.

Command input:

if (eax != 0) beep

bits

Access the contents of a bit-field within a register, MSR or debug variable.

Syntax

```
[[px]] bits(component, bit-offset, bit-size) [= expr]
```

Where:

[px]	is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override
component	is a valid register name or debug variable.
bit-offset	is a valid expression yielding the bit index where the bit field begins. This value must be less than the size of the component specified.
bit-size	is a valid expression yielding the size, in bits, of the bit field. This value must be less than the size of the component specified minus the bit offset.
expr	is a valid numeric expression which is to be assigned to the bit field. This expression, if it results in a value greater than possible in the bit field, will be truncated to the bit-size during assignment.

Discussion

Use the bits function to access the contents of a bit-field within a register, MSR or debug variable. Use bits and #define together to define virtual registers or register components.

Example 1

To access bit 5 of register EAX:

Command input:

```
bits(eax, 5, 1)
```

Result:

1

Example 2

To clear bit 5 of register EAX:

Command input:

```
bits(eax, 5, 1) = 0
```

Example 3

To define a debug alias for bit 5 within the EAX register:

Command input:

```
#define magic_bit bits(eax, 5, 1)
magic_bit          /*output which follows assumes bit was clear*/
```

Result:

```
0
```

Command input:

```
magic_bit = 0xffff      /*value truncated to bit 0*/
magic_bit
```

Result:

```
1
```

Example 4

To define a virtual register mapped to the upper 16 bits of EAX:

Command input:

```
#define myReg bits(eax, 16t, 16t)
eax = 0
myReg = 1234
eax
```

Result:

```
12340000H
```

Example 5

To modify the upper 4 bits of a debug variable:

Command input:

```
define ord4 myData = 0  
bits(myData, 28t, 4) = 4  
myData
```

Result:

40000000H

Break Command

This command exits from a control block.

Syntax

```
break
```

Discussion

Use the **break** control construct to cause termination of the nearest enclosing **while**, **do while**, **for**, or **switch** control construct. The **break** construct only affects control constructs. To terminate program emulation, use the **halt** command.

Example

The following example shows how to use a break construct to terminate the while loop when the variable *n* equals 0.

Command input:

```
define int2 n-3t      /*define integer variable*/
while (1)             /* Begin infinite loop. */
{
  n -= 1              /* Decrement variable n. */
  printf ("n = %d \n",n) /* Display value of n. */
  if (n == 0)         /* Break when n is zero. */
    break
}
```

Result:

```
n=2
n=1
n=0
```

Breakall Control Variable

The **breakall** control variable displays or changes whether all target processors start and stop together in a multiprocessor system

Syntax

```
breakall [ = bool-cond ]
```

Where:

bool-cond specifies a number or an expression that must evaluate to true (non-zero) or false (zero)

Discussion

Use the **breakall** control variable to control whether all target processors in a multiprocessor system start and stop together. The default setting for **breakall** is true. Entering the control variable without an option displays the current setting.

If **breakall** is set to false, each processor in a multiprocessor system can be controlled independently of the others. A viewpoint override or the current viewpoint in which the **Go** command is used determines which processor is run.

If **breakall** is true, all processors in a multiprocessor system start when a go operation is executed.

Examples

1. Example 1

```
Command input:  
breakall     // display the current setting  
true
```

2. Example 2

```
Command input:  
breakall=false  
go           // only P1 processor is run
```

3. Example 3

```
Command input:  
breakall=false  
go           // P0 processor override used to run only P0 processor
```

Busbreak, Busremove, Busdisable, Busenable Commands

These commands are used to set, clear, display, enable, and disable bus analyzer breakpoints

Syntax

busbreak	
busbreak =	[sts-spec,] fetch [,bit-spec] [, loc-spec] [,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] data access [,bit-spec] [, loc-spec] [,data-spec] [,size-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] data read [,bit-spec] [, loc-spec] [,data-spec] [,size-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] data write [,bit-spec] [, loc-spec] [,data-spec] [,size-spec] [,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] i/o access [,bit-spec] [, loc-spec] [,data-spec] [,size-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] i/o read [,bit-spec] [, loc-spec] [,data-spec] [,size-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] i/o write [,bit-spec] [, loc-spec] [,data-spec] [,size-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] interrupt acknowledge [,bit-spec] [, vector-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] special transaction [,bit-spec] [, cycle-type-spec][,bus-seq-spec] [, ext-spec]
busbreak =	[sts-spec,] deferred reply [,bit-spec] [,bus-seq-spec] [, ext-spec] (1)
busbreak =	[sts-spec,] branch trace message [,bit-spec] [, location-spec][,target-spec] [,bus-seq-spec] [, ext-spec] (2)
busremove	[all]
busremove =	{type-spec location-spec size-spec } [...]
busenable =	{type-spec location-spec size-spec processor-spec } [...]
busdisable	[all]
busdisable =	{type-spec location-spec size-spec processor-spec } [...]

Where:

bit-spec: (status-bit-name = { 0 | 1 | X } [,...]) (parentheses required) **(4)**
bus-seq-spec b[us sequence] = { T1 | T2 | T3 | T4 | Q1 | Q2 }
cycle-type-spec: [cycle type] = { any cycle | shutdown | flush | halt | sync | flush ack | nop **(1)** | flush | stop clk ack **(1)** | smi ack **(1)** }
data-spec: d[ata] = masked value
ext-spec e[xternal probes] = 16 bit masked value

loc-spec: l[ocation] = physical address
 processor-spec: p[rocessor] = { 0 | 1 | 2 | 3 } **(3)**
 size-spec: s[ize] = { byte | word | dword }
 sts-spec: { e[nabled] | d[isabled] }
 target-spec: t[arget] = linear address
 type-spec: { execute | fetch | data access | data read | data write | i/o access | i/o read |
 i/o write | reset | smm entry | smm exit | btm | interrupt acknowledge | special
 transaction | deferred reply | advanced | bnc in **(1)** }
 vector-spec: v[ector] = physical address

- (1)** Intel® Pentium® Pro and Pentium II/III processors only
- (2)** Intel Pentium processors: location or target may be specified, but not both
- (3)** Intel Pentium processors: 0 or 1 only
- (4)** Status bit names are processor-specific. See the graphic below for an example of status bit names.

Discussion

The **busbreak** command sets and displays bus analyzer breakpoints (bus breaks). A **busbreak** with no arguments displays a list of the current bus breaks. There are six breaks available.

The **busremove** command removes any or all of the bus breaks. Arguments to this command qualify which bus breaks are to be removed. For instance, **busremove=data write, s=byte** removes all bus breaks with the type set to data write and size set to byte. **Busremove** with no arguments removes all bus breaks.

The **busenable** command selectively enables bus breaks. Arguments to this command qualify which bus breaks are to be affected. For instance, **busenable=fetch** enables only bus breaks with the type set to fetch.

The **busdisable** command selectively disables bus breaks. Arguments to this command qualify which bus breaks are to be affected. For instance, **busdisable=fetch** disables only bus breaks with the type set to fetch. In no arguments are specified, all bus breaks are disabled. Bus breaks can also be set, displayed, etc. from the **Breakpoints** window.

Examples

Command inputs:

Busbreak

busbreak = fetch, location=1000:1234

busbreak = branch trace message, target=2000:1435

Busremove

busremove = data write, size=byte

busremove = i/o access

Busdisable

busdisable = interrupt acknowledge

// display current bus breaks

*// set a bus break on a fetch at location
1000:1234*

*// set a bus break on a btm cycle with
target address=2000:1435*

// remove all bus breaks

*// remove all bus breaks with type set
to data write and size set to bytes*

// remove all I/O bus breaks

// disable all bus breaks

*// disable all bus breaks with type set to
interrupt acknowledge*

busenable = loc=1000:1234

*// enable all bus breaks with location
set to 1000:1234*

Cachememory Control Variable

The **cachememory** control variable displays or changes whether command line memory accesses use cached memory.

Syntax

`cachememory [= bool-cond]`

Where:

`bool-cond` specifies a number or an expression that must evaluate to true (non-zero) or false (zero)

Discussion

Use the **cachememory** control variable to control how SourcePoint handles command line memory accesses. The default setting for **cachememory** is false. Entering the control variable without an option displays the current setting.

When SourcePoint reads target memory, it normally reads blocks of 256 bytes at a time. This minimizes the time it takes for refreshing **Code** and **Memory** windows. The data read are cached in SourcePoint. Whenever a go or step operation is performed, this cache is cleared.

The **Command** window is an exception, however. Whenever a command is executed that results in a memory access (`asm`, `ord1`, `ord2`, `ord4`, etc.), SourcePoint always reads from target memory, even if it already has the data in its cache. It also reads only the amount of data requested (e.g., an `ord4` command reads exactly four bytes). This is so that accesses to memory-mapped I/O works properly.

There are times, however, primarily when executing command files that perform numerous memory accesses, that it is preferable to use the block-read, cached-memory approach. That is the purpose of the **cachememory** control variable. When false, the command window reads and writes only the number of bytes specified and does not cache data read. When true, the command window reads memory in blocks and caches the data read. Command files that perform a number of memory operations run much faster when **cachememory** is set to true.

Examples

1. Example 1

Control input:

```
cachememory // display the current setting
false
```

2. Example 2

Control input:

```
cachememory = true           // enable block memory reads and memory caching
ord4 100                    // display 3 memory values, since cachememory is true, only
```


one target memory will occur

ord4 10
ord4 108

3. Example 3

Control input:

cachememory = false

ord4 100

ord4 10

ord4 108

// disable block memory reads and memory caching

*// display 3 memory values, since cachememory is false,
three target memory reads will occur*

Cause Control Variable

The **cause** control variable displays the reason for the last target stop.

Syntax

`cause`

Discussion

The **cause** control variable returns a string indicating the reason for the last target stop. SourcePoint usually can determine which code breakpoint caused execution to stop. This is not always possible with data breakpoints.

Possible return values include:

"Unknown reason"

"User stop"

"Step completed"

"Target reset"

"Processor breakpoint @ 00010004"

"Software breakpoint @ 00010008"

Note: The cause string is automatically displayed in the SourcePoint status bar on target stop.

Example

Command input:

`cause`

Result:

software breakpoint @ 00010008

Character Functions

Built-in functions for character classification and transformation.

Syntax

[name =] function (char-expr)

Where:

Name	Specifies the debug object to which the function return value is assigned. If name is not specified, or the return value is not used by another command, the return value is displayed on the next line of the screen.
Function	Specifies the name of the character function (see the following table).
(char-expr)	Specifies a quoted character or an expression specifying a character. The parentheses are required.

Discussion

Two classes of **character functions** exist: character classification and character transformation.

The character classification functions return a boolean data type with the value non-zero (true) or zero (false). These functions take a single argument (char-expr) that must be compatible with the int4 data type. The character classification functions include:

isalpha	isupper	islower	isidit
isxdigit	isspace	ispunct	isprint
isctrl	isascii	isalnum	

The character transformation functions return an int4 containing an ASCII-coded value. These functions take a single argument that must be compatible with the int4 data type. The character transformation functions include: toupper, tolower, and toascii.

Descriptions for all character functions are listed in the table below.

Character Functions

Function	Discussion
isalpha	The isalpha function returns true when char-expr is alphabetic. The hexadecimal values for these characters are 41 through 5a (A . . . Z) and 61 through 7a (a . . . z).
isupper	The isupper function evaluates to true when char-expr is an uppercase alphabetic character. The hexadecimal values for these characters are 41

	through 5a (A . . . Z).
islower	The islower function evaluates to true when char-expr is a lowercase alphabetic character. The hexadecimal values for these characters are 61 through 7a (a . . . z).
isdigit	The isdigit function evaluates to true when char-expr is a numeric digit. The hexadecimal values for these characters are 30 through 39 (0 . . . 9).
isxdigit	The isxdigit function evaluates to true when char-expr is a hexadecimal digit. The hexadecimal values for these characters are 30 through 39 (0 . . . 9), 41 through 46(A . . . F), and 61 through 66 (a . . . f).
isalnum	The isalnum function evaluates to true when char-expr is alphanumeric. The hexadecimal values for these characters are 41 through 5a (A . . . Z), 61 through 7a (a . . . z), and 30 through 39 (0 . . . 9).
isspace	The isspace function evaluates to true when char-expr is a blank. This blank can be a single space (hexadecimal value 20), carriage return, line feed (new line or "\n"), tab ("\t"), vertical tab, or form feed (new page or "\p").
ispunct	The ispunct function evaluates to true when char-expr is a punctuation mark (neither a control nor an alphanumeric character). The hexadecimal values for these characters are 21 through 2f, 3a through 40, 5b through 60, and 7b through 7e.
isprint	The isprint function evaluates to true when char-expr is a printable character. The hexadecimal values for these characters are 20 through 7e.
iscntrl	The iscntrl function evaluates true when char-expr is a delete character (hexadecimal 7f) or any control character (hexadecimal 0 through 1f).
isascii	The isascii function evaluates to true when char-expr is a coded value (hexadecimal 0 through 7f).
toupper	The toupper function returns the uppercase value of char-expr. If char-expr does not contain a lowercase letter, the result is the original char-expr, unchanged. The char-expr itself is not changed.
tolower	The tolower function returns the lowercase value of char-expr. If char-expr does not contain a uppercase letter, the result is the original char-expr, unchanged. The char-expr itself is not changed.
toascii	The toascii function clears all bits of char-expr that are not part of a standard ASCII character and returns this value. The char-expr itself is not changed.

Examples

1. This example demonstrates using character classification functions.

Command input:

```
define char cvar = 'a'
define int4 ivar
ivar = cvar
ivar
```

Result:

```
00000061H
```

Command input:

```
isalpha(cvar)
```

Result:

TRUE

Command input:

isalpha(ivar)

Result:

TRUE

Command input:

define int4 answer = isalpha(cvar)
answer

Result:

00000001H

Command input:

cvar

Result:

'a'

Command input:

isupper(cvar)

Result:

FALSE

Command input:

islower(cvar)

Result:

TRUE

Command input:

cvar = 'a'
isupper(cvar)

Result:

TRUE

Command input:

isdigit(cvar)

Result:

FALSE

Command input:

isxdigit (cvar)

Result:

TRUE

Command input:

isalnum(cvar)

Result:

TRUE

Command input:

isspace(cvar)

Result:

FALSE

Command input:

ivar = 20H

isspace(ivar)

Result:

TRUE

Command input:

cvar = '!'

ispunct(cvar)

Result:

TRUE

Command input:

isprint(ivar)

Result:

TRUE

Command input:

cvar = 5

cvar

Result:

'\005'

Command input:

isprint(cvar)

Result:

FALSE

Command input:

iscntrl(cvar)

Result:

TRUE

Command input:*isascii(cvar)***Result:***TRUE*

2. This example demonstrates character transformation functions.

Command input:

```
define int4 ivar = 5
define char cvar
cvar = toascii(ivar)
cvar
```

Result:*'\005'***Command input:**

```
cvar = toascii(61H)
cvar
```

Result:*'a'***Command input:***toascii(cvar)***Result:***00000061H***Command input:***toupper(cvar)***Result:***00000041H***Command input:***cvar***Result:***'a'***Command input:**

```
cvar = toupper(cvar)
cvar
```

Result:*'A'***Command input:**

```
ivar = 41H
cvar = tolower(ivar)
cvar
```

SourcePoint 7.7.1

Result:

'a'

clock

Return the elapsed time (in ms) since SourcePoint started.

Syntax

```
[variable = ] clock ()
```

Where:

variable is an ord4 variable

Discussion

The clock function returns the elapsed time (in ms) since SourcePoint was started. The return value can be assigned to an ord4 variable, or displayed on the command line.

Example 1**Command Input:**

```
clock()
```

Result:

```
0000F124H
```

Example 2

To measure the elapsed time of an operation:

Command Input:

```
define ord4 startTime = clock()
(some operations...)
printf ("elapsed time = %.3f seconds\n", (clock() - startTime) / 1000.0)
```

Result:

```
elapsed time = 3.2 seconds
```

Continue Command

The **continue** command transfers control from within a control block to the end of the block.

Syntax

```
continue
```

Discussion

Use the **continue** control construct to cause a jump to the end of the immediately enclosing iteration statement (**while**, **do**, or **for**).

Example

This example shows a **continue** control construct within an **if** control construct that is nested in a **for** loop. The variable *I* contains the amount of numbers between 0 and 12 whose modulus equals 2.

Command input:

```
define int2 a
define int2 I = 0
for (a = 0; a <= 12; a += 1)
{
  if ((a % 3) != 2)
    continue
  I = I + 1
}
I
```

Result:

```
0006H
```

Command input:

```
a
```

Result:

```
0013H
```

Cos Command

The **cos** command returns the cosine of a radian expression.

Syntax

[name =] cos (expr)

Where:

- name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
- (expr) Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Cpubreak, Cpuremove, Cpudisable, Cpuenable Commands

These commands are used to set, clear, display, enable and disable processor breakpoints.

Syntax

`cpubreak`

`cpubreak = [sts-spec,] type-spec [, processor-spec]`

`cpuremove [all]`

`cpuremove = { type-spec | processor-spec } [,...]`

`cpuenable = { type-spec | processor-spec } [,...]`

`cpudisable [all]`

`cpudisable = { type-spec | processor-spec } [,...]`

Where:

processor-spec	<code>p[rocessor] = { n all }</code> - where n is processor number
sts-spec	<code>{ e[nabled] d[isabled] }</code>
type-spec	<code>{ s[mm entry] smm ex[it] ins[truction set transfer] }</code>

Discussion

The **cpubreak** command sets and displays processor breakpoints (processor breaks). **Cpubreak** with no arguments displays a list of the current processor breaks.

The **cpuremove** command removes any or all of the processor breaks. Arguments to this command qualify which processor breaks are to be removed. For instance, `cpuremove = p=0`, removes all processor breaks associated with processor 0. **Cpuremove** with no arguments removes all processor breaks.

The **cpuenable** command selectively enables processor breaks. Arguments to this command qualify which processor breaks are to be affected. For instance, `cpuenable = p=1` enables only processor breaks associated with processor 1.

The **cpudisable** command selectively disables processor breaks. Arguments to this command qualify which processor breaks are to be affected. For instance, `cpudisable = smm exit`, disables only processor debug breaks with the type set to smm exit. If no arguments are specified, all processor breaks are disabled.

Processor breaks can also be set, displayed, etc. from the **Breakpoints** window.

Examples

Command inputs:

```
cpubreak // display all processor breaks
cpubreak = ins // set break on instruction set transfer
cpubreak = smm // set break on smm entry for processor
entry, p=1
cpubreak = smm exit, // set break on smm exit for all processors
p=all
cpuremove // remove all processor breaks
cpuremove = ins // remove the instruction set transfer processor break
cpuremove = p=1 // remove all processor breaks associated with processor 1
cpuenable = smm // enable any processor breaks with type of smm entry
entry
cpudisable // disable all processor breaks
cpudisable = ins // disable instruction set transfer processor break
```

cpuid_eax

Execute the CPUID assembly instruction and return the value in EAX.

Syntax:

[variable=] [[px]] cpuid_eax [(eax[,ecx])]

Where:

- [px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
- [eax] is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
- [ecx] is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
- variable is an ord4 variable to receive the value of EAX.

Discussion

Execute the CPUID instruction with the specified values of EAX and ECX. The return value (EAX) can be assigned to a debug variable, or displayed on the command line.

Example 1

To run cpuid_eax on the viewpoint processor with EAX=1 and display the value obtained in EAX:

Command input:

```
cpuid_eax
```

Result:

```
00000000
```

Example 2

To run cpuid_eax on the viewpoint processor with EAX=10 and display the value obtained in EAX:

Command input:

```
cpuid_eax(10)
```

Result:

```
00000000
```

Example 3

To run `cpuid_eax` on the viewpoint processor with `EAX=10` and `ECX=5` and store the result obtained in `EAX` to a variable:

Command input:

```
define ord4 o4cpuideax = cpuid_eax(10,5)  
o4cpuideax
```

Result:

```
00000000
```

cpuid_ebx

Execute the CPUID assembly instruction and return the value in EBX.

Syntax:

[variable=] [[px]] cpuid_ebx [(eax[,ecx])]

Where:

- [px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
- [eax] is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
- [ecx] is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
- variable is an ord4 variable to receive the value of EBX.

Discussion

Execute the CPUID instruction with the specified values of EAX and ECX. The return value (EBX) can be assigned to a debug variable, or displayed on the command line.

Example 1

To run cpuid_ebx on the viewpoint processor with EAX=1 and display the value obtained in EBX:

Command input:

```
cpuid_ebx
```

Result:

```
00000000
```

Example 2

To run cpuid_ebx on the viewpoint processor with EAX=10 and display the value obtained in EBX:

Command input:

```
cpuid_ebx(10)
```

Result:

```
00000000
```


Example 3

To run `cpuid_ebx` on the viewpoint processor with `EAX=10` and `ECX=5` and store the result obtained in `EBX` to a variable:

Command input:

```
define ord4 o4cpuidebx = cpuid_ebx(10,5)  
o4cpuidebx
```

Result:

```
00000000
```

cpuid_ecx

Execute the CPUID assembly instruction and return the value in ECX.

Syntax:

[variable=] [[px]] cpuid_ecx [(eax[,ecx])]

Where:

- [px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
- [eax] is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
- [ecx] is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
- variable is an ord4 variable to receive the value of ECX.

Discussion

Execute the CPUID instruction with the specified values of EAX and ECX. The return value (ECX) can be assigned to a debug variable, or displayed on the command line.

Example 1

To run cpuid_ecx on the viewpoint processor with EAX=1 and display the value obtained in ECX:

Command input:

```
cpuid_ecx
```

Result:

```
00000000
```

Example 2

To run cpuid_ecx on the viewpoint processor with EAX=10 and display the value obtained in ECX:

Command input:

```
cpuid_ecx(10)
```

Result:

00000000

Example 3

To run `cpuid_ecx` on the viewpoint processor with `EAX=10` and `ECX=5` and store the result obtained in `ECX` to a variable:

Command input:

```
define ord4 o4cpuidecx = cpuid_ecx(10,5)
o4cpuidecx
```

Result:

00000000

cpuid_edx

Execute the CPUID assembly instruction and return the value in EDX.

Syntax:

[variable=] [[px]] cpuid_edx [(eax[,ecx])]

Where:

- [px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
- [eax] is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
- [ecx] is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
- variable is an ord4 variable to receive the value of EDX.

Discussion

Execute the CPUID instruction with the specified values of EAX and ECX. The return value (EDX) can be assigned to a debug variable, or displayed on the command line.

Example 1

To run cpuid_edx on the viewpoint processor with EAX=1 and display the value obtained in EDX:

Command input:

```
cpuid_edx
```

Result:

```
00000000
```

Example 2

To run cpuid_edx on the viewpoint processor with EAX=10 and display the value obtained in EDX:

Command input:

```
cpuid_edx(10)
```

Result:

```
00000000
```

Example 3

To run `cpuid_edx` on the viewpoint processor with `EAX=10` and `ECX=5` and store the result obtained in `EDX` to a variable:

Command input:

```
define ord4 o4cpuidedx = cpuid_edx(10,5)  
o4cpuidedx
```

Result:

```
00000000
```

cscfg and local_cscfg

Display or change the contents of configuration registers in a chipset device through an ITP debugger interface.

Access		Status	Addressing
device	Mechanism		
	JTAG	Error Status	
TBG_CORE		Error bits set if address given does not map to a valid device address. Bus and Device fields ignored for address mapping error test.	4k configuration space, plus memory mapped addressing. Has transaction-type field, it can be 0 or 1.
	JTAG	Error bit	4k configuration space, no memory mapped addressing. No transaction-type field. If system transaction flow results in results in a stuck or hung PCI bus access, the cscfg command can also be hung. Use the local_cscfg command with the same syntax to work around such hangs.
TNB		Error bit has known problem in certain register address ranges. DE recommends ignoring it. ITP does not test it.	
	JTAG	Error bit	4k configuration space, no memory mapped addressing. No transaction-type field. Does not support the JCONFIG instruction register. ITP maps both cscfg and local_cscfg to the JCONFL instruction register.
XMB		Error bit set if address given does not map to a valid device address. Bus and Device fields ignored for address mapping error test.	
	JTAG	Error bit	4k configuration space, no memory mapped addressing. Has Transaction-type field but it is always 0.
PXH		Error bit not implemented, always 0	
	JTAG	Error bit	4k configuration space, plus memory mapped addressing. Has Transaction-type field, it can be 0 or 1.
LH MCH (Cayuse)		Error bit not implemented, always 0	

Syntax:

Enter the command with required parameters, which must be in parentheses, to read the contents of a register. Add the assignment operator (=) and an expression to the command sequence to write to the register. Use the command local_cscfg instead of cscfg, with the same syntax, if the device supports it and if the register to access is within the device (the register must not need to be forwarded to another device). The local_cscfg form of the command is only

available on certain devices. See the table above to determine whether the device has local_cscfg capability.

If the command entered has missing or extra parameters for the device specified in the device_number field, an error will be displayed.

This command may be issued while the processor is running, accesses are asynchronous to any system software operations. The user is expected to resolve JTAG and system software access synchronization issues.

The user is expected to refer to the device component specification configuration registers to determine which PCI bus/device/function/offset or linear/mapped/test register address to issue when accessing a particular device, and to understand how each device may respond to invalid register access address input. For example, given an invalid register address, some devices return all 0's in the data field while others return a JTAG error bit. If an error is returned by the chipset, ITP will propagate this error back to the user.

For Tylersburg TBG_CORE device:

```
cscfg(device_number, transaction_type, register_address, pci_bus_number, pci_device_number,
pci_function_number, byte_enab)[=expr]
```

Where:

device_number	is the ITP Device ID (DID) or alias.
transaction_type	is an expression that evaluates to a 1-bit valid transaction access type: value of 0b specifies the configuration space registers type and uses bus, device, function, and address; value of 1b specifies the memory mapped registers type and uses region and address.
register_address	is an expression that evaluates to a 12-bit valid chipset configuration space address. For a memory mapped transaction type, this field evaluates to a 32-bit memory mapped address. The high order 8 bits specify the region (MEM_HIGH higher nibble , MEM_LOW lower nibble) and the lower 24 bits specify the address.
pci_bus_number	is an expression that evaluates to an 8-bit value, representing the PCI Bus Number for the configuration space register. Pass the value 0 for this parameter when using the memory mapped transaction type.
pci_device_number	is an expression that evaluates to a 5-bit value, representing the PCI Device Number for the configuration space register. Pass the value 0 for

	this parameter when using the memory mapped transaction type.
pci_function_number	is an expression that evaluates to a 3-bit value, representing the PCI Function Number for the configuration space register. Pass the value 0 for this parameter when using the memory mapped transaction type.
byte_enable	is an expression that evaluates to a 2-bit value, representing the Byte Enable field of the JTAG data register used for PCI configuration space access. For reads, if this value is not the correct read operation code (00b), it will automatically be set to 00b.
expr	is an expression that evaluates to a 32-bit value, for the data field for the PCI configuration space register.

For Lindenhurst MCH, PXH, Twincastle TNB, XMB, Blackford BNB, Clarksboro CNB, Seaburg SNB, San Clemente SCNB, GB, ESB2, Tolapai TLP devices:

```
cscfg(device_number, [transaction_type,] register_address, pci_bus_number,
pci_device_number, pci_function_number, byte_enab)[=expr]
```

Where:

device_number	is the ITP Device ID (DID) or alias.
transaction_type (Lindenhurst MCH, PXH only)	is an expression that evaluates to a 2-bit valid transaction access type: value of 00b specifies the configuration space registers type and uses bus, device, function, and address; value of 01b specifies the memory mapped registers type and uses region and address.
register_address	is an expression that evaluates to a 12-bit valid chipset configuration space address. For a memory mapped transaction type, this field evaluates to an 18-bit memory mapped address. The high order 6 bits specify the region and the lower 12 bits specify the address.
pci_bus_number	is an expression that evaluates to an 8-bit value, representing the PCI Bus Number for the configuration space register. Pass the value 0 for this parameter when using the memory mapped transaction type.
pci_device_number	is an expression that evaluates to a 5-bit value, representing the PCI Device Number for the configuration space register. Pass the value 0 for this parameter when using the memory mapped transaction type.
pci_function_number	is an expression that evaluates to a 3-bit value, representing the PCI Function Number for the configuration space register. Pass the value 0 for this parameter when using the memory mapped transaction type.
byte_enable	is an expression that evaluates to a 2-bit value, representing the Byte Enable field of the JTAG data register used for PCI configuration space access. For reads, if this value is not the correct read operation code (00b), it will automatically be set to 00b.
expr	is an expression that evaluates to a 32-bit value, for the data field for

the PCI configuration space register.

For 82870 devices:

```
cscfg(device_number, reg_addr, bus_number, dev_function, byte_enab)[=expr]
```

Where:

device_number is the ITP Device ID (DID) or alias.

reg_addr is an expression that evaluates to a valid chipset PCI configuration space register address.

bus_number is an expression that evaluates to an 8-bit value, representing the PCI Bus Number for the configuration space register.

dev_function is an expression that evaluates to an 8-bit value, which is a combined field for PCI Device and PCI Function for the configuration space register. The lower 3 bits are for the Function field, and the upper 5 bits are for the Device field.

byte_enable is an expression that evaluates to a 2-bit value, representing the Byte Enable field of the JTAG data register used for PCI configuration space access. See discussion for more details on this parameter. For reads, if this value is not the correct read operation code (00b), it will automatically be set to 00b.

expr is an expression that evaluates to a 32-bit value, for the data field for the PCI configuration space register.

Discussion

JTAG:

Use the cscfg command for data transfers using JTAG access to chip-set device PCI configuration register space. This command can be used to configure any PCI configuration space registers that are accessible via the JTAG PCI config access register, usually this includes the full register space of each chipset device. The chipset device specification defines details and limitations. When reading the configuration register, a 32-bit data value is displayed or returned. For TNB, use the cscfg to access PCI configuration registers both on the device and directed to another device in the system, such as the PXH.

The byte_enable parameter specifies the first two bits of the 3-bit field: 00b = dword (read), 01b = byte (write), 10b = word (write), and 11b = dword (write). The third bit of the field is set when the

command is issued without an assignment (read syntax) and reset when the command is issued with an assignment (write syntax). NOTE: ITP will not issue a warning if a write byte_enable is specified during a read or if a read byte_enable is specified during a write.

The JTAG PCI Configuration access register has a "busy" bit which ITP polls when accessing registers. If the access times out, ITP will issue an error. See the ITP help for ini file timeouts for details on length of timeout or changing the default value.

Example 1

82870 devices:

To read/modify some chip-set registers using the ITP command language features:

Command input:

```
#define SNC_DEV_VID_OFFSET 0
```

```
#define SIOH_DEV_VID_OFFSET 0
```

```
#define SNC_SPAD_OFFSET 0xc4
```

```
#define SIOH_SPAD_OFFSET 0xb0
```

```
#define SNC_PCI_DEV_NUM 0x18 /* assignment on Tiger platform */
```

```
#define SIOH_PCI_DEV_NUM 0x1C /* assignment on Tiger platform */
```

```
#define SNC_SPAD_FCN 0
```

```
#define SIOH_SPAD_FCN 0n5
```

```
#define SNC_PLAIN_DEV_FCN (SNC_PCI_DEV_NUM * 0x8) /* initially, C0 */
```

```

#define SIOH_PLAIN_DEV_FCN (SIOH_PCI_DEV_NUM * 0x8) /* initially, E0 */

#define SNC_SPAD_DEV_FCN ((SNC_PCI_DEV_NUM * 0x8) + SNC_SPAD_FCN)
#define SIOH_SPAD_DEV_FCN ((SIOH_PCI_DEV_NUM * 0x8) + SIOH_SPAD_FCN)

#define TIGER_BUS_NUM 0xff

#define SNCM_DEV_VID_VALUE 0x05008086
#define SIOH_DEV_VID_VALUE 0x05108086

#define BYTE_WRITE 1
#define WORD_WRITE 0x2
#define DWORD_WRITE 0x3
#define DWORD_READ 0

```

Result:

```

// get the read-only DEVICE and VENDOR i.d. values, check them

o4Result = cscfg(SNCM0, SNC_DEV_VID_OFFSET, TIGER_BUS_NUM,
SNC_PLAIN_DEV_FCN, DWORD_READ)

if (SNCM_DEV_VID_VALUE != o4Result)
{
    printf("Bad dword value from readonly SNCM0 Device/Vendor ID register\n")

    printf(" Expected: %08D   Actual: %08D\n", SNC_DEV_VID_VALUE, o4Result)
}

```

```
// write, read the SNC device scratch-pad register

cscfg(SNCM0, SNC_SPAD_OFFSET, TIGER_BUS_NUM, SNC_SPAD_DEV_FCN,
DWORD_WRITE) = 0xdeadbeef

o4Result = cscfg(SNCM0, SNC_SPAD_OFFSET, TIGER_BUS_NUM, SNC_SPAD_DEV_FCN,
DWORD_READ)

if (0xdeadbeef != o4Result)

{

    printf("Bad dword read or write from SNCM0 scratch-pad register\n")

    printf(" Expected: %08D   Actual: %08D\n", 0xdeadbeef, o4Result)

}
```

Example 2

82870 device:

To show an error caused by the device_number being larger than the available devices in the current ITP scan chain:

Command input:

```
[p0]> cscfg( num_jtag_devices+1, 1, 1, 0x78, 0 )
```

ERROR #6418

Result:

NMI: Device was not found

Example 3

Using ITP devicelist command to list device aliases available:

Use the devicelist command to view aliases of all the available devices in the current ITP scan chain. The example scan chain shows an unlikely system configuration to illustrate all the Twincastle and Lindenhurst device names.

Command input:

```
[> devicelist
```

Result:

Brd	DP	CP	Device	Alias	Idcode-Step	Type		
		ThreadId	IsEnabled					
0	0	0	PRES0	PRES00	08304013-A0	PRESCOTT	0	
	Yes							
0	0	1	PRES1	PRES10	08304013-A0	PRESCOTT	1	
	Yes							
0	0	2	TNB	TNB0	0a100013-A0	TNB	0	Yes
1	1	3	XMB	XMB0	0a101013-A0	XMB	0	Yes
1	1	4	PXH	PXH0	00500013-A0	PXH	0	
	Yes							
2	2	5	PRES0	PRES01	08304013-A0	PRESCOTT	0	
	Yes							
2	2	6	PRES1	PRES11	08304013-A0	PRESCOTT	1	
	Yes							
2	2	7	LHMCHPFLHMCHPF0		01102013-A0	LHMCHPF	0	Yes
2	2	8	TWMCHWSTWMCHWS0		01105013-A0	TWMCHWS	0	Yes
2	2	9	LHMCH4PLHMCH4P0		01104013-A0	LHMCH4P	0	Yes

2	2	10	LHMCHMRLHMCHMR0	01103013-A0	LHMCHMR	0	Yes
3	3	11	PXH PXH1	00500013-A0	PXH	0	

Yes

Example 4

Another use of the devicelist command to view aliases of all the available JTAG devices in the current ITP scan chain:

Command input:

```
[.]> define ord1 i  
[.]> for (i=0; i< num_jtag_devices; i++)  
[.]> {  
[.]> devicelist[i].alias  
[.]> }
```

Result:

PRES00

PRES10

TNB0

XMB0

PXH0

PRES01

PRES11

LHMCHPF0

TWMCHWS0

LHMCH4P0

LHMCHMR0

PXH1

Example 5

Reading register 0x10 on bus 0, device 1, function 2, Lindenhurst device, using bus/dev/function addressing:

Command input:

```
[ ]> cscfg(LHMCHPF0, 0, 0x10, 0, 1, 2, 0)
```

Example 6

Reading a register at 0x3fff on Lindenhurst device, using memory-mapped addressing:

Command input:

```
[ ]> cscfg(LHMCHPF0, 1, 0x3fff, 0, 0, 0, 0)
```

Example 7

Reading register 0x10 on bus 0, device 1, function 2, TNB device, using bus/dev/function addressing:

Command input:

```
[ ]> cscfg(TNB0, 0x10, 0, 1, 2, 0)
```

Example 8

Reading the same register on both channels of a PXH device, using bus/dev/function addressing:

Command input:

```
[]> csdebugchain(PXH0) = 0x40    // force accesses to A side
```

```
[]> cscfg(PXH0, 0x10, 0, 1, 2, 0) // access register
```

```
[]> csdebugchain(PXH0) = 0        // switch accesses to B side
```

```
[]> cscfg(PXH0, 0x10, 0, 1, 2, 0) // access same register, other side
```

csdebugchain

dbport

csr

Read/write control bus registers for Uncore devices.

Syntax

[variable=] csr([DID,]reg) [= value]

Where:

reg is an expression that evaluates to a valid control bus register number.
 value is an expression that evaluates to a 32 bit value to load in reg.
 variable is a debug variable to hold the return value.
 DID is the device ID of the target device of this command.

Discussion

This function allows access to the Control Status Registers for Uncore devices. The csr function will also perform a CSRACCESSREAD on read operations in order to scan out the control register contents. During writes, the csr function only executes a CSRACCESS IR/DR scan. The csr function returns a 32 bit value. If a core device is selected, the function will automatically perform the operation on the core device's related Uncore device. If the device selected is not a core device, the function will check and make sure it is; otherwise an error message will be displayed.

Example 1

Command input:

```
csr(0x1001) = 0x12340000
csr(0x1001)
```

Result:

```
12340000H
```

Example 2

Command input:

```
csr(0, 0x1001) = 0x12345678
csr(0, 0x1001)
```

Result:

```
12345678H
```

Example 3

Command input:

SourcePoint 7.7.1

```
define ord4 ord4val = csr(0x1001)
```

Example 4

Command input:

```
define ord4 ord4val = csr(2,0x1001)
```

Ctime Command

The **ctime** command converts the output of the **time** command into a null-terminated ASCII string. The input expression is a value (such as one returned by the **time** function). The output string has the form "day month date hh:mm:ss year\n", where\n is the new-line escape character.

Syntax

```
[ name = ] ctime (expr)
```

Where:

name	Specifies an nstring variable to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
(expr)	Specifies a number or an expression. The parentheses are required.

Example

The following example illustrates the **ctime** function:

Command input:

```
define ord4 now
now = time ( )
ctime (now)
```

Result:

```
Wed Jun 07 16:26:07 2000 //answer will be current day' and time
```

cwd

Set or display the current working directory

Syntax

`cwd [pathname]`

Discussion

The current working directory specifies where include and list (log) files are to be found. The `cwd` command without an argument displays the current working directory.

Example 1

Command input*:

cwd

Result:

Program Files\American Arium\SourcePoint

Command input:

*cwd macro
cwd*

Result:

Program Files\American Arium\SourcePoint\Macro

Command input:

*cwd aa
cwd*

Result:

AA

*Command inputs are cumulative; you must start at the first input to get the results shown from subsequent inputs.

Dataqualmode, Dataqualstart Commands

The **dataqualmode** command is used in setting or displaying bus analyzer data qualification specifications. The **dataqualstart** command specifies the types of bus cycles or start/stop bus cycle types to record in trace.

Syntax

`dataqualmode [= record-spec]`

`dataqualstart [= enabled | disabled]`

Where record-spec is one of the following:

- off
- record on Q1
- record on Q1, Stop on Q2
- record only BTMs
- record BTMs and Q1
- record BTMs and Q1, Stop on Q2

Discussion

The **dataqualmode** command is used to set or display the bus analyzer data qualification specifications. The **dataqualstart** command enables/disables these specifications to execute. Q1 and Q2, bus cycle types, are set via the **Breakpoints** window or using the **busbreak** command. BTM refers to Branch and Trace Messages.

Examples

Command inputs:

<code>dataqualmode</code>	<i>query the current data qualification mode</i>
<code>off</code>	<i>qualification mode displayed</i>
<code>dataqualmode = record on Q1, Stop on Q2</code>	<i>set data qualification mode.</i>
<code>dataqualmode</code>	<i>query the current data qualification mode</i>
<code>record on Q1, Stop on Q2</code>	<i>qualification mode displayed</i>
<code>dataqualstart = enabled</code>	<i>set qualification enabled</i>
<code>dataqualstart</code>	<i>query enabled or disabled</i>
<code>enabled</code>	<i>display enabled or disabled</i>

Dbgbreak, Dbgremove, Dbgdisable, Dbgenable Commands

These commands are used to set, clear, display, enable and disable debug register breakpoints.

Syntax

dbgbreak

dbgbreak = [sts-spec,] execute, location-spec [, processor-spec]

dbgbreak = [sts-spec,] execute in smm, location-spec [, processor-spec]

dbgbreak = [sts-spec,] data access, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] data access in smm, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] data read, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] data write, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] data write in smm, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] i/o access, location-spec, size-spec [, processor-spec]

dbgbreak = [sts-spec,] i/o access in smm, location-spec, size-spec [, processor-spec]

dbgremove [all]

dbgremove = {type-spec | location-spec | size-spec | processor-spec} [,...]

dbgenable = {type-spec | location-spec | size-spec | processor-spec} [,...]

dbgdisable [all]

dbgdisable = {type-spec | location-spec | size-spec | processor-spec} [,...]

Where:

sts-spec	{ e[nabled] d[isabled] }
type-spec	{ e[xecute] data a[ccess] data read data w[rite] i/o access execute in smm data access in smm data write in smm i/o access in smm }
processor-spec	p[rocessor] = { 0 1 2 3 All }
location-spec	l[ocation] = vls-addr
size-spec	s[ize] = { b[yte] w[ord] d[word] }

Discussion

The **dbgbreak** command sets and displays debug register breakpoints (debug breaks). **Dbgbreak** with no arguments displays a list of the current debug breaks. There are four debug register breakpoints available.

The **dbgremove** command removes any or all of the debug breaks. Arguments to this command qualify which debug breaks are to be removed. For instance, **dbgremove=data write, s=byte**, removes all debug breaks with the type set to data write and size set to byte . **Dbgremove** with no arguments removes all debug breaks.

The **dbgenable** command selectively enables debug breaks. Arguments to this command qualify which debug breaks are to be affected. For instance, **dbgenable=execute** enables only debug breaks with the type set to execute.

The **dbgdisable** command selectively disables debug breaks. Arguments to this command qualify which debug breaks are to be affected. For instance, **dbgdisable=execute** disables only debug breaks with the type set to execute. If no arguments are specified, all debug breaks are disabled.

Debug breaks can also be set, displayed, etc. from the **Breakpoint** window.

Examples

Command inputs:

<i>dbgbreak</i>	<i>// display current debug breaks</i>
<i>dbgbreak = data access, location=1000:1234, size=word</i>	<i>// set a debug break on a word access at location 1000:1234</i>
<i>dbgbreak = data read, location=1000p, size=word</i>	<i>// set a debug break on a word read at address 1000p</i>
<i>dbgremove</i>	<i>// remove all debug breaks</i>
<i>dbgremove = data write, size=byte</i>	<i>// remove all debug breaks with type set to data write and size set to bytes</i>
<i>dbgremove = i/o access</i>	<i>// remove all I/O debug breaks</i>
<i>dbgdisable</i>	<i>// disable all debug breaks</i>
<i>dbgdisable = execute</i>	<i>// disable all debug breaks with type set to execute</i>
<i>dbgenable = loc=1000:1234</i>	<i>// enable all debug breaks with location set to 1000:1234</i>

Note: Data read applicable to IA-64 processors only.

defaultpath

Set or display the current working directory.

Syntax

[variable=] defaultpath = newpath

Where:

variable	is an nstring variable to receive the answer.
newpath	is an nstring variable or string constant specifying a new working directory.

Discussion

The current working directory (default path) specifies where **include** and **list** (log) files are to be found.

Example 1

To display the current working directory:

Command input:

defaultpath

Result:

C:\Program Files\American Arium\SourcePoint

Example 2

To set the current working directory:

Command input:

defaultpath = C:\temp
defaultpath

Result:

C:\temp

Example 3

To assign the current working directory to a debug variable:

Command input:

define nstring strpath = defaultpath
strpath

SourcePoint 7.7.1

Result:

C:\Program Files\American Arium\SourcePoint

Define Command

The **define** and **redefine** commands create debug objects.

Syntax

```
define {literally name = "string"  
| [global] data type name [= expr]}
```

Where:

literally	Creates another name or alias for a reserved word, command, or debug object.
name	Specifies a unique, user-defined name for the object being defined.
"string"	Specifies the string of characters replaced by a name. The quotation marks are required to delimit the string.
global	Indicates that a debug variable is globally recognized. Data types are global unless defined inside a debug procedure.
data-type	Specifies an emulator data type.
expr	An expression that assigns an initial value to the object.

Discussion

Use the **define** command to create debug objects. Debug objects customize the debugging environment. Debug objects include literally definitions, debug variables, and debug procedures (procs). A debug object name cannot be the same as a reserved keyword in the command language. If the name specified is the same as a previously defined debug object, then that object is overwritten.

Optionally, you can assign an initial value to a debug variable. If no initial value is specified, the variable is, depending on its data type, a:

- numeric value of zero
- string of zero length
- boolean set to "false"
- pointer set to "invalid"

Arrays of debug variables can also be created. For more information see "[Array Data Types](#)," part of "Command Line Overview," found under, *Command Line Window*.

Examples

1. The following examples show how to use the literally option to abbreviate commands.

Command input:

```
define literally def = "define"
```

Command input:

```
def literally lit = "literally"
```

Note: The **#define** command can also be used to define aliases.

- The following example shows how to use the **literally** option to abbreviate a series of commands that show the processor flags register value in binary.

Command input:

```
define literally def = "define"
def literally lit = "literally"
def lit spf = \
"base = 2t ; eflags ; base = 16t"
spf // execute the command abbreviation
```

Result:

```
11111111111111111001000000000101010Y // result will vary
```

- The following example shows how to define debug variables.

Command input:

```
define ord1 zero = 0
define int2 max = 400
zero max max = zero max
```

Result:

```
00H
```

Command input:

```
max
max = zero
max
```

Result:

```
0400H
```

- The following example shows how to use the **proc** command to define a procedure named "power." This proc returns the result of a value and its exponent.

Command input:

```
base = 10t
define proc power (arg1, arg2)
define int1 arg1
define int1 arg2
{
define int1 index
define ord4 result = 1
for (index = 1 ; index <= arg2 ; index += 1)
result = result * arg1
return (result)
}
power (2t,4t) // Execute the proc
```

Result:
16T

DefineMacro Command

The **DefineMacro** function is used to assign macros (include files) to buttons on the **Macro** toolbar. (The **Macro** toolbar is enabled by selecting **View|Toolbars|Macro**).

Syntax

```
DefineMacro(id, filename[, echo, text])
```

Where:

id	An integer (0-9) indicating the toolbar button to assign. If a button already has an assigned macro, then the previous definition is overwritten.
filename	A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks.
echo	A boolean indicating whether the contents of the command file should be echoed to the Command window. If this argument is omitted, then the contents are not echoed.
text	A string indicating the text to assign to the toolbar button. The text is displayed only if Icons and text is selected (on the Macro toolbar context menu). This argument is optional.

Discussion

Macro (Include) files greatly speed up repetitive debug tasks. Assigning macros to toolbar buttons makes them even easier to use. A simple setup macro can be used to assign multiple toolbar buttons.

The **Macro** toolbar displays up to 10 user-definable buttons. By default, four are displayed. You can right-click on the toolbar, and select **Customize** to change the number of buttons displayed. If you assign a macro to a button not already displayed, then it is displayed automatically.

Examples

1. The following example shows how to assign the macro "c:\test\qa.mac" to the first button in the toolbar. The contents of this macro are echoed to the **Command** window when the button is clicked. The button is labeled **Run QA tests**.

Command input:

```
defineMacro(0, "c:\\test\\qa.mac", true, "Run QA tests")
```

2. The following example shows how to assign the macro "loop.txt" (found in the current working directory) to the 9th button in the toolbar (the index is 0 based). The contents of this macro are not echoed to the **Command** window. The button does not have a text label (icon only).

Command input:

```
defineMacro(8, "loop.txt")
```

3. The following example shows how to clear the macro definition of the first toolbar button.

Command input:
defineMacro(0, "")

devicelist

Display names and other device information of currently attached or configured devices.

Syntax

```
devicelist
[variable =] devicelist[expr]
[variable =] devicelist[expr].fieldname
```

Where:

[expr] is an expression evaluating to an ordinal value specifying the device ID (DID) (may also be a device alias; brackets are required punctuation).

fieldname is the specific devicelist field to be returned (see table below for details).

variable is a debug variable to receive the value returned.

Discussion

The devicelist command displays a summary of the current configuration of all the devices that are known to SourcePoint. Devices are listed in order by DID.

If a device ID (DID) is specified for expr, the devicelist command will only display data for the specified DID.

Use the devicelist[expr].<fieldname> commands to display or retrieve individual fields of the devicelist data described in the table below.

Devicelist Command Form	Displays or Returns	JTAG devicelist[o2did] data type	Other devicelist[o2did] data type	JTAG devicelist display	Other devicelist display
devicelist[o2did]	All information for the specified device (devicelist format)	String	String	N/A	N/A
devicelist[o2did].debugport	The debug port	Ordinal	Ordinal	Yes	Yes
devicelist[o2did].cp	The chain position	Ordinal	Error	No	No
devicelist[o2did].did	The device id	Ordinal	Ordinal	Yes	Yes
devicelist[o2did].tapport	The Tap port id	Ordinal	Error	Yes	No
devicelist[o2did].schain	The scan chain	Ordinal	Error	Yes	No
devicelist[o2did].Threadid	The thread/core ID	Ordinal	Error	Yes	No

	of this device				
devicelist[o2did].device	The device name	String	String	No	No
devicelist[o2did].alias	ITP-assigned alias for this device	String	String	Yes	Yes
devicelist[o2did].idcode	The idcode for JTAG device only	Ordinal	Error	Yes	No
devicelist[o2did].stepping	The device stepping	String	Error	Yes	No
devicelist[o2did].devicetype	The device type	String	String	Yes	Yes
devicelist[o2did].isenabled	The thread/device's enabled status	Boolean	Boolean	Yes	Yes
devicelist[o2did].bustype	The bus type used by the device	String	String	Yes	Yes

Each JTAG device has an associated idcode value, from which a stepping number (stepping) and device type (type) are derived. For devices that contain multiple threads or cores, each thread or core will be listed as a separate device with a zero-based thread/core ID distinguishing them. If a thread/core device can be disabled, the Enabled column will show its current state; otherwise it will always be enabled.

For all non-JTAG devices, Tap Port ID (TP), Scan Chain (SC), Stepping (Step), Idcode, and Thread/Core ID (T/C) of each device are not applicable so they are not displayed. Attempting to access these fields using fieldname will cause an error.

Example 1

To display output of devicelist for a two debug port (DP) target system with two physical processors, each with two enabled threads, — note that the Device ID (DID) value is always displayed as hexadecimal regardless of the base control variable setting:

Command input:

```
devicelist
```

Result:

DID	DP	TP	SC	Alias	Type	Step	Idcode	BusType	T/C
Enabled									

0	0	0	0	P0	YONAH	A0	0005d013	JTAG	0
Yes									
1	0	0	0	P1	YONAH	A0	0005d013	JTAG	1
Yes									
2	0	1	1	P2	YONAH	A0	0005d013	JTAG	0
Yes									
3	0	1	1	P3	YONAH	A0	0005D013	JTAG	1
Yes									

Example 2

To display devicelist data for a single device:

Command input :

```
devicelist[P2]
```

Result:

<i>DID</i>	<i>DP</i>	<i>TP</i>	<i>SC</i>	<i>Alias</i>	<i>Type</i>	<i>Step</i>	<i>Idcode</i>	<i>BusType</i>	<i>T/C</i>
<i>Enabled</i>									

2	0	1	1	P2	YONAH	A0	0005D013	JTAG	0
Yes									

Example 3

To display the idcode of a single device:

Command input:

```
devicelist[P2].idcode
```

Result:

```
08303013H
```

Example 4

To get the idcode of a single device and assign it to a debug variable:

Command input:

```
define ord4 o4MyIdCode
o4MyIdCode = devicelist[P2].idcode
o4MyIdCode
```

Result:

```
0005D013H
```

Example 5

To display just the aliases of all JTAG devices:

Command input:

```
define ord4 o4ID
for (o4ID = first_jtag_device; o4ID < num_jtag_devices; o4ID++)
```

```
{
  devicelist[o4ID].alias
}
```

Result:

```
P0
P1
P2
P3
```

Example 6

To create a custom-format devicelist command:

Command input:

```
define proc devlist()
{
  define ord4 o4ID
  for (o4ID=first_jtag_device; o4ID < num_jtag_devices; o4ID++)
  {
    printf("%2x %8s (%-11s) port %d\n",
      devicelist[o4ID].did,
      devicelist[o4ID].alias,
      devicelist[o4ID].devicetype,
      devicelist[o4ID].debugport)
  }
}
devlist
```

Result:

```
0  P0 (YONAH    ) port 0
1  P1 (YONAH    ) port 0
2  P2 (YONAH    ) port 0
3  P3 (YONAH    ) port 0
```

Displayflag Control Variable

This control variable determines if the value resulting from an assignment operation is to be displayed.

Syntax

`displayflag [= bool-cond]`

Where:

`bool-cond` Specifies a number or an expression that must evaluate to true (non-zero) or false (zero). The default is false.

Discussion

Use the **displayflag** control variable to control whether or not the value resulting after an assignment operation is displayed. If `bool-cond` is true, the results of assignment operations are displayed. If you enter the **displayflag** control variable without options, the current value is displayed.

Example

The following example demonstrates the effect of the **displayflag** control variable.

Command input*:

displayflag = true

Result:

TRUE

Command input:

*define byte a
a = 3*

Result:

*03H *. **

Command input:

*displayflag = false // Set to false, the result is not displayed
a = 5*

**Command inputs are cumulative; you must start at the first input to get the results shown from subsequent inputs.*

Dos Command

The command executes a DOS command.

Syntax

DOS [dos-command]

Where:

dos-command specifies any valid DOS command

Discussion

Use the **dos** command to execute a DOS command. When you enter the **dos** command without an argument, a DOS window is opened. Type exit to return to the emulator.

Text to be passed to the host operating system is expanded with the currently defined literally definitions. To suppress this literally substitution, enclose aliases in single quotes.

Note: This command was formerly called the "@" command.

Examples

Command input:

dos copy c:\tmp\test.list c:\save

Command input:

dos //open a DOS box

Do While Command

This control construct groups and conditionally executes emulator commands.

Syntax

```
do {commands} while (bool-cond)
```

Where:

<code>do</code>	Specifies the beginning of a command block.
<code>{commands}</code>	Specifies one or more emulator commands. The braces are required when you enter multiple commands.
<code>while (bool-cond)</code>	Specifies that the loop ends when <code>bool-cond</code> is false. The <code>bool-cond</code> option specifies a number or an expression that must evaluate to true (non-zero) or false (zero). The parentheses are required.

Discussion

Use the **do_while** control construct to define a loop that is executed at least once. The test for continued execution (evaluation of `bool-cond`) comes after the command (or group of commands) is executed. Always enclose the loop body `{commands}` in braces when there is more than one command. The commands are re-executed while the expression evaluates to true. The **include** command is not executable inside the **do_while** control construct.

Example

The following example shows how to display uppercase alphabetic characters using a **do_while** loop.

Command input:

```
define int4 a = 41h
define char c
do
{
c = toascii (a)
c
a += 1
}
while (a <= 5Ah)
```

Result:

```
'A'
'B'
.
.
.
'Y'
'Z'
```


Dport Command

The **dPort** command displays or changes the contents of a 32-bit I/O port.

Syntax

```
dport (io-addr) [= expr]
```

Where:

- [px] Viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain, where x is the number (0, 1, 2, or 3).
- io-addr Specifies a 16-bit address in the processor I/O space. The available io-addr range is 0 to 0ffffh. Parentheses are optional.
- expr Specifies a 32-bit number or expression. Using this option writes the data to the specified I/O port.

Discussion:

Use the **dPort** command to read from and write to the specified I/O port with the specified 32-bit data.

Examples

1. The following example shows how to assign a 32-bit value to a port and assign one port value to another.

Command input:

```
dport 88h = 87654321h  
dport 90h = dport 88h
```

The following example shows how to create a debug variable named portvar and assign a port value to it.

Command input:

```
define ord4 portvar  
portvar = dport 90 ; portvar
```

Result:

```
FFFFFFFFH
```


Edit Command

The **edit** command opens the file name specified for edit.

Syntax

`edit [proc] [filename]`

`editor [=nstring]`

Where:

`proc` indicates a user-defined debug procedure name follows

`filename` A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks

Discussion

The **edit** command opens the file name specified for edit. The default editor is "notepad.exe" but this can be overridden by specifying a different editor with the **editor** control variable. If the **proc** keyword is specified, then the file is automatically re-parsed (as if the user has typed "include nolist filename").

Note: If a relative path is specified for a file name, then the current working directory (specified with the **cwd** command) is prepended to the path.

Example

Command input:

```
editor="c:\vslick\win\vs.exe" //change editor to slick
cwd macro                    //change working directory to SourcePoint\macro
edit proc fileio.cmd         //edit fileio.cmd and re-parse on exit from editor
```

Editor Control Variable

The **editor** control variable specifies which editor is invoked with the **edit** command.

Syntax

```
editor [= "string"]
```

Where:

"string" Specifies the invocation string (optional path, invocation name, and invocation options) of an editor available on the host. The quotation marks are required.

Discussion

Use the **editor** control variable to specify which editor is invoked when you enter the **edit** command. Entering the **editor** control variable without options displays the current value.

Example

The following example shows how to specify an editor:

Command input:

```
editor="c:\vslick\win\vs.exe"
```

Eflags Command

The **eflags** command displays or modifies processor flags.

Syntax

```
eflags [= expr]
flag-name [= expr]
```

Where:

Expr	Specifies a number or an expression to be assigned to the flag bit(s).
Flag-name	Specifies one of the reserved words in the following table that names a flag. If flag-name is entered without an option, it displays the current value of the specified flag. Expr should evaluate to a one-bit value (0 or 1) except for IOP, which is two bits.

Discussion

Use the **eflags** command to display and modify all the flag values or enter the individual reserved word for each flag bit to display the value of that bit (see the EFLAGS Register bit-pattern table that follows).

You can also access bits 0 through 15 of the Eflags Register using the **flags** command.

Examples

1. The following example shows how to display the value of the flags in hexadecimal base.

Command input:

```
eflags
```

Result:

```
0fffc802aH
```

2. The following example shows how to change the number base to 2T(binary) and display the flags.

Command input:

```
base = 2T
eflags
```

Result:

```
11111111111111001000000000101010Y
```

3. The following examples show different ways to set or clear the trap flag.

Command input*:

tf

Result:

00010116H

Command input:

eflags = eflags | 100000000y

tf

Result:

00010116H

Command input:

tf = 0 ; tf

Result:

00000000H

Command input:

eflags.tf

Result:

FALSE

Command input:

eflags.tf=0

Result:

00000000H

Command input:

eflags.tf=eflags.tf|1y

Result:

00000000H

**Command inputs are cumulative; you must start at the first input to get the results shown from subsequent inputs.*

Emubreak, Emuremove, Emudisable, Emuable Commands

These commands are used to set, clear, display, enable and disable emulator breakpoints.

Syntax

emubreak

emubreak = [sts-spec,] {type-spec}

emuremove [all]

emuremove = {type-spec} [,...]

emuable = {type-spec} [,...]

emudisable [all]

emudisable = {type-spec} [,...]

Where:

sts-spec	sts-spec: { e[nabled] d[isabled] }
type-spec	type-spec { r[eset] }

Discussion

The **emubreak** command sets and displays emulator breakpoints (emulator breaks). **Emubreak** with no arguments displays a list of the current emulator breaks. The only type of emulator breakpoint currently supported is *reset*.

The **emuremove** command removes any or all of the emulator breaks. Arguments to this command qualify which emulator breaks are to be removed. **Emuremove** with no arguments removes all emulator breaks.

The **emuable** command selectively enables emulator breaks. Arguments to this command qualify which emulator breaks are to be affected.

The **emudisable** command selectively disables emulator breaks. Arguments to this command qualify which emulator breaks are to be affected. If no arguments are specified, all emulator breaks are disabled.

Emulator breaks can also be set, displayed, etc. from the **Breakpoints** window.

Examples

Command inputs:

```
emubreak           // display all emulator breaks  
emubreak = reset   // break when reset occurs  
emuremove          // remove all emulator breaks  
emuremove reset    // remove the reset emulator break  
emudisable         // disable all emulator breaks  
emudisable reset   // disable reset emulator break
```

Encrypt Command

The **encrypt** command encrypts an include (macro) file. The file can be executed normally with the include command, but the contents of the file are not readable.

Syntax

```
encrypt( input_file, output_file )
```

Where:

input_file	an nstring variable or string constant specifying the file to encrypt
output_file	an nstring variable or string constant specifying the encrypted file

Discussion

The **encrypt** command allows "include" files to be distributed to users without them being able to examine the contents of the file. This is sometimes required when proprietary code is used to unlock the debug capabilities of a device.

Example

Command input:

```
encrypt("c:\unlock.mac", "c:\secret.mac")  
include c:\secret.mac      // run the encrypted include file
```

error

Change the severity of a SourcePoint error.

Syntax

`error(error-number, severity)`

Where:

`error-number` is a SourcePoint error number.
`severity` is the new severity for the specified error number: `nodisplay`, `warning`, `severe`, `error`, `fatal`.

Use the `error` function to change the severity of a SourcePoint error to any of the 5 possible levels:

Discussion

Severity	Description
<code>nodisplay</code>	The error message will not be displayed in the Command window.
<code>warning</code>	The error message will be displayed as a warning but will not affect the execution of SourcePoint scripts.
<code>severe</code>	The error message will be displayed as a severe error but will not affect the execution of SourcePoint scripts.
<code>error</code>	The error message will be displayed as a normal error and will cause the current script control block to be stopped.
<code>fatal</code>	The error message will be displayed and cause SourcePoint to exit.

Note: The error numbers displayed in the error message are in decimal. However, the error-number base depends on the current number base setting of SourcePoint. To ensure the `error` function can find the correct error number, you need to make sure that the value of `error-number` corresponds to the correct error number in decimal (see example below).

Note: This function is provided for ITP script compatibility. Currently, the only severities supported are `nodisplay` and `error`. Currently, the only error number supported is `0n325` (syntax error).

Example

Command input:

```
error(0n325, nodisplay)
if (_PCI) { ich_inc="blank.itp" }
error(0n325, error)
```


idcode

Display the boundary scan idcode for a device.

Syntax

```
[variable = ][[px]] idcode [(device – number)]
```

Where:

[px]	is a viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
device-number	is the zero-based position of the device in the scan chain.
variable	is an ord4 debug variable to receive the answer.

Discussion

Use the idcode command to display the boundary scan idcode for a device. By default, this command displays the viewpoint processor's idcode. If the device number is specified, the idcode of that particular device is displayed. If both viewpoint override and device number are entered, the viewpoint override is ignored. If an invalid device number or processor override is entered, an error message is displayed.

Example 1

Display current viewpoint processor idcode.

Command input:

```
idcode
```

Result:

```
082E1013H
```

Example 2

Display a particular device's idcode. In the following example, the idcode of device 1 in the boundary scan chain is displayed.

Command input:

```
idcode(1)
```

Result:

084C5013H

Example 3

In the following example, the idcode of the viewpoint processor is assigned to a debug variable.

Command input:

```
define ord4 vpID = idcode  
vpID
```

Result:

082E1013H

Eval Command

Execution of the **eval** command evaluates an address or expression and displays the results.

Syntax

```
eval expr [eval-type]
```

Where:

expr	is the expression to be evaluated
eval-type	physical evaluates the expression as a physical address
	linear evaluates the expression as a linear address
	virtual evaluates the expression as a virtual address

Discussion

Use the **eval** command to calculate and display the result of an expression. The physical, linear, and virtual options enable you to evaluate an expression as an address of the corresponding type using the address translation rules currently in effect for the target system.

Execution Point Command

The **execution point** (\$) control variable displays or changes the current execution point (CS:EIP).

Syntax

\$ [{addr}]

Where:

addr The address of the next instruction to be executed. The \$ control variable is a shorthand way of referring to the LDT:CS:EIP.

Discussion

When an address including an LDT-selector, a segment-selector, and an offset is assigned to the **execution point** control variable, the LDT:CS:EIP register is set to that address. If an offset only is assigned to the control variable, the EIP is changed and the LDT and CS remain the same.

Caution: When you change the execution address with the **execution point** control variable, your disruption of the normal program flow can invalidate the run-time stack.

Examples

1. To display the current execution point:

Command input:

\$

Result:

0008:0030D611 //Result may vary

2. To change the current execution point to the main procedure in the p_main module:

Command input:

\$=:p_main.main

3. To hand-patch a test case at the address 0x0008:000f:000005ff:

Command input:

\$=0x0008:000f:000005ff

asm\$="mov ax,word ptr [0]","mov ebx,eax"

Exit Command

The exit command is used to exit SourcePoint.

Syntax

exit

Discussion

Use the **exit** command to close all open files, terminate the debug session, and return control to Microsoft® Windows®.

Exp Command

The **exp** command returns the exponential function of an expression; that is, the number e raised to the expr power, where e is the base of the natural logarithms. The **exp** function returns an invalid value when the correct return value would overflow.

Syntax

```
[ name = ] exp (expr)
```

Where:

name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

(expr) Specifies a number or an expression of type real8. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Fc Command

The **fc** command compares two text files.

Syntax

FC ("file1", "file2", start column, end column)

Where:

file1, file2	Names of files to be compared.
start column	First column of each line to compare, start column = 0 equals the first column of the line.
end column	The last column of the line to compare.

Note: "Start column" and "end column" are optional input parameters. "Unspecified" defaults to the first and last columns of the line.

Discussion

The **fc** command is used to compare two text files and returns "true" if the files are identical within and including the start and end columns of every line. If the files mismatch, the mismatched line of each file is displayed with an underscore character indicating the first column of the mismatch. If a relative name is specified, then the path specified with the **cwd** command is prepended to the file name. the **fc** command can be used in conjunction with the **list/nolist** command for automatic testing.

Example

Command input:

```
cwd c:\test
list memimage.log
byte DS:0 len 100
nolist
list mem.log
byte DS:0 len 100
no list
if (fc("mem.log", "memimage.log")) printf ("test passed\n")
```

Result:

```
test passed
```

Fclose Command

The **fclose** command closes a file.

Syntax

`fclose (file handle)`

Where:

file handle a file handle returned from a previous `fopen` command

Discussion

The **fclose** command closes a file previously opened by an **fopen** command.

If an **fopen** command is executed within a procedure, then the file handle returned is valid only within that procedure. Files opened outside of a procedure have global scope and may be accessed anywhere. Any files left open when SourcePoint terminates are automatically closed.

Example

Command input:

```
define ord4 file1
file1= fopen ( "test.dat", "w")
fputs ( "this is a test", file1)
fclose (file1)
define nstring buf
file1 = fopen ( "test.dat", "r")
fgets ( buf, file1)
```

Result:

"this is a test"

Command input:

```
fclose (file1)
```


Feof Command

The **feof** command is used to test for end of file (EOF).

Syntax

```
int4 feof ( file handle )
```

Where:

file handle returned from a previous **fopen** command

Discussion

The **feof** command is used to test for the end of file condition. If a file input function has attempted to read past the end of a file, calling the **feof** function will return a value of 00000010H; otherwise, a null value will be returned.

Example

This example illustrates how to read a binary file and write into target memory at address 0.

Command input:

```
define ord4 file1
define ord4 nItemsRead
define ord4 Buf[1000]
define ptr pMem = 0

file1 = fopen ( "test.dat", "r")
while (feof(file1) == 0)
{
  nItemsRead = fread ( Buf, file1)
  ord4 pMem length nItemsRead = Buf
}
fclose (file1)
```

Fgetc Command

The **fgetc** command reads a character from a file.

Syntax

int4 fgetc (file handle)

Where:

file handle file handle returned from a previous **fopen** command

Discussion

The **fgetc** command reads a character from a file previously opened by an **fopen** command. A -1 is returned upon reading end of file.

Example

Command input:

```
define ord4 file1
file1=fopen ( "test.dat","w")
fputc ( 'A',file1)
fclose (file1)
file1=fopen ("test.dat","r")
fgetc (file1)
```

Result:

```
00000041H
```

Fgets Command

The **fgets** command reads a string from a file. Multiple **fgets** commands will get consecutive, new line-delimited strings.

Syntax

nstring fgets (string, file handle)

Where:

string an nstring variable to receive the string read
file a file handle returned from a previous **fopen** command
handle

Discussion

The **fgets** command reads a string from a file previously opened by an **fopen** command. The string is stored in the first argument specified and is also the return value of the function. Strings longer than 1024 characters are truncated, and **fgets** will return an empty string on end of file. The **feof** command should be used to detect end of file.

Example

Command input:

```
define ord4 file1
file1= fopen ( "test.dat", "w")
fputs ( "this is a test", file1)
fclose (file1)
define nstring buf
file1 = fopen ( "test.dat", "r")
fgets ( buf, file1)
```

Result:

```
"this is a test"
```

first_jtag_device

Return the device ID of the first JTAG device.

Syntax

[variable =] first_jtag_device

Where:

variable specifies a debug variable to receive the answer

Discussion

first_jtag_device returns the device ID of the first JTAG device. It can be used with num_jtag_devices and/or last_jtag_device to iterate over JTAG devices.

Example

To create a custom-format devicelist command:

Command input:

```
define proc devlist()
{
  define ord4 o4DevID
  if (num_jtag_devices > 0)
  {
    for (o4DevID=first_jtag_device; o4DevID <= last_jtag_device; o4DevID++)
    {
      printf("%4x %8s (%-11s) port %d, scanchain %d, idcode %x\n",
        devicelist[ o4DevID ].did,
        devicelist[ o4DevID ].alias,
        devicelist[ o4DevID ].devicetype,
        devicelist[ o4DevID ].debugport,
        devicelist[ o4DevID ].scanchain,
        devicelist[ o4DevID ].idcode)
    }
  }
}
```

Flags Command

The **flags** command displays or changes the lower 16 bits of the EFLAGS register.

Syntax

flags [= expr]

Where:

expr specifies a number or an expression to be assigned to the FLAG register

Discussion

Use the **flags** command to display and modify all the flag values or enter the individual reserved word for each flag bit to display the value of that bit. Entering flags without options displays the current value (see the table below).

FLAGS Register Bit Pattern Table			
Bit	Reserved Word	Description	Data Type
15	--	reserved for Intel	--
14	NT	nested task	bit1
13	IOPL	I/O privilege level	bit2
12	IOPL		
11	OF	overflow flag	bit1
10	DF	direction flag	bit1
9	INF	interrupt enable flag	bit1
8	TF	trap enable flag	bit1
7	SF	sign flag	bit1
6	ZF	zero flag	bit1
5	--	reserved for Intel	--
4	AF	auxiliary flag	bit1
3	--	reserved for Intel	--
2	PF	parity flag	bit1
1	--	reserved for Intel	--
0	CF	carry flag	bit1

Examples

1. The following example shows how to display the current value of the EFLAGS register and then modify the register contents by "OR-ing" the current value with 1.

Command input:

flags

Result:

0002H

Command input:

flags = flags | 1
flags

Result:

0003H

2. The following example shows how to display and modify the trap flag (tf).

Command input:

tf

Result:

FALSE

Command input:

tf = true ; tf

Result:

TRUE

flist

Log command line input and responses to a file.

Syntax

```
flist ([filename [,append]])
```

Where:

filename is a string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks.

append is a boolean indicating whether new log data should be appended to or overwrite an existing file. The default is to overwrite.

Discussion

The **flist** function opens a log file. The function performs an action similar to the **list** or **log** commands, except that an nstring variable may be used to specify a file name. The **nolist** command turns logging off. Executing **flist** without specifying a filename displays the currently open log file.

Example 1

To display the current log file:

Command input:

```
flist()
```

Result:

```
"c:\log.txt"
```

Example 2

To open a log file and overwrite an existing file:

Command input:

```
flist ("c:\temp\log.txt")
```

Example 3

To open a log file and append to an existing file:

Command input:

```
flist (c:\temp\log.txt", true)
```


Flush Command

The **flush** command invalidates the L1 and L2 caches.

Syntax

Flush [nowriteback]

[[px]] flush

Where:

nowriteback	nowriteback clears writeback, which is the default condition for the flush command
[px]	Viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain, where x is the number (0, 1, 2, or 3).

Discussion

Use the **flush** command to invalidate the L1 and L2 caches. In a multiprocessor system, only the caches for the current viewpoint are invalidated. The **flush** command can only be used when the target is stopped.

The invd and wbinvd instructions are equivalent to **flush** nowriteback and **flush**, respectively.

Examples

Command inputs:

```
flush           // same as wbinvd instruction
flush nowriteback // same as invd instruction
invd           // same as invd instruction
wbinvd         // same as wbinvd instruction
```

Fopen Command

The **fopen** command opens a file for input or output.

Syntax

```
ord4 fopen (filename,type)
```

Where:

filename	A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks
type	"r" Open an existing file for input. "w" Create a new file, or overwrite an existing one for output. "a" Create a new file, or append to an existing one for output. "b" Open a file as a binary file. Used in conjunction with the above types.

Discussion

The **fopen** command opens a file for input or output. It is similar to the "C" language fopen command except that it returns a file handle of type ord4, rather than a file pointer. This file handle is used in subsequent file I/O commands. If a relative path is specified for a filename, then the current working directory (specified with the **cwd** command) is prepended to the path. If the mode includes "b" after the initial letter, as in "rb" or "w+b", a binary file is indicated. There is no limit to the number of files that may be open.

Use **fclose** to close a file. If an **fopen** command is executed within a procedure, then the file handle returned is valid only within that procedure. Open files are closed automatically when the procedure finishes execution. Files opened outside of a procedure have global scope and may be accessed anywhere. Any files left open when SourcePoint terminates are automatically closed.

Note: If you add "b" to the file type when opening a file, **fgetc** and **fputc** do not perform the translation.

Examples

1. This example illustrates how to create a new file or overwrite an existing one for output.

Command input:

```
define ord4 file1
file1= fopen("test.dat", "w")
fputs("this is a test", file1)
fclose(file1)
define nstring buf
file1 = fopen("test.dat", "r")
fgets(buf, file1)
```

Result:

```
"this is a test:"
```

2. This example illustrates how to write 512 bytes of memory at location 0 to a binary file.

Command input:

```
define ord4 i
define ord4 file1
define ord1 Buf[0x200]
define ord4 MEM_BUFFER=0

file1 = fopen ( "test.dat", "wb")
for (i=0 ; i < 0x200; i++)
Buf[i] = ord1 (MEM_BUFFER + i) ;

fwrite (Buf, file1)
fclose (file1)
```

For Command

This control construct groups and executes commands in a loop.

Syntax

```
for (command1 ; bool-cond ; command2) {commands}
```

Where:

command1	Usually an assignment statement, or a function call, but can be any valid emulator command. If you omit the command1 option, the semicolon (;) must remain as a place holder.
;	The semicolon separates statements and acts as a placeholder. Use two semicolons even if there are not three statements (e.g., for (; i<25 ; i=i+1)).
bool-cond	Specifies the test condition. The bool-cond option must evaluate to true (non-zero) or false (zero). If you omit the bool-cond option, the test condition defaults to true, and the semicolon (;) must remain as a placeholder.
command2	Usually a re-assignment, an increment, or a function call, but can be any emulator command. If you omit the command2 option, the semicolon (;) must remain as a placeholder.
{commands}	One or more emulator commands that are executed when the test condition bool-cond is true. Braces ({ }) indicate the start and end of multiple commands controlled by the for construct. At least one command is required. However, you can enter an empty command, indicated by a semicolon (;).

Discussion

Use the **for** control construct to create iteration constructs that cause the specified commands to be executed one or more times. The iteration continues as long as bool-cond evaluates to non-zero (true) in the following order: command1 is executed once; then if bool-cond evaluates to true, {commands} is executed. After that, command2 is executed and bool-cond is evaluated. This process is repeated as long as bool-cond evaluates to true.

You cannot use the **include** command within a **for** control construct.

Examples

1. The following example shows how to use a **for** control construct. This **for** construct sets an index to zero, sets the test condition to the index being less than 5, and, finally, causes the index to be incremented and displayed. Assume i is defined as an ord1 (byte) data type. The last semicolon is required (it represents an empty command list).

Command input:

```
base = 10t
for (i = 0 ; i < 5t ; i=i+1) ;
i
```

Result:*5T*

2. The following example shows how to increment the value of the BX register using a **for** construct with the AX register as a counter. This **for** construct sets the AX register to zero, sets the test condition to be the value of the AX register being less than or equal to 3, and increments and displays the AX register on each iteration. As long as the AX register is less than 3, the BX register is also incremented by one. The AX register is incremented after the BX register is incremented.

Command input:

```

bx = 10t                               /* Set the value of the BX register. */
for (ax = 0 ; ax <= 3t ; ++ax)
{ bx++ ; printf ("The BX register is: %d \n",bx)
}

```

Result:

```

00000000AH
The BX register is: 11
00000000BH
The BX register is: 12
00000000CH
The BX register is: 13
00000000DH
The BX register is: 14

```

fprintf

Write formatted output to a file.

Syntax

`fprintf (file-handle, format [, expr [, . . .]])`

Where:

`file-handle` is a file handle returned from a previous **fopen** command
`format` is a string constant or nstring variable which determines the format of the display
`expr` is an expression that is evaluated and displayed

Discussion

Use the `fprintf` function to write formatted output to a file. The `fprintf` function is similar to the C-language `fprintf` routine. See [Printf Command](#) for more information.

Example

Command input:

```
define ord4 file1
file1= fopen ( "test.dat", "w")
define nstring myStr = "this is a test"
define ord4 myNum = 1234
define char myChar = 'A'
fprintf (file1, "%s %d %c", myStr, myNum, myChar)
fclose (file1)
```

Fputc Command

The **fputc** command writes a character to a file.

Syntax

`fputc (char, file handle)`

Where:

char character to write

file handle a file handle returned from a previous **fopen** command

Discussion

The **fputc** command writes a character to a file previously opened by an **fopen** command.

Example

Command input:

```
define ord4 file1
file1= fopen ( "test.dat", "w")
fputc ( 'A', file 1)
fclose (file1)
file1 = fopen ( "test.dat", "r")
fgetc ( file1)
```

Result:

65T

Fputs Command

The `fputs` command writes a string to a file.

Syntax

`fputs (string, file handle)`

Where:

`string` a string constant or `nstring` variable to write
`file` a file handle returned from a previous **`fopen`** command
`handle`

Discussion

The **`fputs`** command writes a string to a file previously opened by an **`fopen`** command.

Example

Command input:

```
define ord4 file1
file1= fopen ( "test.dat", "w")
fputs ( "this is a test", file1)
fclose (file1)
define nstring buf
file1 = fopen ( "test.dat", "r")
fgets ( buf, file1)
```

Result:

```
"this is a test"
```


Fread Command

The **fread** command reads binary data from a file into an array.

Syntax

```
ord4 fread ( buffer, file handle )
```

Where:

buffer an array variable to receive the data read
file handle a file handle returned from a previous **fopen** command

Discussion

The **fread** command reads binary data from a file previously opened by an [fopen](#) command. The length of each item of data read is specified by the size of the array. The data are stored in the first argument specified. The returned value is the number of items or data read.

Note: The [feof](#) command should be used to detect end of file.

Example

This example illustrates how to read a binary file and write into target memory at address 0.

Command input:

```
define ord4 file1
define ord4 nItemsRead
define ord4 Buf[1000]
define ptr pMem = 0

file1 = fopen ( "test.dat", "r")
while (feof(file1) == 0)
{
nItemsRead = fread ( Buf, file1)
ord4 pMem length nItemsRead = Buf
}
fclose (file1)
```

fseek

Position at a new location in a file.

Syntax

```
int4 fseek(file_handle, offset, wherefrom)
```

Where:

file_handle	a file handle returned from a previous fopen command
offset	a signed integer specifying a number of bytes
wherefrom	0 = beginning of file, 1 = current location, 2 = end of file

Discussion

The fseek function allows random access within a file. The first argument is a file that is open for input or output. The second argument specifies a position. The third argument is a "seek code," indicating from what point in the file the offset should be measured.

The return value is 0 if successful or nonzero if an error occurs.

Example

This example illustrates how to determine the size of a file.

Command input:

```
define int4 hFile = fopen("test.dat", "r")
fseek(hFile, 0, 2)      // seek to end of file
ftell(hFile)
```

Result:

```
000079A8H           // file size
```

ftell

Returns the current offset within a file.

Syntax

```
int4 fseek(file_handle)
```

Where:

file_handle a file handle returned from a previous [fopen](#) command

Discussion

The ftell function takes a file that is open for input or output and returns the position in the file. The return value is -1 if an error occurs.

Example

This example illustrates how to determine the size of a file.

Command input:

```
define int4 hFile = fopen("test.dat","r")
fseek(hFile,0,2)      // seek to end of file
ftell(hFile)
```

Result:

```
000079A8H           // file size
```

Fwrite Command

The **fwrite** command writes data from an array into a file.

Syntax

ord4 fwrite (buffer, file handle)

Where:

Buffer is an array variable containing the data to write
file is a file handle returned from a previous [fopen](#) command
handle

Discussion

The **fwrite** command writes binary data to a file previously opened by an [fopen](#) command. The data are contained in the first argument specified. The returned value is true if successful.

Example

This example illustrates how to write 512 bytes of memory at location 0 to a binary file.

Command input:

```
define ord4 i
define ord4 file1
define ord1 Buf[0x200]
define ord4 MEM_BUFFER=0

file1 = fopen ( "test.dat", "wb")
for (i=0 ; i < 0x200; i++)
  Buf[i] = ord1 (MEM_BUFFER + i) ;

fwrite (Buf, file1)
fclose (file1)
```

Getc Command

The **getc** command reads a character from the user via the **Command** window.

Syntax

```
char getc ( )
```

Discussion

The **getc** function reads a character from the **Command** window. The character is the return value of the function. The command `getchar` is an alias for the **getc** command.

Example

Command input*:

```
define ord1 ch  
ch = getc()
```

Command input

k // No prompt appears in this space

Command input:

k // Use of particular letter is arbitrary

Result:

6BH 'k'

**Command inputs are cumulative; you must start at the first input to get the result shown.*

getfile

Copy a file from the target to the Sourcepoint host. The file is transferred in binary mode with no ascii translation.

Syntax

```
[result =] getfile(target_path, host_path, permissions)
```

Where:

result	boolean return value is true if the copy succeeded.
target_path	specifies the path on the target host to the file to be copied.
host_path	specifies the path on the host where the file will be created.
permissions	specifies the UNIX file permissions for the host file. This parameter determines the Read/Write/Execute permissions for Owner/Group/Others. It is normally specified as 3 octal digits. For details, refer to the Linux manual page for the chmod command.

Discussion

This command requires Sourcepoint Linux-aware debugging support on the target. Use getfile to retrieve a file from the target and place it on the host filesystem. The Boolean result is true if the file is transferred successfully. In the event of failure, a descriptive error message is displayed in the Command window.

Example 1

Command input:

```
getfile("/tmp/myprog", "C:\\myhome\\myprog", 755o)
```

Example 2

Command input:

```
getfile("/lib/modules/testmod.ko",  
"~/targets/csb337/root_fs/lib/modules/testmod.ko", 644o)
```

Example 3

Command input:

```
getfile("/lib/modules/testmod.ko",  
"U:\\targets\\csb337\\root_fs\\lib\\modules\\testmod.ko", 644o)
```


getNearestProgramSymbol

Returns the nearest program symbol from a given address.

Syntax

```
[ result = ] getNearestProgramSymbol (addr, [proc])
```

Where:

addr is the address to search.
proc is the processor to use (default = current viewpoint).
result is an nstring variable to receive the answer.

Discussion

The `getProgramSymbolAddress` function searches each program loaded in the current context for the specified address. It returns the nearest symbol (either code or data) as a string in “symbol + hex_offset” format.

The optional processor parameter is only meaningful when target memory is configured as not SMP.

An empty string is returned when SP cannot find a symbol.

Example 1

To search the programs loaded on the current viewpoint processor for the symbol nearest to address 0x120:

Command input:

```
getNearestProgramSymbol (0x120)
```

Result:

```
main+0x20
```

Example 2

To search the programs loaded on the current viewpoint processor for the symbol nearest to address 0x100:

Command input:

```
define nstring s  
s = getNearestProgramSymbol(0x100)  
s
```

Result:

```
main
```


GetProgramSymbolAddress Command

The **getProgramSymbolAddress** command returns the address of the symbol referenced by symbol name.

Syntax

```
[ result = ] GetProgramSymbolAddress (symbol_name)
```

Where:

Result	A Pointer variable to which the function return value is assigned.
Symbol	A constant string, nstring, or debug variable specifying the symbol to look up.

Discussion

The **getProgramSymbolAddress** command searches each program loaded in the current context for the specified symbol. If the symbol is found, its address is returned. Otherwise, an error is raised. This function is intended to be used in conjunction with `IsProgramSymbol()` in macro procedure scenarios where program symbols may not be present when the macro is interpreted.

Examples

The following examples demonstrate the **getProgramSymbolAddress** command. Here it is assumed that a program is loaded which contains the data symbol `mydata` at address `C0001000` and a procedure symbol `mycode` at address `C0008000`.

1. First example

Command input:

```
if (IsProgramSymbol("mydata"))
  getProgramSymbolAddress("mydata")
```

Result:

```
0xC0001000
```

2. Second example

Command input:

```
define nstring s = "mydata"
getProgramSymbolAddress(s)
```

Result:

```
0xC0001000
```

3. Third example

Command input:

getProgramSymbolAddress("mycode")

Result:

0xC0008000

4. Fourth example

Command input:

getProgramSymbolAddress("test")

Result:

Error "test" is not a program symbol.

Gets Command

The **gets** command reads a string from the user via the **Command** window.

Syntax

```
nstring gets( string )
```

Where:

string an nstring variable to receive the string read

Discussion

The **gets** function reads a string from the **Command** window. The string is stored in the first argument specified and is also the return value of the function.

go

Start program execution and optionally set a breakpoint.

Syntax

go [forever | tilswb | til vls-addr [length] [type]]

length = {byte | word | dword}

type = {acc | exe | wr | io | rd | smmacc | smmexe | smmwr | smmio}

Where (alphabetically):

acc	specifies that the event to be recognized is a data access (read or write) operation at the specified vls-addr. The default segment of vls-addr is DS.
exe	specifies that the event to be recognized is based on the execution of an instruction at vls-addr. The execute option is the default setting if a type is not specified. The specified vls-addr must identify the first byte of an instruction opcode for it to be recognized. The default segment selector of vls-addr is the current CS.
forever	temporarily disables all breakpoints and begins emulation.
io	specifies that the event to be recognized is an I/O access (read or write) operation at the corresponding port address.
smmacc	specifies that the event to be recognized is a data access (read or write) operation in the SMM address space at the specified vls-addr. The default segment of vls-addr is DS.
smmexe	specifies that the event to be recognized is based on the execution of an instruction in the SMM address space at vls-addr. The execute option is the default setting if a type is not specified. The specified vls-addr must identify the first byte of an instruction opcode for it to be recognized. The default segment selector of vls-addr is the current CS.
smmio	specifies that the event to be recognized is an I/O access (read or write) operation in the SMM address space at the corresponding port address.
smmwr	specifies that the event to be recognized is a memory write operation in the SMM address space. The default segment selector is the current DS.
til	specifies the following event is to be recognized.
tilswb	specifies that emulation continues until a software break is executed. Other breakpoint types are temporarily disabled.
vls-addr	specifies a virtual, linear, or symbolic address (a physical address cannot be entered). If vls-addr is followed by either a write or an access option, then the default segment selector is the current DS. If vls-addr is followed by either the execute option or nothing, the default segment selector is the current CS.
wr	specifies that the event to be recognized is a memory write operation. The default segment selector is the current DS.
byte, word, dword	Specifies the range of the addresses that will cause a break. The byte option is the default setting if a length is not specified.

Discussion

Use the **go** command to control emulation. The **go** command uses the processor debug registers for recognizing vls-addr events.

Hardware breaks are implemented using the on-chip debug registers of the processor. Emulation stops before the instruction at vls-addr is executed. However, if vls-addr is qualified with a write or an access option, the break occurs immediately after the event that caused the match.

When software breakpoints are set, emulation stops before the instruction is executed.

The emulator uses debug registers 0 through 7 and the "Interrupt 1" facilities of the processor . If the target software uses "Interrupt 1" during emulation, an unexpected break occurs. If the target software modifies the processor debug registers while in emulation, the results may be unpredictable.

When you enter the **go** command without any specifications, any breakpoints specified in the **Breakpoints** window will be in effect.

Halt Command

The **halt** command causes the processor to terminate program execution.

Syntax

halt

Discussion

The **halt** command stops target program execution. It is the same as the **stop** command.

Example

Command input:

go til 1000:1034

halt

Help Command

The **help** command displays the Help index.

Syntax

help [topic]

Discussion:

The **help** command opens the SourcePoint Help index. If a topic is specified, the index is opened at the closest match to that topic. A topic can be a partial name. Topics are not limited to Command Language keywords.

Examples

Command input:

help // open the Help index

Command input:

help dbgbreak // open the Help index with the dbgbreak topic selected

Command input:

help memory window // open the Help window with the memory window topic selected

homepath

Return the full path of the directory containing the current SourcePoint executable file.

Syntax

homepath

Discussion

The homepath control variable contains a string that is the full path to the directory where the SourcePoint executable is installed. The string is terminated with a final slash/backslash path delimiter. This variable can be used to avoid hard-coded file paths by referencing them relative to the SourcePoint directory.

Example

Assume SourcePoint executable exists at C:\Program files\Arium\SourcePoint\sp.exe.

Command input:

```
define nstring mymac = homepath + "mac\big.mac";  
mymac
```

Result:

```
"C:\Program Files\Arium\SourcePoint\mac\big.mac"
```

Hotplug Command

The **hotplug** command is used to connect to a running target without destroying target state.

Syntax

hotplug()

Note: The **hotplug** command is a function (the parentheses are required). There are currently no arguments to this function.

Discussion

This function tells the emulator to begin a “hot-plug” operation (a connection to a running target). When this command is entered, a dialog box appears that prompts the user to connect the emulator to the target.

Example

Command input:

Hotplug()

idcode

Display the boundary scan idcode for a device.

Syntax

[variable =][[px]] idcode [(device – number)]

Where:

[px]	is a viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
device-number	is the zero-based position of the device in the scan chain.
variable	is an ord4 debug variable to receive the answer.

Discussion

Use the idcode command to display the boundary scan idcode for a device. By default, this command displays the viewpoint processor's idcode. If the device number is specified, the idcode of that particular device is displayed. If both viewpoint override and device number are entered, the viewpoint override is ignored. If an invalid device number or processor override is entered, an error message is displayed.

Example 1

Display current viewpoint processor idcode.

Command input:

```
idcode
```

Result:

```
082E1013H
```

Example 2

Display a particular device's idcode. In the following example, the idcode of device 1 in the boundary scan chain is displayed.

Command input:

```
idcode(1)
```

Result:

```
084C5013H
```

Example 3

In the following example, the idcode of the viewpoint processor is assigned to a debug variable.

Command input:

```
define ord4 vpID = idcode  
vpID
```

Result:

```
082E1013H
```

If Command

The **if** control construct groups and conditionally executes emulator commands.

Syntax

```
if (bool-cond) {commands1} [ else {commands2}]
```

Where:

(bool-cond)	Specifies a number or an expression which must evaluate as either true (non-zero) or false (zero). The parentheses are required.
{commands1}	Specifies one or more emulator commands (commands1) that are executed when bool-cond evaluates to true (non-zero). The braces ({}) are required when you enter multiple commands.
else {commands2}	Specifies one or more emulator commands (commands2) that are executed if bool-cond evaluates to false (zero). The braces ({}) are required when you enter multiple commands.

Discussion

Use the **if** control construct to group and conditionally execute commands. The **if** control construct tests the bool-cond condition and, if true (non-zero), executes the commands in the commands1 specification. When you use the **else** option, any commands in the commands2 specification are executed when the specified condition evaluates to false (zero).

You can nest **if** constructs. When nested, the optional **else** clause associates with the closest **if** clause. The **if** control construct resembles the C language **if** control construct.

The **else** option must be on the same command line as the end of the {commands1} block. If desired, you can use the continuation character (\) followed by the Enter key at the end of the last line of the {commands1} block to move the **else** option to the next line.

The **include** command is not executable inside the **if** control construct.

Examples

1. The following example shows how to use the **if** control construct to test a condition. If the test condition ($a > b$) evaluates to true, then z takes the value of a . If the test condition evaluates to false, z takes the value of b . Assume that aa , bb , and zz have been previously defined as int1 values.

Command input:

```
define int1 aa
define int1 bb
define int1 zz
if (aa > bb)
    zz = aa
else
```

```
    zz = bb  
zz
```

Result:
00H

2. The following example shows how to use the **if** control construct with the **else** clause. If the test condition ($r0 > 0$) evaluates to true, then increment the R0 and R1 registers; otherwise, do nothing.

Command input:

```
if (r0 > 0)  
{  
    r0 += 1  
} else  
{  
    r1 += 1  
}
```

3. The following example shows how to use the **if** else control construction without the else clause. If the test condition ($ax > 0$) evaluates to true, then increment the AX and BX registers; otherwise, do nothing.

Command input:

```
if (ax > 0)  
{  
    ax += 1  
    bx += 1  
}
```

include

Execute emulator commands from a text file.

Syntax

include [nolist] filename

Where:

nolist	suppresses the echoing of commands to the Command window. Nolog has the same effect as nolist .
filename	is a string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks.

Discussion

Use the **include** command to cause the emulator input to be taken from the named text file. For example, use the include command to do the following:

- Load debug procedures (procs), such as pre-defined sets of tests
- Create debug variables and execute commands
- Create literally (alias) definitions

The output of the **include** command is the same as if the commands had been directly entered in the **Command** window. With the **nolist** option, command echoing is suppressed, but the responses are still displayed. Error messages are displayed if errors occur while processing a command in the file. If the error is severe, inclusion of the file and any nested include files is terminated.

Press Ctrl-Break to abort execution of an **include** file.

Note: If an **include** command appears on a line with multiple commands, it must be the last command on the line. If an **include** command appears within a block (**for**, **if**, etc.) or **proc**, it must be the last command in the block.

Note: Include commands can be nested (a file that is being included can include another file). Nested include files that contain relative paths will be relative to the directory of the parent include file. Otherwise, include files will be relating to the current working directory. (See [Cwd Command](#))

Note: The emulator displays a syntax error when the **include** command processes an undefined debug variable. Define all debug variables before referencing.

Example1

To include a file without echoing commands to the screen:

Command input:

```
include nolist myfile.mac
```

Example 2

To include a file and echo the commands to the **Command** window:

Command input:

```
include myfile.mac  
redefine byte i  
redefine ord2 j  
redefine int2 k  
redefine ord4 r_factor
```


IsDebugSymbol Command

The **IsDebugSymbol** command determines if a string is a debug variable.

Syntax

[result =] isdebugsymbol (symbol)

Where:

result	A boolean variable to which the function return value is assigned. It is TRUE if the symbol exists, or FALSE if it doesn't exist.
symbol	A constant string or nstring specifying the debug variable to look up.

Discussion

The **IsDebugSymbol** command checks the name of each debug variable to see if it matches the specified string. If a match is found, it stops searching and returns TRUE. It returns FALSE if no match is found.

Examples

1. **Command input:**
define ord4 x = 5
isdebugsymbol("x")

Result:
TRUE

2. **Command input:**
define nstring s = "x"
isdebugsymbol(s)

Result:
TRUE

3. **Command input:**
isdebugsymbol("foo")

Result:
FALSE

isem64t

Display whether the specified processor supports Extended Memory 64 Technology.

Syntax

```
[variable=] [[px]]isem64t
```

Where:

[px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.

variable is a debug variable to hold the return value.

Discussion

The isem64t control variable displays whether the specified processor supports Extended Memory 64 Technology (now more commonly known as Intel 64).

Example

Command input:

```
printf("The processor %s Intel 64\n", isem64t ? "supports" : "does not support")
```

Result:

```
The processor supports Intel 64
```

IsProgramSymbol Command

The **IsProgramSymbol** command determines if a string is a symbol within a currently loaded program.

Syntax

[result =] isprogramsymbol (symbol)

Where:

result	A boolean variable to which the function return value is assigned. It is TRUE if the symbol exists, or FALSE if it doesn't exist.
symbol	A constant string or nstring specifying the symbol to look up.

Discussion

The **IsProgramSymbol** command looks in each currently loaded program until it finds the specified symbol. When the first occurrence is found, it stops searching and returns TRUE. It returns FALSE if no instance of that symbol is found within any currently loaded program.

Examples

The following examples demonstrate the isprogramsymbol command. Here it is assumed that a program is loaded which contains the symbols foo and fun.

1. **Command input:**
isprogramsymbol("foo")

Result:
TRUE

2. **Command input:**
define nstring s = "fun"
isprogramsymbol(s)

Result:
TRUE

3. **Command input:**
isprogramsymbol("test")

Result:
FALSE

isrunning

Display whether the specified processor is running.

Syntax

```
[variable=] [[px]] isrunning
```

Where:

[px] is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.

variable is a debug variable to hold the return value.

Discussion

Use the isrunning control variable to determine if a specific target processor is running. This returns false for threads(or processors) that are either halted or disabled. Entering the command at the command line or in an expression returns 0 (false for halted or disabled) or 1 (true for running).

Example 1

To display the state of the viewpoint processor:

Command input:

```
isrunning
```

Result:

```
false
```

Command input:

```
go  
isrunning
```

Result:

```
true
```

Example 2

To save current viewpoint processor state in a user defined variable:

Command input:

```
define ord1 _isrunning  
_isrunning = isrunning
```

Example 3

To use isrunning in an expression:

Command input:

```
go  
printf("processor is %s\n", isrunning ? "running" : "stopped")
```

Result:

```
processor is running
```

issmm

Display whether the specified processor is in system management mode.

Syntax

```
[result=] [[px]] issmm
```

Where:

[px]	is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
result	is a debug variable to hold the return value.

Discussion

Use the issmm control variable to determine if a specific target processor is in system management mode. Entering the command at the command line or in an expression returns 0 for normal mode or 1 for smm.

Example 1

To display the state of the viewpoint processor:

Command input:

```
issmm
```

Result:

```
FALSE
```

Example 2

To display the state of processor P3:

Command input:

```
[P3]issmm
```

Result:

```
TRUE
```

Example 3

To use issmm in an expression:

Command input:

```
printf("processor is %s\n", issmm ? "in smm" : "not in smm")
```

Result:

```
processor is in smm
```

itpcompatible

Enable Intel ITP compatibility.

Syntax

itpcompatible = {true | false}

Discussion

The itpcompatible control variable changes SourcePoint behavior to be more compatible with the Intel ITP command language. When enabled command abbreviations, and alternate register names are not allowed. This helps prevent keyword conflicts when running ITP macro files. The default is false.

Example 1

When ITP compatibility is disabled the keyword length may be abbreviated to “len”.

Command input:

```
itpcompatible = false  
ord4 0 len 10
```

Result:

```
00000000 E59FF018 E59FF018 E59FF018 E59FF018  
00000010 E59FF018 E1A00000 E59FF018 E59FF018  
00000020 FFF01D88 FFF03444 FFF01DBC FFF034E0  
00000030 FFF03520 00000000 FFF0389C FFF02032
```

Example 2

When ITP compatibility is enabled the keyword length may no longer be abbreviated.

Command input:

```
itpcompatible = true  
ord4 0 len 10
```

Result:

```
Syntax error: ord4 0 _len 10
```


JtagChain Command

The **JtagChain** command is used to both display and define the target JTAG configuration.

Syntax

Jtag[Chain] ([jtag_id][,jtag_id]+)

Where:

Jtag_id an expression resolving to a 32-bit JTAG ID

Discussion

If no arguments are specified, then the current JTAG configuration is displayed. There is one line of display per JTAG device. If SourcePoint is not connected to a target, then an error message is displayed.

If JTAG ID values are specified, then this command defines the target JTAG configuration. The order of IDs listed indicates the order of devices on the JTAG chain. This configuration is downloaded to the emulator and saved in flash memory.

The emulator uses this JTAG configuration only when it is not auto-detecting the JTAG chain. This is controlled by the **Auto-detect JTAG device** option in the **Emulator Configuration** dialog box. If a JTAG configuration is specified, and the emulator is currently configured to auto-detect the JTAG chain, then a warning is issued. All JTAG IDs specified must already be defined in either the internal tables of SourcePoint and/or the emulator, or via the [JtagDeviceAdd](#) command. An error message is displayed if this is not the case.

Examples

1. The first example shows how to find JTAG chain data automatically.

Command input:

```
Jtagchain()        //display current jtag chain data
```

Result:

```
JTAG Chain
ID = 0x0005C013, IR length = 7, Max Jtag Rate = 16 MHz, Processor = 0x020D - x86
Family 6 (Merom)
```

2. This example shows how to display JTAG chain information when you know the JTAG chain ID.

Command input:

```
Jtagchain(0x0005C013) //display jtag chain data at JTAG chain ID 0x0005C013
```

Result:

```
JTAG Chain
ID = 0x0005C013, IR length = 7, Max Jtag Rate = 16 MHz, Processor = 0x020D - x86
Family 6 (Merom)
```

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

JtagDeviceAdd Command

The **JtagDeviceAdd** command is used to add a JTAG ID to both SourcePoint and the emulator.

Syntax

```
jtagdeviceadd ( jtag_id , ir_length , processor_id [, max_rate] )
```

Where:

jtag_id	an expression resolving to a 32 bit JTAG ID
ir_length	an expression resolving to an integer between 1 and 128
processor_id	an expression resolving to a processor id value
max_rate	an expression resolving to an integer between 0-40 (MHz)

Discussion

The **Jtagdeviceadd** command is used to add a JTAG device definition to SourcePoint. This action is persistent. Cycling power on the emulator or restarting SourcePoint does not remove the new ID.

Processor ID is a hex value that indicates to SourcePoint the processor type of the new device. Legal ID values can be obtained from the [JtagDevices](#) command. A value of 0 (zero) indicates a non-processor device.

The max_rate argument specifies the maximum JTAG rate that may be specified (via the **Emulator Configuration** dialog box). This argument is optional. If a value is not specified, then 16 MHz is assumed.

New IDs are saved in the SourcePoint INI file. If an ID already exists, then the new ID settings take precedence.

Example

Command input:

```
Jtag device add (0X09271013,7,0x606,12)
```

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

JtagDeviceClear Command

The **JtagDeviceClear** command is used to remove a JTAG ID from both SourcePoint and the emulator.

Syntax

`JtagDeviceClear (jtag_id)`

Where:

`jtag_id` an expression resolving to a 32 bit JTAG ID

Discussion

Th **JtagDeviceClear** command is useful for dealing with ARM processors that have either duplicate or un-initialized JTAG IDs. This action is persistent. Cycling power on the emulator or restarting SourcePoint does not restore the deleted ID.

SourcePoint maintains two lists of JTAG IDs: a permanent list in the code, and a separate list in the INI file. Entries in the INI file are generated by the user, either by manually editing the INI file or via the [JtagDeviceAdd](#) command. When SourcePoint processes the **JtagDeviceClear** command, it will look for the ID in the INI file. If it is found, then it is removed. If the ID is found in the permanent list, however, then a special deleted entry is saved in the INI file. This entry indicates to SourcePoint that the ID in the permanent list should be ignored.

Example

Command input:

JtagDeviceClear (0)

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

JtagDevices Command

The **JtagDevices** command displays SourcePoint's internal table of JTAG devices.

Syntax

JtagDevices ()

Discussion

The **JtagDevices** command displays SourcePoint's internal table of JTAG devices. There is one line of display per device. For each device, the JTAG ID, in length, max JTAG rate, and processor ID and type are shown.

Example

Command input:

Jtag devices ()

Result:

[multiple lines of code]

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

JtagScan Command

The **JtagScan** command forces the emulator to re-scan the target JTAG chain.

Syntax

```
jtagscan()
```

Note: The **JtagScan** command is a function (the parentheses are required). There are currently no arguments to this function.

Discussion

The **JtagScan** command tells the emulator to re-scan the target JTAG chain.

Example

Command input:

```
JtagScan()
```

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

Keys Command

The **keys** command simulates keyboard input from within a command file.

Syntax

`Keys("keyststring" [, "keyststring"]+)`

Where:

keyststring A key name:

F1-F12
 control (ctrl), alt (menu), shift
 up, down, left, right
 insert, delete
 home, end
 pgup (next), pgdn (prior)
 bs, tab, enter (return), esc, pause
 apps (displays context menu)
 One or more of the following characters:
 a-z
 A-Z
 0-9
 ` ~ ! @ # \$ % ^ & * ()
 _ = + [] { } \ | ; : ' "
 , . < > / ? (space)

Discussion

The **keys** command is used to simulate keyboard input from within a command file. The three mode keys (Control, Alt, and Shift) apply to all the rest of the keys in the command, e.g., `keys("ctrl", "f", "g")` simulate pressing the keys ctrl-f followed by ctrl-g, not ctrl-f followed by a "g". Simple, single character keys can be combined within a single keyststring, e.g., `keys("123")` is the same as `keys("1", "2", "3")`.

Examples

1. **Command input:**
`keys("alt", "v", "c")` // opens a **Code** window
2. **Command input:**
`keys("alt", "v", "c")` // opens a **Code** window
`keys("ctrl", "f")` // opens the **Find** dialog box

```
keys("123", "enter")           // searches for the string 123
```

3. **Command input:**

```
keys("alt", "v", "c")          // opens a Code window  
keys("alt", "f", "a")          // opens File|Save As dialog box  
keys("abcdef.txt")             // names the output file "abcdef.txt"  
keys("enter")                  // saves the file
```


Last Command

This function is used to return that last address of a symbol.

Syntax

```
last(symbol)
last(:module.procedure)
```

Where:

<i>symbol</i>	Symbolic reference to a program item (label, variable, array, structure, constant, procedure, module, or program)
<i>module</i>	Symbolic reference to a module
<i>procedure</i>	Symbolic reference to a procedure

Discussion

The **last** command returns the last address occupied by a program item whose symbol is supplied as the argument to last. The address is byte aligned. The **last** function is particularly useful in setting ETM range breaks. This function may be used in SourcePoint wherever an address is used.

There are some caveats to using the **last** function:

- The return value of **last** when the argument is a label is the same address you get when you just type the label.
- Local variables are stack variables and do not have an address that can be determined beforehand, so last function will not work unless those variables are in scope. The problem is even in the same procedure such a variable may go out of scope or the stack frame may change.
- Register variables do not have addresses so the last function won't work with them.
- The return value of **last** when the argument is a non-external procedure in a module that hasn't been analyzed will be the same address that is returned when just then procedure name is typed, which is incorrect. Because the module has not been analyzed, the symbol for the procedure is just a label and will not return an address that is the last address of the procedure (see Bullet 1). To insure that the module is analyzed, use the second syntax shown above.

Examples

Note: The symbols shown in the examples below were derived using the Arium sample file ARMFLAT_nosemi.axf.

1. This example finds the first and last address of the global structure *fooStruct*. Note that an '&' must be prepended to the symbol *fooStruct*; otherwise the command language evaluates *fooStruct* and return its contents.

Command input:

```
&fooStruct
```

Result:

000080C8

Command input:

last(fooStruct)

Result:

000080D3

2. The example illustrates the finding of the first and last address of the procedure *fooFunk*.

To get the first address of a procedure, type its name at the command line:

Command input:

fooFunk

Result:

00000240

To get the last address of the procedure *fooFunk*, first try *last* with just the procedure name as the argument. If the module containing the procedure *fooFunk* has not been analyzed, the result is the same as in the step above since *fooFunk* is now just a label and the last address of a label is the same as the first.

Command input:

last(fooFunk)

Result:

00000240

Now try the *last* command using the syntax that also specifies the module (*CSAMPLE2*) containing the procedure *fooFunk*. This causes the module to be analyzed and so you get the correct last address of *fooFunk*.

Command input:

last(:CSAMPLE2.fooFunk)

Result:

00000273

last_jtag_device

Return the device ID of the last JTAG device.

Syntax

[variable =] last_jtag_device

Where:

variable specifies a debug variable to receive the answer

Discussion

last_jtag_device returns the device ID of the last JTAG device. It can be used with num_jtag_devices and/or first_jtag_device to iterate over JTAG devices.

Example 1

To create a custom-format devicelist command:

Command input:

```

define proc devlist()
{
  define ord4 o4DevID
  if (num_jtag_devices > 0)
  {
    for (o4DevID=first_jtag_device; o4DevID <= last_jtag_device; o4DevID++)
    {
      printf("%4x %8s (%-11s) port %d, scanchain %d, idcode %x\n",
        devicelist[ o4DevID ].did,
        devicelist[ o4DevID ].alias,
        devicelist[ o4DevID ].devicetype,
        devicelist[ o4DevID ].debugport,
        devicelist[ o4DevID ].scanchain,
        devicelist[ o4DevID ].idcode)
    }
  }
}

```

Left Command

The **left** command extracts a number of characters from the beginning of a string.

Syntax

`left (string-expr, n)`

Where:

string-expr	specifies an nstring variable or string constant
n	specifies the number of characters to extract

Discussion

The **left** command returns a substring from the beginning of a string. If the number of characters to extract is greater than the length of the string, then the entire string is returned.

Example

Command input:

```
base = 10t           // Set number base to decimal
define nstring month = "January"
define nstring temp = left(month, 3)
temp
```

Result:

```
"Jan"
```

libcall

Define a function interface to a DLL library.

Syntax

```
libcall (strLibrary, strInterface, [strName], ret-type [{byref} arg-type][, . . . ] [ , . . . ])
```

Where:

strLibrary	is a string expression indicating the library containing the function to import into SourcePoint.
strInterface	is a string expression specifying the function within the library.
strName	is a string expression defining the keyword to be used by SourcePoint when accessing the function. This name must not be the same as SourcePoint keywords or previously defined user functions. If not specified the keyword will be the same as strInterface.
ret-type	is the return type of function: {void string data-type}
arg-type	is the type of an argument passed to the function {string data-type data-type[]}
data-type	is a parameter type specification.
void	specifies there is no return type.
byref	indicates that the following parameter is to be passed in as a reference.
string	is a string parameter.
data-type[]	is an array parameter.
...	(ellipsis) indicates a variable type/length parameter list. If specified, it must be last.

Discussion

Use the libcall command to create an interface to an exported function in a DLL that can then be called from the SourcePoint command line as a user defined function.

The first parameter specified is the library in which the exported function resides.

The second parameter specified is the case-sensitive interface name of the exported function within the external library.

The third parameter is the case-sensitive alias within SourcePoint with which the exported function may be accessed. This alias may be omitted (though the trailing comma still must be present) in which case the same name as the exported function is used.

NOTE: If the function defined already exists in SourcePoint (e.g., printf), then the internal function in SourcePoint will be used.

After specifying the user-defined function's name and where to access it in a DLL, the return type and parameter list must be specified in a manner which will match the exported function within the DLL.

The return type of a user-defined function may either be a ord, int, real, void, or a string. Return types cannot be specified as references, arrays, structures, etc.

A parameter defined with the `byref` keyword is an out or in/out parameter. Without the `byref` keyword, the parameter is an in parameter.

Parameter types of `byref`, `string`, and `array` are passed as pointers to the external library call.

An ellipsis as a parameter list or at the end of a parameter list indicates the user-defined function has a variable number and type of parameters.

`Libcall` determines whether a function is declared in a dll as either `_cdecl` or `_stdcall`.

Arrays are not currently supported.

Example 1

To call some functions in the Microsoft run-time library `MSVCRTD.DLL`:

Command input:

```
libcall ("msvcrt.dll", "atof",,,double,string)
define double pi
pi=atof ("3.14159")
```

```
libcall ("msvcrt.dll", "rand",,,ord2)
define ord4 result
result=rand()
```

Example 2

To call the `MessageBox()` function in the Microsoft library `USER32.DLL`:

Command input:

```
libcall ("user32.dll", "MessageBoxA",,,ord4 ord4, string, string, ord4)
define ord4 result
result=MessageBoxA(0, "Test Text", "Caption", 3)
```

Example 3

To call the `sscanf()` function in the Microsoft run-time library and rename it to `_sscanf`:

Command input:

```
libcall ("msvcrt.dll", "sscanf",_sscanf,ord4,string,string,...)
define string _str = "1/2/2008"
define ord4 mon, day, yr, result
result = _sscanf(_str, "%d/%d/%d",byref mon,byref day,byref yr)
```

Example 4

To call the `TestCall1()` function in a user-created dll called `testdll.dll` which returns a value in the `byref` parameter:

Command input:

```
libcall ("testdll.dll","TestCall1",,ord4, byref ord4)  
define ord4 outvalue  
TestCall1 (byref outvalue)
```

License Command

The license command provides information on the license available with your SourcePoint software.

Syntax

license()

Note: The information box refers to a "certification file" if you are debugging with a Linux host.

Linear Command

The **linear** command translates an address to a linear address.

Syntax

`linear (addr)`

Where:

`linear` calls the linear address translation function
(`addr`) specifies an address to be translated to a linear address

Discussion

Use the **linear** command to translate the specified `addr` to a linear address using the address translation rules currently in force in the target system (e.g., paging or current processor mode).

- When you enter a linear address, it is returned unchanged.
- When entering a virtual address, it's translated to a linear address.

Example

1. The following example shows how to translate a real mode virtual address.

Command input:
`linear(1234:5678)`

Result:
`000179b8L`

2. The following example shows how to translate a protected mode virtual address.

Command input:
`linear(18h:14h:0)`

Result:
`00C03000L`

List, Nolist Commands

The **list**, **nolist** commands record an emulator session to a file or close the file.

Syntax

list [[append | overwrite] filename]

nolist

Where:

append	Causes information from the current debug session to be added to the end of an existing file.
overwrite	Overwrites filename with the current information.
filename	A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks
nolist	Closes a list file. The nolist command has no effect if a list file is not open.

Discussion

Use the **list** command to make a log of a debugging session. **List** records all emulator command interaction within the **Command Line** window during a debug session to a specified file. You can append data to an existing file, overwrite an existing file, or create a new list file. If you enter the **list** command without options, the current list file is used.

To create a list file, enter the **list** command followed by a filename. This file keeps a record of all commands entered and all data displayed in the **Command** window during the debug session. If a list file is currently open, the current list file is closed and the new list file opened. If the filename already exists, a query to overwrite the file is displayed. If the command **list existing file name** is issued from an include file, the command fails and a warning is displayed.

To discontinue sending data to the list file, enter **nolist**.

Example

This example shows how to open a list file named session.log, enter the current register contents, and then close the file.

Command input:

```
list session.log
eax ; ebx ; ecx ; edx ; esi ; edi ; ebp ; esp
```

Result:

```
00000000H
00000000H
00000000H
00000673H
00000000H
```

00000000H
00000000H

Command input:

ds ; es ; fs ; gs ; ss ; cs

Result:

0000H
0000H
0000H
0000H
0000H
F000H

Command input:

nolist

Load Command

The **load** command loads object-module files into target memory.

Syntax (Executable File)

```
load filename [init] [nocode] [AT address | OFFSET expr]
```

Where:

filename A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks

init Specifies that registers are to be initialized from values in load file.

nocode Specifies that object code is not loaded into memory during a load operation.

address Specifies the load address for a non-relocatable file.

expr Specifies a relocation offset for a relocatable file.

Discussion

Use the load command to read an executable file into target memory and/or to load a file's symbols onto the host for symbolic display.

	elf	aout	bin	exe	hex	omf86	omf386	PE	textsym
load symbols	x	x					x	x	x
load target	x		x	x	x	x	x	x	
relocate address			x					x	
relocate offset	x			x	x				x
initialize	*						x		

*Limited processor initialization

Memory writers are verified depending on the state of the [verify](#) control variable.

LoadProject Command

The **LoadProject** command loads a SourcePoint project file.

Syntax

LoadProject ([string-expr])

Where:

string-expr	an nstring variable or string constant specifying the location of the project file to load
-------------	--

Discussion

The **LoadProject** command loads the specified project file. A project file contains all SourcePoint settings including the position and size of each window.

If a relative path is specified, then the current working directory path is prepended to the path (see the [cwd](#) control variable for more information). If a project file is not specified, then the name of the currently loaded project file is displayed.

Examples

1. This example shows how to load a project file.

Command input:

LoadProject("c:\\test\\test.prj")

2. This example displays the name of the currently loaded project file.

Command input:

LoadProject()

Result:

c:\\test\\test.prj"

LoadTarget Command

The **LoadTarget** command loads a target configuration.

Syntax

LoadTarget (string-expr)

Where:

string-expr	an nstring variable or string constant specifying the target configuration to load
-------------	--

Discussion

The **LoadTarget** command loads the specified target configuration. A target configuration includes memory map settings, safe mode settings, flash programming parameters, emulator configuration parameters, event macro specifications, and **Device** window files to load. Target configurations are provided by Arium. User-defined target configurations can be created via the **Target Configuration** dialog box (under the **Options** menu).

Example

Command input:

LoadTarget("myTargetConfig")

Log, Nolog Commands

The **log**, **nolog** command saves the **Command Line** window I/O to a file (same as **list**, **nolist**).

Syntax

```
log [ [ append | overwrite ] filename ] nolog
```

Where:

append	Causes information from the current debug session to be added to the end of an existing file.
overwrite	Overwrites filename with the current information.
filename	A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks
nolog	Closes a log file. The nolog command has no effect if a log file is not open.

Discussion

The **log** command records a history of all information in the **Command Line** window to a file. All data in the **Command Line** window is recorded, including: prompts, input line echoes, and error messages.

To display the name of the open **log** file, enter "log" (without the parentheses).

The **nolog** command closes the current log file. Opening another log file with the **log** command or exiting the **Command Line** window also closes the current file. Using an existing filename overwrites that file.

Example

Command input:

```
log session.log
eax ; ebx ; ecx ; edx ; esi ; edi ; ebp ; esp
```

Result:

```
00000000H
00000000H
00000000H
00000673H
00000000H
00000000H
00000000H
```

Command input:

```
ds ; es ; fs ; gs ; ss ; cs
```

Result:

```
0000H
0000H
```

SourcePoint 7.7.1

0000H
0000H
0000H
F000H

Command input:

nolog

Log10 Command

The **log10** command returns the base 10 logarithm of an expression.

Syntax

[name =] log10 (expr)

Where:

Name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

(expr) Specifies a number or an expression of type real8. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Loge Command

The **loge** command returns the natural logarithm of an expression.

Syntax

[name =] loge (expr)

Where:

- Name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
- (expr) Specifies a number or an expression of type real8. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

MacroPath Control Variable

The **macroPath** control variable contains the full path to the directory containing the macro currently being executed.

Syntax

MacroPath

Discussion

The **macroPath** control variable is a string that contains the full path to the directory where the currently executing macro is located. The string is terminated with a final slash/backslash path delimiter. If this variable is referenced from a context outside of macro file execution, the result will be an empty string.

Example

Assume the currently executing macro C:\Program Files\Arium\SourcePoint\mac\big.mac.

Command input:

```
define nstring mymac = macropath + "other.mac";  
mymac
```

Result:

```
" C:\Program Files\Arium\SourcePoint\mac\other.mac"
```

messagebox

Display a user-defined message box.

Syntax

```
ord4 message[box] (string-expr [, icon, buttons])
```

Where:

string-expr specifies the text to display; can be an nstring variable or string constant
 icon specifies the icon type to display in the message box
 buttons specifies the button layout of the message box

Discussion

The **messagebox** function displays a user-defined message box with variable text, icons, and button layouts.

The icon argument is optional. If not specified, then MB_ICONEXCLAMATION is assumed. Possible icons include:

MB_ICONINFORMATION	Displays an information icon
MB_ICONEXCLAMATION	Displays an exclamation mark icon
MB_ICONQUESTION	Displays a question mark icon

The button argument is optional. If not specified, then MB_OK is assumed. Possible button layouts include:

MB_OK	Display a single OK button
MB_OKCANCEL	Displays OK and Cancel buttons
MB_YESNO	Displays Yes and No buttons
MB_YESNOCANCEL	Displays Yes, No, and Cancel buttons
MB_RETRYCANCEL	Displays Retry and Cancel buttons
MB_ABORTRETRYIGNORE	Displays Abort, Retry, and Ignore buttons

The messagebox command returns a value corresponding to which button was pressed. Possible return values include:

ID_OK	OK button was pressed
ID_YES	Yes button was pressed
ID_NO	No button was pressed
ID_RETRY	Retry button was pressed
ID_IGNORE	Ignore button was pressed
ID_CANCEL	Cancel button was pressed
ID_ABORT	Abort button was pressed

Example 1

To open a message box with a single OK button:

Command input:

```
messagebox("This is a test")
```

Example 2

To open a multi-line message box:

Command input:

```
messagebox("This is line 1\n\nAnd this is line 2")
```

Example 3

To open a message box with Yes and No buttons and check whether the Yes button was pressed:

Command input:

```
if (message("Yes or No?", MB_ICONQUESTION, MB_YESNO) == ID_YES)  
{  
    // execute some additional code  
}
```

Mid Command

The **mid** command extracts a number of characters from the middle of a string.

Syntax

`mid (string-expr, n, m)`

Where:

string-expr	specifies an nstring variable or string constant
n	specifies the 0-based index of the first character to extract
m	specifies the number of characters to extract

Discussion

The **mid** command returns a substring from the middle of a string. If the number of characters to extract exceeds the number of characters in the string, then the command behaves like the **right** command.

Example

Command input:

```
base = 10t           // Set number base to decimal
define nstring month = "January"
define nstring temp =mid(month, 3, 3)
temp
```

Result:

```
"uar"
```

Msgclose Command

The **msgclose** command completes the construction of the message specified by handle.

Syntax

[result =] msgclose (msg-handle)

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the JTAG message was successfully closed. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG message was created.

Discussion

Use the **msgclose** command after all the scans have been added to the message. No more scans can be added to the JTAG message after msgclose executes. An error is returned if the JTAG message contains no scans.

Example

Command input:

```
// Create a JTAG message
define handle h = 0
msgopen (h)
msgir (h, 7, 2)
msgdr (h, 20, 2)
msgclose (h)
```

Msgdata Command

The **msgdata** command retrieves the return data of all scans in a JTAG message previously scanned to the target device(s).

Syntax

[result=] msgdata (msg-handle, return-array)

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the JTAG message was successfully cleared. FALSE indicates an error occurred, such as the specified JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG command was created.
return-array	The previously defined array of ord1, ord2, or ord4 in which the data returned from the scan of target device(s) is stored..

Discussion

Use the **msgdata** command to retrieve the data that was generated by the [msgscan](#) command. The scan data are associated with the JTAG message specified by handle. The return array used to store the scan data can be of type ord1, ord2 or ord4. An error is returned if the msgscan command has not been run on the JTAG message specified by handle.

If multiple read scans are done and more than one set of scan data is expected, the sets of scan data are packed bit-aligned (not separated by any bits). For example, if two read scans are performed and a 5-bit data set containing 10001 and a 7-bit data set containing 0111110 are expected, then an ord1 return array of size 2 (two bytes) would contain all 12 bits next to each other with the four extra bits set to zero as follows. Note that the actual data is in bold and the filler 0 bits are normal.

MSB 1000101111100000 LSB

data [0]=E0H ". "

data [1]=8BH ". "

Example

Command input:

```
// Read JTAG ID from processor
define handle h = 0
define ord2 device = 0
msgopen (h)
msgir (h, 4, 2)
msgdr (h, 20, 2)
msgclose (h)
msgscan (h, device)
define ord4 count = 0
```



```
msgreturndatasize (h, count, device)  
define ord1 data[count]  
msgdata( h, data)  
data  
msgdelete (h)
```

Msgdelete Command

The **msgdelete** command deletes the message and releases the JTAG message handle.

Syntax

```
[result =] msgdelete (msg-handle)
```

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the JTAG message was successfully deleted. FALSE indicates an error occurred, such as the JTAG message was not found
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG message was created

Discussion

Use the **msgdelete** command to release a JTAG message handle so it can be used again.

Example

Command input:

```
define handle h = 0  
msgopen(h)  
msgdelete(h)
```

Msgdr Command

The **msgdr** command records a DR scan into the existing JTAG message specified by handle.

Syntax

```
[result =] msgdr (msg-handle, dr-length, readwrite[drscan-option])
```

Where drscan-option is one of the following:

```
[drscan-option] = [, write-array, [scan-chain, [0, [stop-state[, 0]]]]]
```

```
[drscan-option] = [, write-value, [scan-chain, [0, [stop-state[, 0]]]]]
```

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the DR scan was successfully added to the JTAG message. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG message was created.
dr-length	An ord4 that contains the number of bits to be scanned to a data register (DR).
readwrite	An ord1 that specifies the type of DR scan. See valid readwrite values below.
write-array	An array of type ord1, ord2, or ord4, of bits to scan to the data register. If no write-array is specified, then zeros are scanned.
write-value	An ord1, ord2, or ord4 value to scan to the data register. If no write-value is specified, then zeros are scanned.
scan-chain	An ord1 that specifies which scan chain to select on the debug port. This may only be either 0 or 1. If scan-chain is not specified, then scan chain 0 is used.
stop-state	An ord1 that specifies the TAP state in which to stop at the end of the scan. If stop-state is not specified, then 0 (RTI) is used.

Discussion

Use the **msgdr** command to add a data register (DR) scan to the open JTAG message. The DR length must be specified. The additional section of parameters, drscan-option, is optional. This command returns an error if the JTAG message has been closed.

The legal readwrite values for DR scans are:

0	write-only
1	readwrite
2	read0 (read by writing 0s to DR)
3	read1 (by writing 1s to DR)

The legal stop-state values are:

0	RTI : default
1	CAPTURE-PAUSE: stop in the DR pause state, no data are shifted
2	PAUSE: stop in the DR pause state
3	CAPTURE-RTI: force through the DR capture state, no data are shifted and stop in RTI
4	RTI-DUAL: go to RTI from the DR PAUSE, clocks both time bases, no data are shifted
5	CAPTURE-PAUSE-DUAL: stop in the DR pause state, clocks both time bases, no data are shifted

Example

Command input:

```
define handle h = 0
msgopen (h)
msgdr (h, 20, 2)
```

Msgir Command

The **msgir** command records an IR scan to the JTAG message.

Syntax

```
[result =] msgir (msg-handle, ir-length, write-array, [irscan-option])
```

```
[result =] msgir (msg-handle, ir-length, write-value, [irscan-option])
```

Where irscan-option is:

```
[irscan-option] = [, readwrite, [scan-chain, [0, [stop-state, [0]]]]]
```

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the IR scan was successfully added to the JTAG message. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG message was created.
ir-length	An ord4 that contains the number of bits to be scanned to an instruction register (IR).
write-array	An array of type ord1, ord2, or ord4, of bits to scan to the instruction register. If no write-array is specified, then zeros are scanned.
write-value	An ord1, ord2, or ord4 value to scan to the instruction register. If no write-value is specified, then zeros are scanned.
readwrite	An ord1 that specifies the type of IR scan. See valid readwrite values below.
scan-chain	An ord1 that specifies which scan chain to select on the debug port. This may only be either 0 or 1. If scan-chain is not specified, then scan chain 0 is used.
stop-state	An ord1 that specifies the TAP state in which to stop at the end of the scan. If stop-state is not specified, then 0 (RTI) is used.

Discussion

Use the **msgir** command to add an instruction register (IR) scan to the open JTAG message. The IR length must be specified. The additional section of parameters, irscan-option, is optional. This command returns an error if the JTAG message has been closed.

The legal readwrite values for IR scans are:

0	write-only
1	readwrite

The legal stop-state values are:

0	RTI: default
---	--------------

1	CAPTURE-PAUSE: stop in the IR pause state, no data are shifted
2	PAUSE: stop in the IR pause state
3	CAPTURE-RTI: force through the IR capture state, no data are shifted and stop in RTI
4	RTI-DUAL: go to RTI from the IR PAUSE, clocks both time bases, no data are shifted
5	CAPTURE-PAUSE-DUAL: stop in the IR pause state, clocks both time bases, no data are shifted

Example

Command input:

```
define handle h = 0
msgopen (h)
msgdr (h, 20, 2)
```

Msgopen Command

The **msgopen** command creates a new JTAG message.

Syntax

[result =] msgopen (msg-handle)

Where:

variable	A boolean variable that contains the return value of this command. TRUE indicates the IR scan was successfully added to the JTAG message. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is a reference parameter that is modified by msgopen. After msgopen completes, msg-handle contains an unique value that identifies this JTAG message.

Discussion

Use the **msgopen** command to create an empty JTAG message and assign a unique identifier to msg-handle. The specified msg-handle must exist or an error is reported. If the msg-handle points to a JTAG message that is already open (even if it has been closed), an error is reported.

Example

Command input:
define handle h = 0
msgopen (h)

Msgreturndatasize Command

The **msgreturndatasize** command retrieves the size (in bytes) of the return data that the JTAG message generates when scanned to the target devices.

Syntax

[result =] msgreturndatasize (msg-handle, data-size-var, device-array)

[result =] msgreturndatasize (msg-handle, data-size-var, device-id)

Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the IR scan was successfully added to the JTAG message. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is a reference parameter that is modified by msgopen. After msgopen completes, msg-handle contains an unique value that identifies this JTAG message.
data-size-var	An ord4 debug variable that contains the return data size (in bytes). This is a reference parameter and is modified by the msgreturndatasize command.
device-array	The previously defined ord2 array of device ids (so that multiple devices can be scanned simultaneously).
device-id	The device id for a single target system device. This is a boundary scan list device position.

Discussion

Use the msgreturndatasize command to determine the size (in bytes) of the array to pass in to msgdata. The return-array used in the msgscan and msgdata commands must be at least this large, or an error is returned. The command returns an error if called before the JTAG message has been closed.

Different sets of devices can be used with this command as the number of bytes returned depends on the devices specified in the scan command. An error is returned if the devices in the device-array are not on the same debug port; no other verification of the list is done.

Example

Command input:

```
// Read JTAG ID from processor
define handle h = 0
define ord2 device = 0
msgopen(h)
msgir(h, 4, 2)
msgdr(h, 20, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
```


Msgscan Command

The **msgscan** command sends a closed JTAG message to the emulator and scans it to the target devices.

Syntax

[result =] msgscan (msg-handle, device-array [, return-array])

[result =] msgscan (msg-handle, device-id [, return-array])Where:

result	A boolean variable that contains the return value of this command. TRUE indicates the scan was successful. FALSE indicates an error occurred, such as the JTAG message was not found.
msg-handle	The name of a previously defined debug variable of type handle. This is the variable that was passed in to msgopen when the JTAG message was created.
device-array	The previously defined ord2 array of device IDs (so that multiple devices can be scanned simultaneously).
device-id	The boundaryscanlist device position (ord2) for a single target system device.
return-array	A previously defined array of ord1, ord2, or ord4 in which the data returned from the scan of target device(s) is stored

Discussion

Use the **msgscan** command to send a JTAG message to the emulator. This command returns an error if the devices in the device-array are not on the same debug port; no other verification of the list is done. The command returns an error if called before the JTAG message has been closed.

If a return-array is specified, the command waits for the JTAG message to complete and copies the scan data to the array. If the command is used without a return-array, the JTAG message begins to scan. The [msgdata](#) command must be used to access the return data.

Example

Command input:

```
// Read JTAG ID from processor
define handle h = 0
define ord2 device = 0
msgopen(h)
msgir(h, 4, 2)
msgdr(h, 20, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
define ord1 data[count]
msgdata(h, data)
data
msgdelete(h)
```

MsR Command

The **msr** command displays or changes the contents of a specified MSR (Model Specific Register).

Syntax

`msr (n) [= expr]`

Where:

(n) Specifies an MSR number. The use of parentheses is required.

expr Specifies a 64-bit number. Using this option changes the contents of the selected MSR

Examples

1. The following example shows how to display the contents of a specified MSR.

Command input:

msr (5)

Result:

12345678H#87654321

2. The following example shows how to change the contents of a specified MSR.

Command input:

msr (5) = 12345678#87654321 ; msr 5

Result:

12345678H87654321H

num_devices

Display the number of JTAG devices in the target system.

Syntax

num_devices

Discussion

Use the num_devices control variable to determine the number of JTAG devices in the target system. Using the control variable in an expression returns the current value.

This control variable has been replaced by num_jtag_devices and is provided only to support legacy operation. Please use num_jtag_devices instead.

Example 1

To check the number of JTAG devices in a system with 9 devices:

Command input:

```
num_devices
```

Result:

9

Example 2

To use the control variable in an expression:

Command input:

```
define ord2 o2NumDev  
o2NumDev=num_devices  
o2NumDev
```

Result:

9

num_jtag_devices

Display the number of JTAG devices in the target system.

Syntax

num_jtag_devices

Discussion

Use the num_jtag_devices control variable to determine the number of JTAG devices in the target system. Using the control variable in an expression returns the current value.

Example 1

To check the number of JTAG devices in a system with 9 devices:

Command input:

```
num_jtag_devices
```

Result:

9

Example 2

To use the control variable in an expression:

Command input:

```
define ord2 o2NumJtagDev  
o2NumJtagDev=num_jtag_devices  
o2NumJtagDev
```

Result:

9

num_processors

Display the number of target processors.

Syntax

```
[variable=] num_processors
```

Where:

variable is a debug variable to hold the return value.

Discussion

Use the num_processors control variable to determine the number of processors in the target system. Using the control variable in an expression returns the current setting.

Example 1

To check the number of processors in a system with eight processors:

Command input:

```
num_processors
```

Result:

```
8
```

Example 2

To use the variable in an expression:

Command input:

```
define ord2 o2NumPro
o2NumPro=num_processors
o2NumPro
```

Result:

```
8
```

pause

Suspend macro execution until a key is pressed.

Syntax

[variable =] pause

Where:

variable is a debug variable for storing the character entered.

Discussion

Use the pause command to suspend macro execution until a key is pressed.

NOTE: The following keys will not complete a pause: F1-F12, Page up, Page down, Num Lock, Caps Lock, Scroll Lock, Shift, Ctrl, Alt.

Example 1

To delay execution of a macros and save the character entered:

Command input:

```
puts("waiting for user input:\n")
define char ch = pause
```

Result:

```
waiting for user input:
```

Example 2

To delay execution of a macro without saving the character entered:

Command input:

```
puts("press any key to continue:\n")
pause
```

Result:

```
press any key to continue:
```

Physical Command

The **physical** command converts an address to a physical address.

Syntax

`physical (addr)`

Where:

(addr) Specifies an address to be translated into a physical address. The parentheses are optional.

Discussion

Use the **physical** command to convert the specified `addr` to a physical address using the address translation rules currently in force in the target system (e.g., paging or current processor mode).

- If you enter a physical address, it is returned unchanged.
- If you enter a virtual address, it is first translated to a linear address and then to a physical address. If the translation is not allowed, an error message is returned. See the **Addresses** in this "Commands and Topics" section for more information on virtual address translation.
- If you enter a linear address in Page-Protected mode (the PG bit =1 and the PE bit=1), the page tables accessible using the current page directory base (CR3) are searched for a page containing the specified linear address. The search begins with the first entry in the page directory table (PDT). The first match found is reported. If no match is found, an error message is returned. If paging is not enabled (PG bit = 0), then the linear address is returned.

Example

The following example shows how to translate a virtual address.

Command input:

physical 1000:1234

Result:

11234P

Port Command

The **port** command displays or changes the contents of an 8-bit I/O port.

Syntax

```
[[px]] port (io-addr) [= expr]
```

Where:

- [px] Viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain, where x is the number (0, 1, 2, or 3).
- io-addr Specifies a 16-bit address in the processor I/O space. The available io-addr range is 0 to 0ffffh. The use of parentheses is optional.
- expr Specifies a 8-bit number or expression. Using this option writes the data to the specified I/O port.

Discussion

Use the **port** command to read from and write to the specified I/O port with the specified 8-bit data. You can access up to 64K 8-bit ports.

Examples

1. The following example shows how to display and change the contents of the I/O port at address 88h.

Command input:

```
port 88h
```

Result:

```
0088H FFH ". "
```

Command input:

```
port 88h = 0abh
port 88h
```

Result:

```
ABH ". "
```

2. The following example shows how to assign one port value to another port.

Command input:

```
port 90h = port 88h
```

Result:

```
FF55H
```


3. The following example shows how to create a debug variable named portvar and assign a port value to it.

Command input:

```
define ord1 portvar  
portvar = port 90h  
portvar
```

Result:

```
FFH //Result may vary
```

Pow Command

The **pow** command raises the value specified by `expr` to the power specified by `pow-expr`. The **pow** function returns an invalid value when the correct value would overflow. The return value is of type `real8`.

Syntax

```
[ name = ] pow (expr, pow-expr)
```

Where:

name	Specifies a debug object of type <code>real8</code> to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
expr	Specifies a number or an expression of type <code>real8</code> that is to be raised to the power of <code>pow-expr</code> . The parentheses are required.
pow-expr	Specifies a number or an expression of type <code>real8</code> to which the the power <code>expr</code> is to be raised. The parentheses are required.

Note: Values returned by this command (a math function) are in `real8` or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Print Command

The **print** command is used to print trace cycles from the **Trace** window. All or a portion of the trace can be printed.

Syntax

print cycles [startCycle to endCycle | startCycle length count]

Where :

startCycle	Trace cycle state number
endCycle	Trace cycle state number
count	Integer expression

Examples

Command inputs:

print cycles // prints the entire trace buffer

print cycles 0 length 100 // print 100 cycles of trace beginning at state 0

print cycles -100 to 0 // print trace beginning at state -100 and ending at state 0

Type topic text here.

Print Cycles Command

The **print cycles** command formats and prints the contents of the trace buffer.

Syntax

`print cycles [range]`

Where

<code>print cycles</code>		specifies a parameter to format and print
<code>[range]</code>	<code>all</code>	default
	<code>to</code>	defines a specific range of addresses
	<code>length</code>	defines a specific length from a particular address

Discussion

The **print cycles** command "prints" or saves the entire trace buffer to a file named "trc.txt" in the DWD path; it does not dump the information directly into the **Commands** window. If the "trc.txt" file does not exist, one will be created. If the file does exist, it will be overwritten with the new **print cycles** command.

The range is an optional cycle range. If the range is omitted, print cycles default to all cycles.

Examples

Command inputs:

print cycles all

print cycles -10 to -1

print cycles -10 length 10

Note: Length may be shortened to len. All, to, and length are not case sensitive.

Availability

This command is available with all Arium TRC emulators.

printf

Write formatted output to the Command window.

```
printf ("format" [, expr ] [...])
```

Where:

"format" is a list of conversion specifications that corresponds to the like-ordered items in the list of expressions. Quotation marks are required.

expr is an expression that is evaluated and displayed.

Discussion:

Use the printf function to write formatted output to the Command window. The printf command is similar to the C-language printf routine.

The format string is comprised of a series of conversion specifications of the form:

```
"% [flags] [width] [.precision] [data-length] conversion-operator"
```

These fields are defined as follows:

Flags

The flags element can be one of the following:

Flag	Description
- (minus)	causes the output to left-justify.
+ (plus)	causes signed numeric output to always display a sign.
0 (zero)	causes the field to zero fill.
(space)	causes the field to space fill (default)

Width

Use the digits 0 through 9 to define the minimum width of a field. Use an asterisk (*) to assign this value from an expression.

Precision

Use the digits 0 through 9 to define the decimal precision of a field.

Data-length

The following list gives the data size operators and their descriptions. If not specified the length is determined from the expression itself.

Operator	Description
----------	-------------

- h typecasts the corresponding argument to a 16-bit value.
- l, L typecasts the corresponding argument to a 32-bit value.
- l64 typecasts the corresponding argument to a 64-bit value (SourcePoint extension).
- l128 typecasts the corresponding argument to a 128-bit value (SourcePoint extension).

Conversion-operator

The following list gives the conversion operators and their descriptions.

Operator	Description
d, i	displays corresponding argument in signed decimal
u	displays corresponding argument in unsigned decimal
o	displays corresponding argument in octal
x, X	displays corresponding argument in hexadecimal
e, E, f, g, G	displays corresponding argument as floating-point
c	displays corresponding argument as a character
s	displays corresponding argument as a null-terminated string
b, B	displays corresponding argument as a boolean (SourcePoint extension).
y, Y	displays corresponding argument in binary (SourcePoint extension).
D	displays corresponding argument in the the current default number base (SourcePoint extension).
p	displays corresponding argument as a pointer (SourcePoint extension).

Escape Characters

The printf function accepts the following escape characters. The leading backslash is required.

Escape Character	Description
\b	backspace
\f	form-feed
\n	new line (flushes output to the Command line)
\r	carriage return
\t	tab
\\	backslash
\"	double quote
\nnn	a three-digit octal number that represents the ASCII value of the character. This value enables characters that are not directly available from the keyboard to be inserted into a character string.
\xnn	a two-digit hexadecimal number that represents the ASCII value of the character. This value enables characters that are not directly available from the keyboard to be inserted into a character string. The x indicates that a hexadecimal number follows

Example 1

To print a simple message to the screen (the \n character is required to flush output to the Command line):

Command input:

```
printf ("This is my message.\n")
```

Result:

This is my message.

Example 2

To use character strings to print a date:

Command input:

```
define nstring date = "Saturday"  
define ord1 day = 3  
printf("Today is %s, the %drd of July.\n",date,day)
```

Result:

Today is Saturday, the 3rd of July.

Example 3

To print a message with an audible beep (007 octal is the ASCII code for beep):

Command input:

```
printf ("\007ATTENTION: Emulation has stopped \n")
```

Result:

ATTENTION: Emulation has stopped

Proc Command

The **proc** command displays a debug procedure

Syntax

```
proc proc-name
```

Discussion

The **proc** command displays a debug procedure (proc) that was previously defined with the **define** command.

Example

Command input:

```
define proc btmon( )
{
define int2 svViewpoint = viewpoint // save current viewpoint
define int2 processor
for (processor = 0; processor < 4; processor++)
if (IsProcAvail(processor)) {
viewpoint = processor
debugctlmsr=0#41
}
viewpoint = svViewpoint // restore original viewpoint
}
proc btmon
```

Result:

```
{
define int2 svViewpoint = viewpoint // save current viewpoint
define int2 processor
for (processor = 0; processor < 4; processor++)
if (IsProcAvail(processor)) {
viewpoint = processor
debugctlmsr=0#41
}
viewpoint = svViewpoint // restore original viewpoint
}
```


ProcessorControl Control Variable

The **ProcessorControl** control variable specifies which processors in the target system are to be controlled by the emulator.

Syntax

ProcessorControl [= expr]

Where:

expr	a mask value indicating which processors are to be controlled by the emulator
------	---

Discussion

This control variable allows some processors to be under the control of the emulator while other processors are left alone. The mask value includes one bit per processor with Bit 0 corresponding to the first processor in the JTAG chain, Bit 1 corresponding to the second processor in the JTAG chain, and so on. A value of 1 indicates the processor is controlled by the emulator. A value of 0 indicates the emulator will not access that processor.

Typing **ProcessorControl** without an expression displays the current mask value.

Notes:

This control variable is only applicable in multiprocessor targets.

- A mask value of 0 is not allowed.
- Upper bits (beyond the number of processors in the JTAG chain) are ignored.
- The **Viewpoint** view displays a status of “unavailable” for processors that are not under control of the emulator.
- Masked processors always have a “not ready” status.

Examples

1. The first example show how to display the current mask value:

Command input:

ProcessorControl

Result:

3 // P0 and P1 are under control of the emulator

2. The following example shows how to enable run control of only the second processor in a two processor target.

Command input:

ProcessorControl = 2

Note: Commands are not case sensitive. Capitalized letters are used here to make the command name clearer.

ProcessorFamily Function

The **processorFamily** function displays a string identifying the family to which the processor belongs.

Syntax

processorfamily

Discussion

Use **processorFamily** to get a unique string that identifies the family of the current processor. In a multiprocessor system, the family of the processor with the current viewpoint is displayed. This function is read-only.

Example

Command input:

processorfamily

Result:

P6

Processors Control Variable

The **processors** control variable displays the number of processors present in the target system.

Syntax

processors

Discussion

The **Processors** control variable displays the number of processors in the current base setting.

Example

Note: Assume two processors are present.

Command input:

```
base=dec
processors
```

Result:

 $2T$

Command input:

 $2T$

Result:

 $2T$

Command input:

```
base=bin
processors
```

Result:

000000000000000010Y

Command input:

000000000000000010Y

Result:

[illegible]

ProcessorMode Command

The **processorMode** command displays a string identifying the operating mode of the current processor

Syntax

processormode

Discussion

Use the processormode command to get a unique string that identifies the mode of the current processor. In a multiprocessor system, the mode of the processor with the current viewpoint is displayed. This control variable is read-only.

Examples

Input:

processormode

Result:

Real

ProcessorType Function

The **processorType** function displays a string identifying the processor.

Syntax

processortype

Discussion

Use **processorType** to get a unique string that identifies the current processor. In a multiprocessor system, the identifier of the processor with the current viewpoint is displayed. This function is read-only.

Example

Command input:

processortype

Result:

x86 Family F Model 3 (P/N/P)

projectpath

Display the project file path

Syntax

[variable=] project path

Where:

variable is an nstring debug variable to receive the answer

Discussion

The **projectpath** control variable contains a string that is the full path to the directory where the SourcePoint project file is located. The string is terminated with a final slash/backslash path delimiter. This variable can be used to avoid hard-coded file paths by referencing them relative to the SourcePoint project file directory.

Example

Assume the current project file is C:\Program Files\Arium\SourcePoint\sp.prj.

Command input:

```
define nstring mymac = projectpath + "mac\big.mac";  
mymac
```

Result:

```
" C:\Program Files\Arium\SourcePoint\mac\big.mac"
```

Putchar Command

The **putchar** command displays a character in the **Command** window.

Syntax

putchar (char-expr)

Where:

(char-expr) specifies a quoted character or an expression that evaluates to a character

Example

Command input:

```
define char cvar = 'a'  
putchar (cvar); putchar('cvar+1'); putchar("\n")
```

Result:

ab

Puts Command

The **puts** command displays a string on the screen.

Syntax

`puts (string-expr)`

Where:

string-expr	Specifies an nstring variable, quoted string constant, or an expression that evaluates to a string. The parentheses are required.
-------------	---

Examples

Command input:

```
define nstring date = "6/2/53\n"  
puts(date)
```

Result:

6/2/53

Command input:

```
puts ("string constant \n")
```

Result:

string constant

Rand Command

The **rand** command returns a pseudo-random number of int4 data type. If you previously executed the **srand** function, the **rand** function uses the output of the **srand** function as its source expression. If the **srand** function has not been previously executed, the **rand** function generates a less-random number.

Syntax

```
[ name = ] rand ()
```

Where:

name Specifies a debug object of type int4 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

Example

The following example illustrates the **srand** and **rand** functions:

Command input:

```
base = 10t  
define int4 card  
srand (3)  
card = rand ()  
card
```

Result:

```
16838T // Result may vary
```

Command input:

```
rand ()
```

Result:

```
5758T // Result may vary
```

ReadSetting Command

The **readSetting** command is used to read settings within SourcePoint.

Syntax

```
ord4 readSetting(type, name)
```

Where

type	an nstring or string constant specifying the type of setting
name	an nstring or string constant specifying the setting name
value	an ord4 specifying the value to assign to name

Discussion

This **readSetting** command is used to read settings within SourcePoint. Usually, these settings are changed via the UI (e.g., the **Emulator Configuration** dialog box). There are times, however, when it is convenient to be able to change these settings within a macro file.

The type argument specifies the type of setting to change. Currently, the only type supported is “em” for emulator configuration settings.

The name argument specifies the name of the setting to change. The name is not what is displayed in the UI, but rather the name used in the project file. Names can be obtained by looking in the project file in the Emulator Configuration section.

The value argument can be obtained by changing the emulator configuration setting in question and looking in the project file. For checkbox settings, the value is TRUE or FALSE. For radio buttons, the value usually (but not always) is the zero-based index of the button selected. For drop down lists, the value usually (but not always) is the zero-based index of the entry selected. The safest way to determine value is to look in the project file.

Note: Values in the project file are usually decimal. Regardless, values in the command language are specified using the current command language input radix (specified with the base control variable). If the input radix is hex, and you want to specify a value of 200 decimal, then you need to use 200T as 200 is interpreted as 200H = 512 decimal.

Example

The following example returns the Adaptive TCK setting. The possible values are 0, 1 and 2 corresponding to which radio button is selected in the UI.

Command input:

```
readsetting("em", "AdaptiveTck")
```

Result:

```
00000002H    // 2 = "Auto-detect use of adaptive TCK"
```

reg

Display or change the contents of a specified register.

Syntax

```
[[px]] reg-name[=expr]
```

```
[[px]] reg-name.bit-name [=expr]
```

Where:

[px]	is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override
reg-name	specifies the name of a register listed in the register keywords table. For more information, see "Registers Keywords Table" in the <i>Appendix</i> .
bit-name	specifies the name of a bit within a register.
expr	specifies a number or expression.

Discussion

Use the reg command to set or display the contents of a specified register. Register contents are displayed in the current number base. Processor register names can also be used in expressions.

Use the assignment operator (=) followed by an expression to assign a value to the register.

Example 1

To set the value of EAX to 20:

Command input:

```
EAX = 20
EAX
```

Result:

```
20H
```

Example 2

To display the value of the carry flag bit in eflags:

Command input:

```
EFLAGS.CF
```

Result:

TRUE

Reload Command

The **reload** command repeats the last **load** command

Syntax

```
reload
```

Discussion

The **reload** command repeats the last executed **load** command. The filename and arguments specified on the **load** command will be the same.

ReloadProject Command

The **ReloadProject** command reloads the current SourcePoint project file.

Syntax

ReloadProject ()

Discussion

The **ReloadProject** command reloads the current project file. This command causes SourcePoint to reestablish communications with the emulator.

Example

Command input:

ReloadProject()

Remove Command

The **remove** command deletes procedures loaded into the **Command** window environment.

Syntax

remove object

Where:

object Any emulator data base item.

wild- Specifies the name of a debug object. You can use the * and ? as wild card characters.

name When wild-name is a literally definition, the literally option must proceed it.

Discussion

Use the **remove** command to erase debug objects from the emulator database that have been defined with the define or redefine commands. A warning message is displayed if wild-name does not exist. Use the **show** command to verify removal of the specified objects.

Examples

1. The following example shows how to remove all debug variables starting with the letters var.

Command input:

*remove debugvar var**

2. The following example shows how to remove a single debug variable.

Command input:

show

Result:

btmon()

cregs()

dregs()

regs()

ABC()

Command input:

remove ABC

show

Result:*btmon()**cregs()**dregs()**regs()*

Reset Command

The **reset** command resets specified target system functions.

Syntax

reset emulator

reset [target [0 | 1]]

reset tap [(jtag chain)]

Where:

emulator	resets the emulator
target	resets the target and the target processor; entering reset target while running halts execution. (See details below.)
tap	resets the target Test Access Port (TAP) by asserting/deasserting the TRST signal
jtag chain	an optional parameter that specifies on which jtag chain to assert the reset

Discussion

Use the **reset** command to reset the target, emulator, or JTAG chain. All active SourcePoint windows are refreshed with the reset command regardless of the option used.

When the reset target command is used, it implies waiting for the emulator to return a status 18 (stopped and ready to debug) for the target. If this condition is not met, a macro containing reset waits indefinitely. If the target argument is used with a value of 0, the macro continues and does not wait for a stopped status. (See examples below.)

Examples

1. The following example shows how to reset the target system. All three forms behave the same way. The macro being executed pauses until the emulator senses the stopped state.

Command input:

```
reset
reset target
reset target (1)
```

2. The following example also shows how to reset the target system. In this case, the macro being executed proceeds no matter what state is returned from the emulator.

Command input:

```
reset target (0)
```

3. The following example shows how to reset all jtag chains.

Command input:

reset tap

4. The following example shows how to reset the jtag chain 0.

Command input:

reset tap (0)

5. The following example shows how to reset the emulator.

Command input:

reset emulator

Right Command

The **right** command extracts a number of characters from the end of a string.

Syntax

`right (string-expr, n)`

Where:

string-expr	specifies an nstring variable or string constant
n	specifies the number of characters to extract

Discussion

The **right** command returns a substring from the end of a string. If the number of characters to extract is greater than the length of the string, then the entire string is returned.

Example

Command input:

```
base = 10t           // Set number base to decimal
define nstring month = "January"
define nstring temp =right(month, 3)
temp
```

Result:

`"ary"`

restart

Re-initialize processor registers, allowing for faster reload of a program.

Syntax

restart

Discussion

The **restart** command provides a faster way to load a program, performing the equivalent of the INIT option of the [load](#) command. Load speed is improved because the **restart** command does not load code or symbols; it only re-initializes processor registers.

For ELF/Dwarf files, only the EIP is initialized.

For OMF files, all the registers stored in the TSS are initialized.

This command restarts the last program loaded. If multiple programs were loaded, only the last one is affected.

Example

This example assumes a program named “test.omf” has been loaded prior to using the **restart** command.

Command input:

```
restart    // restart test.omf
```

Safemode Control Variable

The **safemode** control variable displays or changes whether target memory reads are suppressed for areas designated as DRAM by the memory map.

Syntax

safemode [= bool-cond]

Where:

bool-cond specifies a number or an expression that must evaluate to true (non-zero) or false (zero)

Discussion

Use the **safemode** control variable to disable automatic target memory reads before DRAM has been configured. The default setting for **safemode** is false. Entering the control variable without an option displays the current setting.

If **safemode** is set to false, all target memory reads are allowed. If **safemode** is set to true, SourcePoint suppresses a target memory read if the address range falls within a DRAM range in the memory map.

Memory accesses by commands run in the **Command** window are not affected by safemode. Safemode is bypassed when accessing memory in this way.

If **safemode** is enabled, the title bar in SourcePoint will display (safe mode) after the project file path.

Examples

Example 1

Command input:

```
safemode           // display the current setting
```

Result:

```
FALSE
```

Example 2

Command input

```
safemode=true     // enable safemode
```

Command input

```
safemode           // display the new setting
```

Result:

```
TRUE
```

SequenceType and SequenceCount Commands

The **sequenceType** and **sequenceCount** commands are used to specify a bus analyzer trigger sequence.

Syntax

`sequenceType [= seq-type-spec]`

`sequenceCount [= counter-value]`

Where seq-type-spec is one of the following:

off	no bus sequence
T1	trigger when T1 occurs
T1 then T2	trigger when T1 then T2 occurs
T1 then T2 then T3	trigger when T1 then T2 then T3 occurs
T1 then T2 then T3 then T4	trigger when T1 then T2 then T3 then T4 occurs
count of T1	trigger when T1 occurs a certain number of times
T1 and not T2	trigger when T1 occurs without T2

Where:

counter-value specifies a 16-bit number or expression (only used if **sequenceType** is set to **Count of T1**)

Discussion

The **sequenceType** command is used to select or display the bus analyzer trigger sequence. The trigger sequence can also be set in the breakpoint window.

A Trigger Sequence is a combination of terms (T1, T2, T3 or T4) that defines when a trigger will occur. Individual bus breakpoints are tagged as participating in term 1 (T1), term 2 (T2), etc. If more than one bus break has the same sequence term, then they are "or'ed" together (if either occurs, that portion of the trigger sequence is satisfied). If no Trigger Sequence is specified (**sequenceType** is none), then all breakpoints are "or'ed" together.

For instance, to trigger when a value of 55H is written to address 1000H, followed by a value of AAH written to address 2000H, select the two-level sequence type **T1 then T2**. Then define two bus breakpoints, one with Location=1000H, Data=55H, Bus=T1, and the other with Location=2000H, Data=0AAH, Bus=T2.

If the sequence type: **Count of T1** is selected, then the **sequenceCount** command can be used to specify the value of the counter at which to trigger. The maximum counter value is 65535.

Limitations

- In multi-level sequences, up to 2 bus breaks can be specified as T1 or T2. Only one bus break can be specified as T3 or T4.
- In the **count of T1** and **T1 and not T2** sequences, only one bus break can be specified as T1 or T2.
- If start / stop data qualification is used (see [dataqualmode](#)), then the sequence **T1 then T2 then T3 then T4** is not available.

Examples

Example 1

Command input:

sequenceType

displays the current trigger sequence type

Result:

off

Command input:

sequenceCount

displays the current trigger sequence counter value

Result:

10T

Command input:

*busbreak = fetch, location=1000:1200, bus=T1
busbreak = fetch, location=1000:1234, bus=T2
sequenceType = T1 then T2
go*

*set a bus break on a fetch at 1000:1200
set a bus break on a fetch at 1000:1234
select a 2 level trigger sequence
start the processor, a trigger when
1000:1200 is fetched, then 1000:1234 is
fetched*

Example 2

Command input:

busbreak=data read, location=1000:1234, size=byte

*set a bus break on a byte read at
1000:1234*

sequenceType = count of T1

*set trigger sequence to count byte reads at
1000:1234*

sequenceCount = 100

set sequence trigger on 100th read

go

start processor, trigger after 100 reads

Availability

These commands are available with all Arium TRC emulators.

Shell Command

The **shell** command executes an operating system command.

Syntax

```
shell [shell-command]
```

Where:

shell-command specifies any valid shell command

Discussion

The **shell** and **dos** commands are equivalent on Windows; however, on Linux the command output is displayed in the console window used to start SourcePoint.

Text to be passed to the host operating system is expanded with the currently defined literal definitions. To suppress this literal substitution, enclose aliases in single quotes.

Examples

Command input:

```
shell cp c:/tmp/test.list /save
```

Command input:

```
shell ls -al
```

Show Command

The **show** command displays the definition and value of debug objects.

Syntax

```
show [debug | data-type | debugvar | literally | alias | proc | wildcard-name]
```

Where:

debug	If entered without a specific debug object type, it will display all debug definitions.
data-type	Displays the current value of the specific variable type (see Data Types Command).
debugvar	Displays the current value of all debug variables.
literally	Displays all literally definitions.
alias	Same as literally.
proc	Displays all debug procedure declarations.
wildcard-name	Specifies the name of an existing debug object. This option limits the display to a specified subset of all available debug objects. You can use the * and ? as wildcard characters. When wild-name is a literally definition, the literally option must proceed it.

Discussion

Use the **show** command to display the value and definitions of debug objects.

Examples

1. The following example shows how to list all of the alias definitions currently defined.

Command input:

```
show alias
```

Result:

```
bbl_cr_mc4_addr    alias    "MC4_ADDR"
bbl_cr_mc4_ctrl    alias    "MC4_CTRL"
bbl_cr_mc4_misc    alias    "MC4_MISC"
```

etc.

2. The following example shows how to display all type ord2 debug variables beginning with the letters var:

Command input:

```
show ord2 var*
```

Result:

var1 ord2 0003H
var2 ord2 0005H

3. The following example shows how to display only a debug procedure declaration for *proc1*.

Command input:

show proc1

Result:

ord4 proc1 (ord2 arg1, ord4 arg2)

Sin Command

The **sin** command returns the sine of a radian expression.

Syntax

[name =] sin (expr)

Where:

name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

(expr) Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

sleep

The **sleep** command suspends input to the **Command** window for the specified number of seconds.

Syntax

`sleep(expr)`

Where:

`expr` is a number or an expression which evaluates to a value between .1 and 60 seconds.

NOTE: `expr` uses the default input radix (base control variable). If base = hexadecimal, then `sleep(10)` sleeps for 16 seconds.

Example #1

To suspend Command window input for 5 seconds:

Command input:

`sleep(5)`

Example #2

To suspend Command window input for 2/10 seconds:

Command input:

`sleep(.2)`

Softbreak, Softremove, Softdisable, Softenable Commands

These commands are used to set, clear, display, enable, and disable soft breakpoints

Syntax

`softbreak`

`softbreak = [sts-spec,] location-spec`

`softremove [all]`

`softremove = location-spec`

`softenable = location-spec`

`softdisable [all]`

`softdisable = location-spec`

Where:

location-spec:	[location] = vls-addr
sts-spec:	{ e[nabled] d[isabled] }

Discussion

The **softbreak** command sets and displays soft breakpoints (soft breaks). **Softbreak** with no arguments displays a list of the current soft breaks. There are 64 debug register breakpoints available.

The **softremove** command removes any or all of the soft breaks. **Softremove** with no arguments removes all soft breaks. **Softremove** with a location specified removes a single soft break.

The **softenable** command enables a softbreak at the specified location. The **softdisable** command disables a softbreak at the specified location.

Soft breaks can also be set, displayed, etc. from the **Breakpoints** and **Code** windows.

Examples

Command inputs:

<code>softbreak</code>	<code>// display current soft breaks</code>
<code>softbreak = location=1000:1234</code>	<code>// set a soft break at location//1000:1234</code>
<code>softremove</code>	<code>// remove all soft breaks</code>
<code>softremove = location=1000:1234</code>	<code>// remove soft break at 1000:1234</code>

```
softdisable // disable all soft breaks  
softdisable = location=1000:1234 // disable soft break at 1000:1234  
softenable = loc=1000:1234 // enable soft break at 1000:1234
```

sprintf

Write formatted output to an nstring variable.

Syntax

```
sprintf ( nstring, format [, expr [ , . . . ] ] )
```

Where:

nstring is an nstring debug variable.

format is a quoted string of characters that determines the format of the display. The format can contain two types of characters: ordinary characters and conversion specification characters.

expr is an expression that is evaluated and displayed.

Discussion

Use the sprintf function to write formatted output to an nstring debug variable. The sprintf function is similar to the C-language sprintf routine. See [Printf Command](#) for more information.

Example

Command input:

```
define nstring mystring
define ord4 o4test = 3
sprintf (mystring, "this is test #%%d", o4test)
mystring
```

Result:

```
this is test #3
```


Sqrt Command

The **sqrt** command returns the square root of an expression. **Sqrt** returns 0 (zero) when expr is negative.

Syntax

```
[ name = ] sqrt (expr)
```

Where:

name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

(expr) Specifies a number or an expression of type real8. The parentheses are required.

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

Srand Command

The **srand** command sets the starting point for generating a pseudo-random number using the **rand** command. Set the **rand** function to a random starting point by calling the **srand** function with an **expr** other than 1. Using 1 initializes a response.

Syntax

```
srand (expr)
```

Where:

(**expr**) Specifies a number or an expression of type **ord4**. The parentheses are required.

step

Execute one or more instructions.

Syntax

```
[[px]] step [into | over | out | branch] [step-cnt]
[[px]] step-cmd [step-cnt]
```

Where:

[px]	is a viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
step-cnt	specifies the number of instructions to step (1-255).
step-cmd	{ stepinto stepover stepoutof bstep istep lstep pstep }
stepinto	step into function calls
stepover	step over function calls
stepout	step out of a function call
bstep	step til the next branch instruction (Intel only)
istep	step into function calls (same as stepinto)
lstep	step out of a function call (same as stepover)
pstep	step over function calls (same as stepout)

Discussion

Use the step commands to step a processor one or more instructions. Step commands can also be executed from the Processor menu or the Processor toolbar.

You can control whether stepping takes place at the source level or machine level via the Code window. If a single Code window is open, then the display mode of that window controls how stepping is performed. If the display mode is Source, a line of source code will be stepped. If the display mode is Mixed or Disassembly, a single assembly language instruction will be stepped.

Interrupts can be enabled, or disabled during steps. This preference is set in Options | Emulator Configuration | General.

Breakpoints can be enabled or disabled during steps. This preference is set in Options | Emulator Configuration | General.

The source level step algorithm uses a combination of go's and steps depending on the instructions contained in the source line. During go operations, interrupts and breakpoints will be enabled.

The step out command sets a temporary breakpoint at the return address of the current function.

The step branch command steps until a branch instruction is executed, or until an exception or interrupt occurs. Conditional branches that are not taken will not terminate the step. This command is only available on Intel IA-32 processors.

If a step count larger than 255 is specified, then the step count is truncated. Note that the step count uses the default input radix. If the input radix is set to hex, then step 10 will step 16 times.

Examples

```
step                // step viewpoint processor a single instruction (step
into calls)

[p1]step            // step processor 1 (p1) once

step 5              // step 5 instructions (step into function calls)

step into 5         // step 5 instructions (step into function calls)

stepinto 5          // step 5 instructions (step into function calls)

step out            // step out of the current function

stepout             // step out of the current function

[p2]step over 5     // step p2 5 instructions (step over function calls)

step branch         // step til next branch instruction executed
```

Related topics

[go](#)
[stop](#)

Stop Command

Use the **stop** command to halt the processor.

Syntax

```
stop
```

Discussion

The stop command stops target program execution. It is the same as the halt command.

Example

Command input:

```
go til 1000:1034  
stop
```

Strcat Command

The **strcat** command appends the second string (string-expr2) to the right end of the first string (string-expr1). Copying of the second string starts from the left end of the string and continues until a null terminating character is copied. The string-expr2 is left unchanged.

Syntax

strcat (string-expr1, string-expr2)

Where:

string-expr1	specifies an nstring variable or an address of a string.
string-expr2	specifies a pointer expression (address), an nstring variable, or a string constant

Example

The following example illustrates the strcat function:

Command input:

```
base = 10t al // Set number base to decimal
define nstring month = "10"
define nstring date = "22."
strcat (date, month)
date
```

Result:

"22.10"

strchr

Find a character in a string.

Syntax

```
[variable = ] strchr(string, ch)
```

Where:

variable	is an nstring variable to receive the return value.
string	is an nstring variable, or string constant, to search.
ch	is the character to search for.

Discussion

The strchr function finds a character in a string. The return value is a substring containing the first instance of the character found, and the rest of the string following it. The return value can be assigned to an nstring variable, or displayed on the command line.

Example 1

Command Input:

```
define nstring test = "123456"  
strchr(test, '3')
```

Result:

```
3456
```

Example 2

Command Input:

```
define nstring strAnswer = strchr("123456", '4')  
strAnswer
```

Result:

```
456
```

Strcmp Command

The **strcmp** command compares two ASCII strings, character by character. The comparison stops when a mismatch is found or when a null character is encountered in one of the strings. The return value depends on the difference between the hexadecimal values of the characters at the stopping position.

Syntax

```
[ name = ] strcmp (string-expr1, string-expr2)
```

Where:

name	Specifies the debug object of type int2 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
string-expr1	Specifies an nstring variable, a quoted string constant, or an address of a string.
string-expr2	Specifies an nstring variable, a quoted string constant, or an address of a string.

Discussion

The **strcmp** command compares two ASCII strings, character by character. The comparison stops when a mismatch is found or when a null character is encountered in one of the strings. The return value depends on the difference between the hexadecimal values of the characters at the stopping position. The return value is one of the following:

-1	The final character in string-expr2 is greater than the final character in string-expr1.
0	The final character in string-expr2 is equal to the final character in string-expr1.
1	The final character in string-expr2 is less than the final character in string-expr1.

Example

The following example illustrates the strcmp function:

Command input:

```
base = 10t // Set number base to decimal
define nstring name1 = "rosenberg"
define nstring name2 = "rosenbaum"
define int4 order = strcmp (name1, name2)
order
```

Result:

```
1T
```


Command input:

strcmp (name2, name1)

Result:

-1T

Strcpy Command

The **strcpy** command copies the second string (string-expr2) into the first string (string-expr1) until the second string terminating null character is copied. This function overwrites any data in the first string. The second string remains unchanged.

Syntax

strcpy (string-expr1, string-expr2)

Where:

string-expr1	specifies an nstring variable or an address of a string
string-expr2	specifies an nstring variable, a quoted string constant, or an address of a string

Example

The following example illustrates the strcpy function:

Command input:

```
base = 10t // Set number base to decimal
define nstring month = "October"
define nstring year = "1990"
define nstring date
strcpy (date, month)
date
```

Result:

October

Command input:

```
strcpy (date, year)
date
```

Result:

1990

String [] (Index Into String) Functions

The [] operator returns the nth character in a string. If the index specified is beyond the end of the string, an error message is displayed.

Syntax

```
[char=] string-expr [index]
```

Where:

string-expr	specifies an nstring variable or a quoted string constant
Index	the character position to return

Examples

Command input:

```
define nstring myString = "Hi There!"
myString [0]
```

Result:

```
'H'
```

Command input:

```
define char myChar
myChar = myString [3]
myChar
```

Result:

```
'T'
```

Command input:

```
myString[100]
```

Result:

```
[ ] operator: invalid index
```

Strlen Command

The **strlen** command returns the length of an ASCII string length, excluding any null terminating character.

Syntax

```
[ name = ] strlen (string-expr)
```

Where:

name	Specifies the debug object of type ord4 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
(string-expr)	Specifies an nstring variable, a quoted string constant, or an address of a string. The parentheses are required.

Example

The following example illustrates the strlen function:

Command input:

```
base = 10t // Set number base to decimal
define nstring month = "October"
define nstring year = "1990"
define int4 ans
strlen (month)
```

Result:

```
7T
```

Command input:

```
ans = strlen (year)
ans
```

Result:

```
4T
```

Strncat Command

The **strncat** command appends the specified number of characters (*expr*) from the second string (*string-expr2*) to the right end of the first string (*string-expr1*). Copying of the second string starts from the left end of the string and continues until a null terminating character is copied or the specified number of character have been copied. The *string-expr2* is left unchanged.

Syntax

```
strncat (string-expr1, string-expr2, expr)
```

Where:

<i>string-expr1</i>	specifies an nstring variable or an address of a string.
<i>string-expr2</i>	specifies a pointer expression (address), an nstring variable, or a string constant
<i>expr</i>	specifies a number or an expression of type int4 that specifies the maximum number of characters to concatenate.

Example

The following example illustrates the strncat function:

Command input:

```
base = 10t                /* Set number base to decimal
define nstring month =
"10.86"
define nstring date = "22."
strncat (date, month, 2)
date
```

Result:

```
"22.10"
```

Strncmp Command

The **strncmp** command compares a specified maximum number of characters (*expr*) in two ASCII character strings. The comparison stops when a mismatch is found, when a null character is encountered in one of the strings, or when the specified number of character positions have been compared.

Syntax

```
[ name = ] strncmp (string-expr1, string-expr2, expr)
```

Where:

name	Specifies the debug object of type int2 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
string-expr1	Specifies an nstring variable, a quoted string constant, or an address of a string.
string-expr2	Specifies an nstring variable, a quoted string constant, or an address of a string.
expr	Specifies a number or an expression that specifies the maximum number of characters to compare.

Discussion

The **strncmp** command compares a specified maximum number of characters (*expr*) in two ASCII character strings. The comparison stops when a mismatch is found, when a null character is encountered in one of the strings, or when the specified number of character positions have been compared. The return value depends on the difference between the hexadecimal values of the characters at the stopping position. The return value is one of the following:

-1	The final character in string-expr2 is greater than the final character in string-expr1.
0	The final character in string-expr2 is equal to the final character in string-expr1.
1	The final character in string-expr2 is less than the final character in string-expr1.

Example

The following example illustrates the strncmp function:

Command input:

```
base = 10t // Set number base to decimal
define nstring name1 = "rosenbaum"
define nstring name2 = "rosenberg"
define nstring name3 = "rosen"
strncmp (name1, name2, 7)
```

Result:

-1T

Command input:

strncmp (name1, name3, 9)

Result:

1T

Command input:

strncmp (name1, name3, 3)

Result:

0T

Strncpy Command

The **strncpy** command copies the specified maximum number of characters (*expr*) from the second string (*string-expr2*) to the first string (*string-expr1*). Copying stops when a null terminating character is copied or when the number of characters specified have been copied. If *expr* is greater than the length of *string-expr2*, the *string-expr1* resulting from the copy is *string-expr2*.

Syntax

```
strncpy (string-expr1, string-expr2, expr)
```

Where:

<i>string-expr1</i>	specifies an <i>nstring</i> variable or an address of a string
<i>string-expr2</i>	specifies an <i>nstring</i> variable, a quoted string constant, or an address of a string
<i>expr</i>	specifies a number or an expression of type <i>int4</i> that specifies the maximum number of characters to copy

Example

The following example illustrates the *strncpy* function:

Command input:

```
base = 10t           // Set number base to decimal
define nstring month = "October"
define nstring year = "1990"
define nstring date
strcpy (date, month, 3)
date
```

Result:

```
"October"
```

Command input:

```
strcpy (date, year, 7)
date
```

Result:

```
"1990"
```


`_strdate`

Copy the current system date to an nstring variable.

Syntax

```
[variable = ] _strdate(string)
```

Where:

variable is an nstring variable to receive the answer.

string is an nstring variable.

Discussion

The `_strdate` function copies the current system date into an nstring variable. The date is formatted as mm/dd/yy. The return value can also be assigned to an nstring variable, or displayed on the command line.

Example 1

Command Input:

```
define nstring buffer
_strdate(buffer)
buffer
```

Result:

```
12/29/08
```

Example 2

Command Input:

```
define nstring buffer
define nstring strAnswer = _strdate(buffer)
strAnswer
```

Result:

```
12/29/08
```

_strlwr

Convert a string to lowercase.

Syntax

[variable=] `_strlwr(string)`

Where:

variable is an nstring variable to receive the return value.

string is an nstring variable.

Discussion

The `_strlwr` function converts any uppercase letters in a string to lowercase. All other characters are left unchanged. The return value can be assigned to an nstring variable, or displayed on the command line.

Example 1

Command Input:

```
define nstring strHello = "HELLO"  
_strlwr(strHello)
```

Result:

hello

Example 2

Command Input:

```
define nstring strHello = "HELLO"  
define nstring strAnswer = _strlwr(strHello)  
strAnswer
```

Result:

hello

strpos

Find a character in a string.

Syntax

```
[index = ] strpos(string, ch)
```

Where:

index	is an ord4 variable to receive the return value.
string	is an nstring variable, or string constant, to search.
ch	is the character to search for.

Discussion

The strpos function finds a character in a string. The return value is the index of the first instance of the character found. The return value can be assigned to an ord4 variable, or displayed on the command line.

Example 1

Command Input:

```
define nstring test = "123456"  
strpos(test, '3')
```

Result:

2

Example 2

Command Input:

```
define ord4 nIndex = strpos("123456", '4')  
nIndex
```

Result:

3

Strstr Command

The **strstr** command searches an ASCII string for the occurrence of a given sub-string.

Syntax

```
[ ret_val = ] strstr (string1, string2)
```

Where:

ret_val	Specifies the debug object of type int4 to which the function return value is assigned. If ret_val is not specified, the return value is displayed on the next line of the screen.
string1	Specifies an nstring variable, a quoted string constant, or an address of a string to search.
string2	Specifies an nstring variable, a quoted string constant, or an address of a sub-string to find within string1.

Discussion

A case sensitive search is performed on string1, looking for string2. If string2 is found within string1, a non-negative int4 value is returned indicating the index to the start of string2. A return value of -1 indicates that string2 was NOT found within string1.

Example

The following example illustrates the **strstr** function:

Command input:

```
base = dec;//set base to decimal
define nstring string1 = "AaBbCcDdEeFf"
define nstring string2 = "Dd"
define int4 ret_val = strstr(string1, string2)
ret_val
```

Result:

```
6T
```

Command input:

```
strstr(string1, "DD")
```

Result:

```
-1T
```

`_strtime`

Copy the current system time to an nstring variable.

Syntax

```
[variable = ] _strtime(string)
```

Where:

variable is an nstring variable to receive the answer.

string is an nstring variable.

Discussion

The `_strtime` function copies the current system time into an nstring variable. The time is formatted as hh:mm:ss. The return value can also be assigned to an nstring variable, or displayed on the command line.

Example 1

Command Input:

```
define nstring buffer  
_strtime(buffer)  
buffer
```

Result:

```
08:37:35
```

Example 2

Command Input:

```
define nstring buffer  
define nstring strAnswer = _strtime(buffer)  
strAnswer
```

Result:

```
08:37:36
```

Strtod Command

The **strtod** command converts a string into a real8 variable.

Syntax

[real8 =] strtod (string-expr)

Where:

string-expr Specifies an nstring variable or a quoted string constant.

Discussion

The function strtod expects the number to be converted to consist of:

1. An optional plus or minus sign
2. A sequence of decimal digits, possible containing a single decimal point
3. An optional exponent part, consisting of the letter e or E, an optional sign, and a sequence of decimal digits

The conversion stops at the end of the string or after encountering an illegal character. If no conversion can be performed, then zero is returned.

Examples

Command input:

```
strtod("1.2")
```

Result:

```
1.2
```

Command input:

```
define real8 answer  
answer=strtod ("23.345")  
answer
```

Result:

```
23.345
```

Command input:

```
define nstring myString = "4.56 inches"  
strtod (myString)
```

Result:

```
4.56
```

Strtol Command

The **strtol** command converts a string into an int4 variable.

Syntax

```
[int4=] strtol(string-expr, base)
```

Where:

string-expr	specifies an nstring variable or a quoted string constant
base	the number base to be used in the conversion (2-36)

Discussion

The **strtol** command converts a string into an int4 variable. The **strtol** function expects the number to be converted to consist of:

1. An optional plus or minus sign.
2. A sequence of digits whose legal values are indicated by the base specified (e.g., a base of 16 indicates 0-9, a-f and A-F are legal values.)
3. As a special case, if base is 16, then the string may begin with a 0x or 0X.

The conversion stops at the end of the string or after encountering an illegal character. If no conversion can be performed, then zero is returned.

Examples

Command input:

```
base=hex
strtol ("1000", 16t)
```

Result:

```
00001000H
```

Command input:

```
strtol ("1000", 10t)
```

Result:

```
000003E8H
```

Command input:

```
strtol ("1000", 8t)
```

Result:

```
00000200H
```

SourcePoint 7.7.1

Command input:
strtol ("1000", 2t)

Result:
00000008H

Strtoul Command

The **strtoul** command converts a string into an ord4 variable.

Syntax

```
[ord4=] strtoul(string-expr, base)
```

Where:

string-expr	specifies an nstring variable or a quoted string constant
base	the number base to be used in the conversion (2-36)

Discussion

The **strtoul** command converts a string into an ord4 variable. The **strtoul** function expects the number to be converted to consist of a sequence of digits whose legal values are indicated by the base specified (e.g., a base of 16 indicates 0-9, a-f and A-F are legal values.) As a special case, if base is 16, then the string may begin with a 0x or 0X.

The conversion stops at the end of the string or after encountering an illegal character. If no conversion can be performed, then zero is returned.

Examples

Command input:

```
base=10t
strtoul("123", 10)
```

Result:

```
123T
```

Command input:

```
base=16t
define ord4 answer
answer = strtoul("0x1000", 16t)
answer
```

Result:

```
00001000H
```

Command input:

```
base=10t
define nstring myString = "2048 cars"
strtoul (myString, 10)
```

Result:

```
2048T
```


`_strupr`

Convert a string to uppercase.

Syntax

```
[variable = ] _strupr(string)
```

Where:

variable	is an nstring variable to receive the return value.
string	is an nstring variable.

Discussion

The `_strupr` function converts any lowercase letters in a string to uppercase. All other characters are left unchanged. The return value can be assigned to an nstring variable, or displayed on the command line.

Example 1

Command input:

```
define nstring strHello = "hello"  
_strupr(strHello)
```

Result:

```
HELLO
```

Example 2

Command input:

```
define nstring strHello = "hello"  
define nstring strAnswer = _strupr(strHello)  
strAnswer
```

Result:

```
HELLO
```

Swbreak Command

The **swbreak** command displays or modifies software breakpoints.

Syntax

```
swbreak [=] vls-addr [...]
```

Where:

vls-addr	Specifies a memory location with a virtual, linear, physical, or symbolic address. The specified memory location must be in RAM and must be the beginning of an instruction.
----------	--

Discussion

Use the **swbreak** command to set or display code patch emulation breakpoints. You can specify a maximum of 64 software breakpoints. Restrict software breakpoints to code segments that remain present in memory. Use the **swremove** command to remove software breakpoints. A list of software breakpoints can be viewed in the **Breakpoints** window.

The emulator inserts software breakpoints into memory before entering emulation. When the emulator executes a breakpoint, the emulator removes all software breakpoints from memory (this feature causes the software breakpoints to appear to be invisible). When re-entering emulation using the **go** command, the emulator automatically single-steps the first instruction if a software breakpoint has been inserted there. The emulator then re-inserts all the software breakpoints into memory and continues emulation until the next specified breakpoint.

Do not use software breakpoints in a paged memory system. The emulator modifies the code to place the breakpoints. If the code is paged out of memory, the modifications remain in the code, corrupting it. Use the debug register breakpoint option of the **go** command to set execution breakpoints in paged memory systems.

Note: Do not set software breakpoints in a data area. The emulator may report errors on breaking from emulation.

Note: This command does not display if a software breakpoint is enabled or disabled. See the **swbreak** command for enable/disable information.

Example

The following example shows how to add a series of software breakpoints.

Command input:

```
sw-break=5400:0000
swbreak
```

Result:

```
Soft Break: 5400:00000000
```

Command input:

Softbreak=e,loc=5300:0000
swbreak

Result:

SoftBreak: 5400:00000000
Soft Break: 5300:0000

or

Soft Break: E,Execute,L-000054000P X=Once
Soft Break: E,Execute,L-000053000P X=Once

Switch Control Construct

This control construct causes execution to branch to one of several case statements.

Syntax

```
switch (expr)
{
case label-expr: [ commands ] [ ... ]

[ default: commands ]

}
```

Where:

expr	Specifies a number or an expression. The value of expr is compared to the value of the label in each case statement.
{ }	Braces are used to group all of the case statements and one optional default statement.
case	Indicates the beginning of one case statement.
label-expr:	Specifies a number or an expression whose value is compared to expr. The colon (:) is required punctuation.
commands	Any emulator commands, including break (which causes an immediate exit from the switch control construct). You cannot use the include command.
default:	Specifies the statement that is executed if none of the case statements label-expr: values match that of expr. The colon (:) is required punctuation.

Discussion

Use the **switch** control construct to transfer execution control to any commands following the case label-expr: statement whose label-expr: value matches the value of the switch expression. If no case label-expr: matches, no commands are executed unless there is a default statement. You can specify only one default statement. Once command execution begins at a case label-expr:, it continues through all remaining case commands unless the **break** command is encountered.

Note: You can put the case control constructs in any order, but the default statement, if used, must be last.

The **include** command is not executable inside the **switch** control construct.

Examples

1. The following example shows how the format of a switch control construct.

Command input:

```
switch (entry)
{
    case 1: base= 2t; break
    case 2: base= 10t; break
    case 3: base = 16t; break
    default: base
}
```

2. The following example shows the **switch** control construct used in a debug procedure (proc). When there is a match, another proc is called. When there is no match, the default statement is executed; assume proc1, proc2, proc3, and proc4 have been previously defined.

Command input:

```
define int4 response = 2
redefine proc menu ()
{
    switch (response)
    {
        case 1: proc1
        break
        case 2: proc2
        break
        case 3: proc3
        break
        case 4: proc4
        break
        default: printf("Not a valid choice \n")
        break
    }
}
```

Swremove Command

The **swremove** command removes software breakpoints.

Syntax

```
swremove {all | [=] vls-addr [...]}
```

Where:

vls-addr	specifies that the software Breakpoint at the specified virtual, linear, physical, or symbolic address is removed
all	removes all software emulation breakpoints

Discussion

Use the **swremove** command to remove software breakpoints. If you specify more than one vls-addr, use a comma as a separator. You can specify a maximum of 64 addresses.

Example

The following example shows how to remove the software breakpoint at offset 100h.

Command input:

```
swremove 100h
```


Tabs Control Variable

The **tabs** control variable displays or changes the tab spacing for **I/O** commands.

Syntax

`tabs [= expr]`

Where:

`expr` Specifies a value between 1 and 8, inclusive

Discussion

This control variable display or changes the tab spacing for **I/O** commands. The default value for tab spacing is 4. This variable can also be used in an expression.

Example

The following example shows how to save the current tab spacing, set a new spacing and then restore the old value.

Command input:

```
define int2 svtabs = tabs
tabs = 8
printf("%t\t%d\t%d", x, y, z)
tabs = svtabs
```

Tan Command

The **tan** command returns the tangent of a radian expression.

Syntax

[name =] tan (expr)

Where:

- name Specifies a debug object of type real8 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.
- (expr) Specifies a number or an expression of type real8 evaluated in radians. The parentheses are required

Note: Values returned by this command (a math function) are in real8 or 64-bit floating point precision. These values are displayed in the Command window rounded to 6 decimal digits. However, assignments and comparisons are performed on the full 64-bit value.

TargPower Control Variable

The **targpower** control variable displays whether the target is powered on or not.

Syntax

targpower

Discussion

Use **targpower** to determine if SourcePoint detects that the target is powered on. This is a read-only variable.

Example

Command input:

targpower

Result:

TRUE // the target is powered on

TargStatus Control Variable

The **targstatus** control variable displays the status of the target.

Syntax

targstatus

Discussion

Use **targstatus** to determine what state the target is in. This is a read-only variable.

The following is a list of possible status strings returned by **targstatus**.

- NoPower
- Waiting
- Stopped
- Running
- Stepping
- Flushing
- Halting
- Resetting
- Sleeping
- ShutdownPending
- Shutdown

Examples

Example 1

Command input:

targstatus

Result:

Stopped

Example 2

Command input:

go

Example 3

Command input:

targstatus

Result:

Running

Taskattach Command

Cause the debugger to attach to and control a task already running on the target operating system.

Syntax

```
[ result = ] taskattach(filename, pid)
```

Where:

result	specifies the debug object of type ord4 to which the function return value is assigned. If result is not specified, the return value is displayed on the next line of the screen. The return value is 1 if successful and 0 if not successful.
filename	is a string that specifies the path and filename of the symbol file on the host that is associated with the target program.
pid	specifies the program identifier (PID) of the program to attach to and debug. The program is already running on the target operating system.

Discussion

The taskattach command attaches the debugger to the program associated with the specified program identifier (PID) that is already running on the target operating system. If successful, the task will be stopped and ready for debugging. The PID for a program can be obtained by using the [taskgetpid](#) command.

Example

The following example demonstrates attaching the debugger to the target operating system program with a PID equal to 5.

Command input:

```
taskattach("c:\\prog\\hello", 5)
```

Result:

```
0001H
```

Related Topics

[taskend](#)
[taskgetpid](#)
[taskstart](#)

Taskbreak, Taskremove, Taskdisable, Taskenable Commands

Set, clear, display, enable and disable task breakpoints.

Syntax

taskbreak

```
taskbreak = [ name-spec, ] [ sts-spec, ] location-spec [, task-spec] [,
macro-spec]
```

taskremove [all]

```
taskremove = { name-spec | location-spec | task-spec | macro-spec } [
,... ]
```

```
taskenable = { name-spec | location-spec | task-spec | macro-spec } [
,... ]
```

taskdisable [all]

```
taskdisable = { name-spec | location-spec | task-spec | macro-spec } [
,... ]
```

Where:

name-spec	Breakpoint name
sts-spec	{ e[nabled] d[isabled] }
task-spec	p[rogram] = program name
location-spec	l[ocation] = vls-addr
macro-spec	f[file] = path of macro file to execute when break hits

Discussion

The taskbreak command sets and displays task breakpoints. Taskbreak with no arguments displays a list of the current task breaks.

The taskremove command removes any or all of the task breaks. Arguments to this command qualify which task breaks are to be removed. For instance, taskremove=l=1002, n=Break01 removes the task break with the name Break01 and address = 1002. Taskremove with no arguments removes all task breaks.

The taskenable command selectively enables task breaks. Arguments to this command qualify which task breaks are to be affected. For instance, taskenable=f=c:\OnBreak.mac enables only task breaks that will run the macro C:\OnBreak.mac on break.

The `taskdisable` command selectively disables task breaks. Arguments to this command qualify which task breaks are to be affected. For instance, `taskdisable=p=/home/hello` disables only task breaks for the task `/home/hello`. If no arguments are specified, all task breaks are disabled.

Task breaks can also be set, displayed, etc. from the Breakpoints window.

Note: For `taskremove`, `taskenable`, and `taskdisable` the location-spec will not match when a task break is inactive i.e. the break does not apply to the current task. This is because if a task is out of context, its address space is not valid.

Examples

Command inputs:	
<code>taskbreak</code>	<code>// display current task breaks</code>
<code>taskbreak = location=1234, f=C:\OnBreak.mac</code>	<code>// set a task break at address 1234 for the current task that will run the macro OnBreak.mac on break</code>
<code>taskbreak = l=1000p, p=/bin/ls</code>	<code>// set a task break at address 1000p for the task /bin/ls</code>
<code>taskremove</code>	<code>// remove all task breaks</code>
<code>taskremove = p=/home/hello</code>	<code>// remove all task breaks associated with task /home/hello</code>
<code>taskdisable</code>	<code>// disable all task breaks</code>
<code>taskdisable = n=firstBreak</code>	<code>// disable task break with name firstBreak</code>
<code>taskenable = p=/home/hello</code>	<code>// enable all task breaks with task /home/hello</code>
<code>taskenable = f=C:\OnBreak.mac</code>	<code>// enable all task breaks that will run the macro file C:\OnBreak.mac on break</code>

Taskend Command

Stop debugging a task on the target operating system.

Syntax

```
taskend ( vp )
```

Where:

vp is an integer that specifies the viewpoint of the task being debugged.

Discussion

The taskend command causes the debugger to stop debugging a task that is running on the target operating system. If the task was started using taskstart then the debugger halts the task and closes the context of that debugging session. If the task was attached to using [taskattach](#), then the debugger allows the task to continue running but releases control and disconnects from the task.

Note: When debugging a task on a target operating system, the value of the viewpoint for that task is always 40H or higher. This differentiates task viewpoints from processor viewpoints, which are always 0H to 3FH.

Example

The following example successfully stops debugging the target operating system program whose viewpoint is 40H.

Command input:

```
taskend ( 40H )
```

Result:

```
0001H
```

Related topics:

[taskattach](#)
[taskgetpid](#)
[taskstart](#)

Taskgetpid Command

Retrieves the program identifier (PID) of a task running on the target operating system.

Syntax

```
taskgetpid(vp)
```

Where:

vp is an ord4 that specifies the viewpoint of the task being debugged.

Discussion

The taskgetpid command returns the program identifier (PID) of the task corresponding to the specified viewpoint. The task being debugged is running on the target operating system and was launched using [taskstart](#) or attached to using [taskattach](#).

Example

The following example gets the PID from the target operating system program with viewpoint 40H.

Command input:

```
taskgetpid(40H)
```

Result:

```
0007H
```

Related Topics:

[taskattach](#)

[taskend](#)

[taskstart](#)

Taskstart Command

Cause the target operating system to start a program under the control of the debugger. If successful, the new task will be stopped at its program entry point.

Syntax

```
[ vp = ] taskstart(filename, targetpath, arguments)
```

Where:

vp	specifies the debug object of type ord4 to which the function return value is assigned. If vp is not specified, the return value is displayed on the next line of the screen. The return value is the viewpoint of the task to debug.
filename	is a string that specifies the path and filename of the program on the host to start.
targetpath	is a string that specifies the path and filename of the program on the target operating system to start.
arguments	is a string that specifies the command line arguments for the program. An empty string "" specifies no arguments. This parameter should be enclosed by double quotes, especially if there are spaces or multiple arguments. If there are arguments that are strings, then their quotes will have to be escaped (e.g., "1 2 \"three\" 4").

Discussion

The taskstart command uses the symbols from the program file on the host (filename) and launches the corresponding program (targetpath) on the target operating system under debugger control with the supplied command line parameters (arguments). The program files on both the host and target operating system must exist for this command to succeed. The third parameter (arguments) should be an empty string if no arguments are needed.

Use the [taskend](#) command to stop debugging the task.

Examples

The following examples show various forms of starting a program on the target operating system.

Command input:

```
taskstart("hello", "/home/hello", "")
taskstart("c:\\prog\\hello", "/home/hello", "25 /x /b")
taskstart("hello", "/home/hello", "13 \"test string\" /g")
```

Related Topics

[taskattach](#)
[taskend](#)
[taskgetpid](#)

Tck Control Variable

The **tck** control variable displays or changes the emulator's current JTAG clock rate.

Syntax

tck [= clockrate]

Where:

clockrate a string specifying the new value for the JTAG clock rate

Discussion

The JTAG clock rate controls the speed of the interface between the emulator and the target. A higher frequency provides better response from the target, but not all targets support all frequencies as this is hardware dependent.

The **tck** control variable provides command support for the JTAG clock rate setting found on the **Emulator Configuration** dialog under [Options|Emulator Configuration|JTAG Clock](#).

The clockrate argument must be delimited by double-quotes, is case-sensitive, and must be identical to one of the JTAG clock rate strings found on the aforementioned dialog.

For more information, go to the topic, "[Options Menu - Emulator Configuration Menu Item](#)," part of "SourcePoint Overview," found under SourcePoint Environment.

Example

tck // display the current setting

tck = "2.0 MHz" // set JTAG clock rate to 2 MHz

Time Command

The **time** command puts the date and time function in internal format (seconds since Greenwich Mean Time, January 1, 1970). The time function returns the value as a ord4 data type.

Syntax

```
[ name = ] time ( )
```

Where:

name Specifies a debug object of type ord4 to which the function return value is assigned. If name is not specified, the return value is displayed on the next line of the screen.

Triggerposition Command

The **triggerposition** command is used in setting or displaying the bus analyzer trigger position.

Syntax

```
trig[gerposition] [= {trigpos-spec | position-value}]
```

Where trigpos-spec is one of the following:

begin	beginning of the trace buffer
middle	middle of the trace buffer
end	end of the trace buffer

and where:

position value specifies a percentage of the trace buffer

Discussion

The **triggerposition** command is used to select or display at what point in the trace buffer the bus analyzer trigger will occur. The trigger position can also be set in the breakpoint window.

Example

Command input:

<i>trig = middle</i>	<i>//sets the trigger position to the middle of the trace buffer</i>
<i>triggerposition = 25</i>	<i>//sets the trigger position at 25 percent of the trace buffer</i>

Command input:

<i>triggerposition</i>	<i>//query the current trigger position</i>
------------------------	---

Result:

25T

Unload Command

The **unload** command unloads program files from SourcePoint, removing all source and symbol information.

Syntax

```
unload [all | filename]
```

Where:

all Specifies that all programs are to be unloaded.

filename A string constant or nstring variable naming the file to open. Filenames with spaces must be enclosed in quotation marks

UnloadProject Command

The **UnloadProject** command unloads the current SourcePoint project file.

Syntax

UnloadProject ()

Discussion

The **UnloadProject** command unloads the current project file. All views, except for the **Log** and **Command** windows, are closed.

This command is rarely used since SourcePoint needs a project file loaded to connect to a target. To load a new project file, use the [LoadProject](#) command.

Example

Command input:

UnloadProject()

Use Control Variable

The **use** control variable sets the default address size used by the **asm** command.

Syntax

```
use [{expr | use16 | use32}]
```

Where:

expr Specifies a number or an expression that must evaluate to 16t or 32t. The default is 16t

use16 Indicates 16-bit addressing.

use32 Indicates 32-bit addressing.

Discussion

Use the **use** control variable to set the default address size used by the **asm** command. Entering the control variable without selecting an option displays the current setting. When set to **use16** (the default), the debug tool interprets assembler addresses as 16-bit. When set to **use32**, the debug tool interprets assembler addresses as 32-bit.

The **use** command is identical in function to the **asmmode** command.

Ver Command

The **ver** command returns the current version of SourcePoint and boot ROM code/flash code in the target emulator.

Syntax

```
ver()
```

Discussion

Use the **ver** command to display the current software and firmware version numbers.

Example

The following example shows the output from SourcePoint command window:

Command input:

```
ver()
```

Result:

"SourcePoint v4.4.0.609 Build 609 Official; Run Control: Boot: v1.78, Flash: v2.13"

Verify Control Variable

The **verify** control variable specifies read-back checks on writes to target memory.

Syntax

verify [= true | false]

Where:

true specifies that memory writes to target memory are verified

false specifies that target memory is not verified

Discussion

Use the **verify** control variable to specify read-back checks on commands that write to target memory. If **verify** is false, read back checks do not occur. If **verify** is true, read-back checks do occur. The default setting is false.

Setting **verify** to true detects errors when writing to memory; however, read-back checks increase the time needed to do memory write operations.

Examples

1. To display the current value of **verify**:

```
>verify
```

2. To change **verify** to true:

```
>verify=true
```

Viewpoint Control Variable

The **viewpoint** control variable displays or changes the current viewpoint.

Syntax

viewpoint [=Pn]

Where:

=Pn specifies the nth processor in scan chain

Discussion

Use the **viewpoint** control variable to define which processor in a multiprocessor system is the current processor. Entering the command without an option displays the current setting. The default setting for **viewpoint** is P0. The current viewpoint is also displayed and changed in the **Viewpoint** status window.

Linux Task Debugging

Viewpoint identifiers that represent processors are in the range of 00h-3fh. Task identifiers are 40h and greater. Users of the task debugging feature find that the **Viewpoint** window is the interface of choice to view and alter the viewpoint in SourcePoint. However, the viewpoint control variable may be useful in some macro program scenarios where it is necessary, say, to switch from task mode to halt mode, set a kernel breakpoint, and then continue to work in the original task.

vpalias

Display or change a viewpoint alias

Syntax

```
[[px]] vpalias [=expr]
```

Where:

[px] Viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.

expr is a string or an expression that evaluates to a valid string.

Discussion

Use the vpalias command to define an alias for a processor. Specifying an empty string clears an alias.

Letters must be used in the alias string and numbers can also be used. However, the first character of the string must be a letter. Alias names are limited to 6 characters.

Aliases can also be viewed and changed in Options | Target Configuration | Devices.

Example 1

To name the current viewpoint "bob":

Command input:

```
vpalias = "bob"
```

Result:

```
bob>
```

Example 2

To name the [p3] processor of the boundary scan chain "jane":

Command input:

```
[p3]vpalias = "jane"  
view = jane
```

Result:

jane>

Example 3

To display the alias for the current viewpoint processor:

Command input:

vpalias

Result:

jane

Example 4

To clear the alias for P3 (jane):

Command input:

*[p3]vpalias = ""
view = p3*

Result:

P3>

Wait Command

The **wait** command suspends command execution until a breakpoint is encountered.

Syntax

`wait`

Discussion

Use the **wait** command to prevent the emulator from accepting commands until an emulation breakpoint is reached. This command is especially useful in debug procedures (procs) and macro files that have **go** commands. The **wait** command prevents subsequent commands in the proc or include file from executing until the processor is in a stop condition.

The **wait** command has no effect when emulation is stopped.

Example

The following example shows how to define a procedure named `go_reg` that executes a **go** command, then waits until the **go** command breaks from emulation before executing the rest of the procedure.

Command input:

```
define proc go_reg
{
  go til 08:42BH
  wait
  eax ; edx
}
go_reg
```

Result:

```
12345678H
00000000H
```

While Command

The **while** control construct groups and executes commands while a condition is true.

Syntax

```
while (bool-cond) {commands}
```

Where:

bool-cond	Specifies a number or an expression that is evaluated and tested. The loop repeats until expr evaluates to false (zero). The parentheses are required.
commands	Any emulator commands. When you enter more than one command, you must enclose them in braces ({ }).

Discussion

Use the **while** control construct to execute the specified commands 0 (zero) or more times, as long as bool-cond evaluates to true (non-zero).

The **include** command is not executable inside a **while** control.

Examples

1. The following example demonstrates a **while** control construct. While the index is greater than zero, decrement the index and add 5 to the sum on every iteration of the loop.

Command input:

```
define ord1 i = 5
define ord1 sum = 0
while (i > 0)
{
  i -= 1 ; sum += 5
  printf ("i = %d sum = %d \n", i, sum)
}
```

Result:

```
i = 4 sum = 5
i = 3 sum = 10
i = 2 sum = 15
i = 1 sum = 20
i = 0 sum = 25
```

2. In the following example, while the index is greater than zero, decrement the index and subtract 10 from the variable "a". When a is less than zero, exit the loop.

Command input:

```
base = 10t
```



```

define ord1 i = 5
define int2 a = 30
while (i > 0)
{
    i--
    a -= 10
    if (a < 0) break
}
i

```

Result:

```

5T
4T
3T
2T

```

Command input:

```
a
```

Result:

```
2T
```

3. The following example shows how to single-step through code until the AX register is non-zero.

Command input:

```

while (ax > 0)
istep

```

Wport Command

The **wport** command displays or changes the contents of a 16-bit I/O port.

Syntax

```
[[px]] wport (io-addr) [= expr]
```

Where:

[px]	Viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set to processor x of the boundary scan chain, where x is the number (0, 1, 2, or 3).
io-addr	Specifies a 16-bit address in the processor I/O space. The available io-addr range is 0 to 0ffffh. The use of parentheses is optional.
expr	Specifies a 16-bit number or an expression. Using this option writes the data to the specified I/O Port.

Discussion

Use the **wport** command to read from and write to the specified I/O port with the specified 16-bit data.

Examples

1. The following example shows how to assign a 16-bit value to a port.

Command input:

```
wport 88h = 4321h
```

2. The following example show how to assign one port value to another port.

Command input:

```
wport 90h = wport 88h
```

3. The following example shows how to create a debug variable named portvar and assign a port value to it.

Command input:

```
define ord4 portvar
portvar = wport 90
portvar
```

Result:

```
00004321H
```

WriteSetting Command

The **writeSetting** command is used to modify settings within SourcePoint.

Syntax

```
ord4 writeSetting(type, name)
```

Where

type	an nstring or string constant specifying the type of setting
name	an nstring or string constant specifying the setting name
value	an ord4 specifying the value to assign to name

Discussion

This **writeSetting** command is used to modify settings within SourcePoint. Usually, these settings are changed via the UI (e.g., the **Emulator Configuration** dialog box). There are times, however, when it is convenient to be able to change these settings within a macro file.

The type argument specifies the type of setting to change. Currently, the only type supported is “em” for emulator configuration settings.

The name argument specifies the name of the setting to change. The name is not what is displayed in the UI, but rather the name used in the project file. Names can be obtained by looking in the project file in the Emulator Configuration section.

The value argument can be obtained by changing the emulator configuration setting in question and looking in the project file. For checkbox settings, the value is TRUE or FALSE. For radio buttons, the value usually (but not always) is the zero-based index of the button selected. For drop down lists, the value usually (but not always) is the zero-based index of the entry selected. The safest way to determine value is to look in the project file.

Note: Values in the project file are usually decimal. Regardless, values in the command language are specified using the current command language input radix (specified with the base control variable). If the input radix is hex, and you want to specify a value of 200 decimal, then you need to use 200T as 200 is interpreted as 200H = 512 decimal.

Example

The following example returns the Adaptive TCK setting. The possible values are 0, 1 and 2 corresponding to which radio button is selected in the UI.

Command input:

```
writesetting("em", "AdaptiveTck") //0 = "Don't use adaptive TCK"
```

Yield Command

The **yield** command is used inside macro files to allow another process to run.

Syntax

yield

Discussion

The **yield** command pauses execution of a macro file to allow another process to run, SourcePoint windows to be updated, and Ctrl-Break processing to occur. A macro file normally yields at every loop iteration. For more information, see the **yieldflag** command.

Yieldflag Control Variable

The state of `yieldflag` affects macro file execution. The command window displays output immediately as it happens. By using the **yieldflag** control variable, you can watch the execution of a macro line by line. Errors and notifications appear as they happen.

Syntax

`yieldflag [= true | false]`

Where:

`true` specifies that the emulator pauses macro file execution at every loop iteration
`false` specifies that the emulator yields control only when an explicit **yield** command is encountered

Discussion

When **yieldflag** is true, the emulator pauses macro file execution at every loop iteration. This allows other processes to run, the SourcePoint windows to update, and Ctrl-Break processing to occur. When **yieldflag** is false, the emulator yields control only when an explicit **yield** command is encountered. The default state of **yieldflag** is true.

For more information, see the **yield** command.

Examples

1. The following example adds output to the SourcePoint windows one line at a time.

Command input:

yieldflag

Result:

True

Command input:

*define intl i
 for (i=0; i<=1000; i++) //add output one line at a time
 i*

Result:

*0
 1
 .
 .
 1000*

2. In this example the output is added to the SourcePoint windows when the loop terminates.

Command input:

```
yieldflag = false  
for (i=0; i<=1000; i++) //add output when loop finishes  
i
```

Result:

```
0  
1  
.  
.  
1000
```

Putfile Command

The putfile command copies a file from the Sourcepoint host to the target. The command is transferred in binary mode with no ascii translation.

Syntax

```
[ result = ] putfile(host_path, target_path, permissions)
```

Where:

result	Boolean return value is true if the copy succeeded.
host_path	Specifies the path on the debug host to the file to be copied.
target_path	Specifies the path on the target where the file will be created.
permissions	Specifies the UNIX file permissions for the target file. This parameter determines the Read/Write/Execute permissions for Owner/Group/Others. It is normally specified as 3 octal digits. For details, refer to the Linux manual page for the chmod command.

Discussion

The putfile command requires Sourcepoint Linux-aware debugging support on the target. Use putfile to place a file on the target filesystem. The Boolean result is true if the file was transferred successfully. In the event of failure, a descriptive error message is displayed in the Command window.

Example 1

Command input:

```
putfile("C:\\myhome\\myprog", "/tmp/myprog", 755o)
```

Example 2

Command input:

```
putfile("~/targets/csb337/root_fs/lib/modules/testmod.ko",  
"/lib/modules/testmod.ko", 644o)
```

Example 3

Command input:

```
putfile("U:\\targets\\csb337\\root_fs\\lib\\modules\\testmod.ko",  
"/lib/modules/testmod.ko", 644o)
```

Osaware Control Variable

The **osaware** control variable displays or changes the target operating system selected for debugging.

Syntax

```
osaware [ = string-expr ]
```

Where:

string-expr specifies a target operating system for debugging

Discussion

Use the **osaware** control variable to enable or disable debugging of supported target operating systems such as Linux. Setting the control variable to "[None]" disables target operating system debugging. The default setting for **osaware** is disabled. Entering the control variable without an option displays the current setting.

Currently, only Linux is supported. Future versions of SourcePoint may support other target operating systems.

Examples

Command inputs:

```
osaware                // display the current setting
[None]

osaware = "Linux"      // enable Linux debugging on the target

osaware                // display the current setting
Linux

osaware = "[None]"     // disable target operating system debugging

osaware                // display the current setting
[None]
```


Index

#

#define command356

#undef command357

—

_strdate615

_strlwr616

_strtime619

_strupr625

A

Aadump command360

Abort command361

Abs command362

Acos command363

Arium - contacting15

Array data type313

Asin command364

Asm command365

Asmmode control variable376

Atan command377

Atan2 command378

B

Base control variable379

Bell (beep) command381

Binary values - changing244

bits 382

Bookmarks 308

Break command 385

Break on 123, 127

Sequence and data qualification 127

Breakall control variable 386

Breakpoints 123

Adding/editing 127, 133

Bus analyzer 123, 142

Emulator 123, 143

Enabling/disabling 138

Icon definitions 126

Menu 127

Opening 123

Preferences 137

Processor 129, 144

Removing 139

Resources window 127

Setting 140

Software 129, 145

Types 123, 129

Bus analyzer (breakpoint type) 123

Bus analyzer sequence 123

Busbreak command 387

Busdisable command	387	Confidence tests details	174
Busenable command	387	Continue command	400
Busremove command	387	Control constructs	318
Byte command	337	Control variables	319
C		Copy	40
Cachememory control variable	390	Help topics	6
Cause command	392	Cos command	401
Character functions	393	Cpubreak command	402
Character strings	315	Cpudisable command	402
clock	399	Cpuenable command	402
Code disassembly	156	cpuid_eax	404
Code window	147	cpuid_ebx	406
Icon definitions	149	cpuid_ecx	408
Menu	150	cpuid_edx	410
Opening	147, 155, 181, 254, 256, 257	Cpuremove command	402
Preferences	154	cscfgandlocal_cscfg	412
Saving contents	158	csr	358, 423
Saving settings	157	Ctime command	425
Command window	159	Ctrl-break	162
Control register bit maps and names	316	Cut	57
Menu	161	cwd	426
Opening	159	D	
Syntax and conventions	160	Data qualification	123
Confidence tests	97, 171	Data types	321, 323
Confidence test failures and symptoms	177	Dataqualmode command	427

Dataqualstart command	427	Do_while command	444
Dbgbreak command	429	Docking windows	35
Dbgdisable command	429	Dos command	443
Dbgenable command	429	Dport command	446
Dbgremove command	429	DREPLY	123
Debug pointer types	328	Dword command	337
Debug procedures	324	E	
Debug registers	123	ECM-50	28
Debug variables	329	ECM-700	21
defaultpath	431	ECM-HDT	18
Deferred reply	123	ECM-XDP	24
Define command	433	Edit	57
DefineMacro command	436	Edit menu	57
Descriptor tables	179	Editor control variable	448
Cache	283	EFI address finder	115
Menu	181	EFI Framework debug	285
Opening	179	Eflags command	449
Replacing	183	Emubreak command	451
devicelist	438	Emudisable command	451
Devices window	185	Emuenable command	451
Accessing in Command window	196	Emulator - configuring	85
Creating - an example	198	Emulator - flash update	53
Menu	194	Emulator - resetting	96
DHCP	28	Emulator connection - adding	100
Displayflag control variable	442	Emulator connection - verifying	121

Emulator connections	94	Flush command	471
Emulator data types and type classes.....	355	Fopen command.....	472
Emulator operators	323, 343	For command.....	474
Emuremove command	451	fprintf	476
error	454	Fputc command	477
Eval command	457	Fputs command	478
Execute command	161	Fread command.....	479
Execution point command	458	fseek	480
Exit command.....	55, 459	ftell	481
Exp command.....	460	Function keys.....	35
Expressions	330	Functions	333
F		Fwrite command	482
Fc command	461	G	
Fclose command	462	GDT	179
Feof command.....	463	Getc command.....	483
Fgetc command	464	getfile	484
Fgets command	465	getNearestProgramSymbol	486
File - closing.....	462	GetProgramSymbolAddress command ..	488
File menu	40	Gets command.....	490
first_jtag_device.....	466	go	62, 491
Flags command	467	H	
Flash - updating emulator.....	53	Halt command.....	493
Flash - updating target.....	75	Help command.....	494
Flash code - updating	33	Hexadecimal values - changing.....	245
flist	469	homepath	495

Host system requirements.....17

Hot plugging.....110

Hotplug command496

I

IACK123, 129

Icon groups - editing109

Icons250

Icons - arranging.....98

Icons - displaying text108

idcode455, 497

IDT179

If command499

include501

Index command99

Index into609

Installation - SourcePoint.....28

Interrupt acknowledge (breakpoint type) 123, 129

Interrupts.....140

IsDebugSymbol command503

isem64t504

IsProgramSymbol command505

isrunning506

issmm508

itpcompatible.....510

J

JtagChain command 511

JtagDeviceAdd command..... 513

JtagDeviceClear command..... 514

JtagDevices command 515

JtagScan command 516

K

Keys command 517

L

Last command 519

last_jtag_device 521

Layout 42

LDT 179

LDTR..... 179

Left command 522

libcall 523

License command..... 526

Licensing..... 31

Linear command 527

Linux - Set Breakpoints..... 224

Linux OS-Aware Debugging 215

Linux Task Debugging 270

List command..... 528

Load command 530

Load target configuration files..... 83

LoadProject command.....	531	Memory map	75
LoadTarget command	532	Memory Map Tab.....	75
Log window.....	201	Memory window	207
Icon definitions	203	Changing values	214
Menu	204	Display fields	207
Log10 command.....	535	Menu	209
Loge command.....	536	Opening.....	212
M		Preferences	211
Machine level step into	62	Size and radix.....	209
Machine level step over.....	62	Target memory - copying	337
Macro commands	40	Target memory - filling	337
MacroPath control variable	537	View memory at address.....	213
Macros	165	messagebox.....	538
Configuring.....	105, 107	Mid command	540
Editing from a Command window	164	Modifying.....	111
Loading	165	Msgclose command.....	541
Recent macros command	55	Msgdata command	542
Macros and procs	335	Msgdelete command	544
Differences	335	Msgdr command	545
Executing	335	Msggir command	547
Limitations	336	Msgopen command	549
Memory access.....	337	Msgreturndatasize command	550
Copy and fill	337	Msgscan command.....	551
Memory casting	292	Msr command	552
Memory command.....	59	Multi-clustering.....	296

N

New project wizard41

 Using117

Nolist command528

Nolog command.....533

num_devices.....553

num_jtag_devices.....554

num_processors555

O

Operand character delimiters342

Operands323, 330, 342

Operating System Tab.....79

Operators.....323, 343

Options menu item63

ord1.....337

ord2.....337

ord4.....337

ord8.....337

Osaware Control Variable662

P

Page translation window.....227

 Opening.....227

Paste.....57

pause556

PCI devices window229

Menu 233

Opening PCI registers view..... 234

PCI registers..... 233

Refreshing 235

PE format support..... 293

Physical command..... 557

Port command..... 558

Pow command 560

Preferences..... 64

Print..... 51

 Print preview..... 51

 Print setup 51

Print command..... 561

Print cycles command..... 562

printf 563

Proc command..... 566

Procedures, debug..... 324

Processor determination..... 310

Processor menu..... 62

ProcessorControl control variable..... 567

ProcessorFamily function 569

ProcessorMode command..... 571

Processors control variable 570

ProcessorType function 572

Procs - deleting..... 163

Procs - displaying	163	Modify.....	237
Procs - loading.....	165	Opening.....	237
Program commands	43	Removing	241
Program Flash Tab	77	Reload command.....	580
Project commands.....	41	ReloadProject command	581
projectpath	573	Remove command.....	582
Putchar command	574	Reset command.....	62, 584
putfile	661	Reset link	63
Puts command.....	575	Reset NET	63
Q		restart.....	587
Quick watch	275	Retrieve commands.....	161
Qword command	337	REXX	166
R		Right command.....	586
Rand command	576	S	
ReadSetting Command	577	Safemode control variable	588
Real numbers command	345	Save Trace.....	268
reg.....	578	Sequence.....	127, 589
Register lists - printing	237	SequenceCount command	589
Register window - menu	241	SequenceType command.....	589
Registers.....	237	Shell Command	591
Copying to user page.....	241	Show command	592
Customizing window	237	Sin command	594
Editing	241	sleep	595
Expanding	241	SMM.....	123
Keyword table	300	Softbreak command.....	596

Softdisable command	596	Stop Command.....	603
Softenable command.....	596	Strcat command.....	604
Softremove command	596	strchr	605
Source path command	40	Strcmp command.....	606
SourcePoint	35	Strcpy command.....	608
Installation	28	Strlen command.....	610
Menu	98	Strncat command.....	611
New features.....	13	Strncmp command.....	612
Refreshing windows.....	113	Strncpy command.....	614
Toolbar	38	strpos	617
Customizing.....	59	Strstr command.....	618
SourcePoint online help.....	3	Strtod command.....	620
Changing font color	9	Strtol command.....	621
Changing window background color	10	Strtoul command.....	623
Changing window size	5	Sub-expressions	347
Copying a topic	6	Swbreak command	626
Finding a topic.....	4	Switch control construct.....	628
Help command.....	494	Swremove command	630
Hiding/showing contents	11	Symbolic references	349
Menu	99	Symbolic text format	306
Printing a topic	7	Symbols window	247
sprintf	598	Adding/expanding registers in a Watch view.....	280
Sqrt command	599	Changing values	258
Srand command	600	Classes view	253
Step command.....	601	Globals view	254

Locals view.....	256	Opening.....	259
Menus	251	Printing	267, 562
Stack view	257	Trace command	259
T		Trace Configuration	263
Tabs control variable	631	Trace Display Settings	265
Tan command.....	632	Triggerposition command	644
Target configuration.....	75	Type conversions.....	354
Loading	83	U	
Saving	84	Unload command.....	645
Target Devices Tab	81	UnloadProject command	646
TargPower control variable.....	633	Use control variable	647
TargStatus control variable.....	634	Using Help	3
taskattach	635	V	
taskbreak	636	Ver command.....	648
taskdisable.....	636	Verify control variable	649
taskenable	636	View menu commands	59
taskend	638	Viewpoint control variable.....	650
taskgetpid	639	Viewpoint Window.....	269
taskremove	636	Disabling processors.....	273
taskstart	640	Menu	271
Tck control variable	642	vpalias.....	651
Technical support	16	W	
Textsym	306	Wait command.....	169, 653
Time command.....	643	Watch window.....	275
Trace.....	259	Adding symbols.....	282

Adding/expanding registers in.....	280	WriteSetting Command.....	657
Menu	278	Y	
While command.....	654	Yield command	658
Word command	337	Yieldflag control variable.....	659
Wport Command.....	656		