

SourcePoint 7.12

Table of Contents

Contacting ASSET InterTech	1
Introduction to SourcePoint.....	3
What's New in SourcePoint 7.12	3
What's New in SourcePoint 7.11	4
What's New in SourcePoint 7.10.4	6
What's New in Sourcepoint 7.10.3	7
What's New in SourcePoint 7.10.2	8
What's New in SourcePoint 7.10.1	9
What's New in SourcePoint 7.10	10
What's New in SourcePoint 7.9	11
What's New in SourcePoint 7.8	12
SourcePoint Environment	15
SourcePoint Overview	15
SourcePoint Parent Window Introduction	15
SourcePoint Icon Toolbar	18
File Menu	20
File Menu - Project Menu Item.....	21
File Menu - Layout Menu Item	22
File Menu - Program Menu Item	23
File Menu - Macro Menu Item.....	28
File Menu - Print Menu Items.....	32
File Menu - Update Emulator Flash Menu Item	34
File Menu - Program Target Devices Menu Item.....	35
File Menu - Other Menu Items	36

Edit Menu	38
View Menu	41
Processor Menu	44
Options Menu	45
Options Menu - Preferences Menu Item	46
Options Menu - Target Configuration Menu Item	57
Options Menu - Load Target Configuration File Menu Item	62
Options Menu - Save Target Configuration File Menu Item	63
Options Menu - Emulator Configuration Menu Item	64
Options Menu - Emulator Connection Menu Item	76
Options Menu - Emulator Reset Menu Item	78
Options Menu - Confidence Tests Menu Item	79
Window Menu	80
Help Menu	81
How To -- SourcePoint Environment	83
How to Add Emulator Connections	83
How to Configure Custom Macro Icons	91
How to Configure Autoloading Macros	92
How to Display Text on the Icon Toolbar	93
How to Edit Icon Groups to Customize Your Toolbars	94
How to Modify a Defined Memory Region	95
How to Refresh SourcePoint Windows	97
How to Save a Program	98
How To Start SourcePoint With Command Line Arguments	99
How to Use the New Project Wizard	100

How to Verify Emulator Network Connections.....	103
Breakpoints Window	105
Breakpoints Window Introduction	105
Add/Edit Dialog	108
Breakpoint Types and Resources.....	110
How To - Breakpoints	112
Set Breakpoints From Other SourcePoint Windows.....	112
Code Window	115
Code Window Introduction.....	115
Code Window Icon Definitions.....	118
Code Window Menu.....	119
Code Window Preferences	123
How To - Code Window.....	124
How to Open a Code Window	124
How to Disassemble Code at a Specific Location	125
How to Save Code Window Settings	126
How to Save Code Window Contents.....	127
Command Window	129
Command Window Introduction.....	129
Confidence Tests Window	133
Confidence Tests Window Overview.....	133
Confidence Tests Window Introduction	133
Confidence Tests Tabs.....	136
Table of Confidence Test Failures and Symptoms.....	139
Descriptors Tables Window	141

Descriptors Window Introduction	141
Descriptors Window Menu	144
How To - Descriptors	146
How to Replace a Descriptor Entry	146
Devices Window	147
Devices Window Introduction	147
Devices Window Menu	157
Accessing Devices Window Cells in the Command Window	159
How To - Devices Window	161
How to Create a Simple Devices Window	161
Log Window	163
Log Window Introduction	163
Log Window Icon Definitions	165
Log Window Menu	166
Memory Window	169
Memory Window Introduction	169
Memory Window Menu	171
Memory Window Preferences.....	173
How To - Memory Window	174
How to Open a Memory Window	174
How to View Memory at an Address.....	175
How To Change Memory Values.....	176
Page Translation Window	177
Page Translation Window Introduction	177
PCI Devices Window.....	179

PCI Devices Window Introduction	179
PCI Devices Window Menu	183
How To - PCI Devices Window	184
How to Open the PCI Registers View From the PCI Devices Window	184
How to Refresh a PCI Devices Dialog Box.....	185
Registers Window	187
Registers Window Introduction	187
How To - Registers.....	191
Customize the Registers Window.....	191
Print a Register List.....	192
Symbols Windows.....	193
Symbols Window Introduction	193
Symbols Window Icon Definitions.....	196
Symbols Window Menus.....	197
Classes Tab	199
Globals Tab.....	200
Locals Tab	202
Stack Tab.....	203
How To - Symbols Window	204
How to Change Values in the Symbols Window	204
Trace Window	205
Trace Window Introduction	205
Trace Configuration	211
Trace Display Settings	227
How To - Trace Window	229

How to Print Trace	229
How to Save Trace	230
Advanced.....	231
Timestamp	231
Trace Hub Metadata	233
Viewpoint Window.....	239
Viewpoint Window Introduction.....	239
Viewpoint Window Menu	240
Watch Window	241
Watch Window Introduction	241
Watch Window Menu	244
How To - Watch Window	246
How to Add and Expand Registers in a Watch View	246
How to Add Symbols to a Watch or Quick Watch View.....	248
Technical Notes	249
Descriptor Cache: Revealing Hidden Registers	249
UEFI Framework Debugging	251
Overview	251
UEFI Macros	251
PEI Debugging.....	251
DXE Debugging	253
HOBs.....	254
System Configuration Table	255
Notes.....	255
Memory Casting	257

Defining Debug Variables of a Symbol Type as Defined in a Loaded Program.....	257
Casting Blocks of Target Memory as a Symbol Type as Defined in a Loaded Program	257
Microsoft® PE Format Support in SourcePoint.....	258
Overview	258
FAQs	259
Known restrictions of PE/PDB support in SourcePoint.....	260
Multi-Clustering.....	261
Hardware Setup	261
Software Setup	262
Running in Multi-Cluster Mode	262
Timing	263
Python/CScripts Introduction, Installation and Usage	264
Introduction	264
Requirements.....	264
Installation	264
Usage in SourcePoint	264
Invoking CScripts under SourcePoint	265
OpenIPC Integration, Installation and Usage	266
Introduction	266
Installation	266
Usage.....	266
SourcePoint Configuration	267
Sample Code	267
Registers Keyword Table	268
SourcePoint Licensing	271

Perpetual Model.....	271
Subscription Model	271
Mobile Licensing	272
Installing the SourcePoint Vendor Daemon.....	272
Current License File Information.....	272
Stepping.....	273
Strategies for Source Level Stepping	273
Symbolic Text Format (Textsym)	276
File Format.....	276
Example	277
Target Configuration.....	278
Overview	278
Simple Targets.....	278
Complex Targets.....	278
Configuration Command Overview.....	278
Advanced Topics	279
Using Bookmarks	282
Adding/Removing Bookmarks	282
Navigating Bookmarks.....	282
Clearing Bookmarks.....	282
Bookmark Indications.....	282
Which Processor Is Which.....	283
Introduction	283
What Does "Last on the Chain, First on the Chain" Mean?.....	283
How Is This Related to the PROCESSORCONTROL Variable in SourcePoint?	284

What Does It Mean to Control More Than One Processor?	284
Getting Started with DbC	285
Overview	285
Preparing the Firmware image	285
To enable DbC for a specified USB2.0 port:	285
Enabling DbC in BIOS	286
Connecting the Host to the Target.....	286
Installing the Intel DCI Driver	287
Testing the Connection	287
Connecting with SourcePoint.....	288
SourcePoint Command Language	289
Introduction	289
Syntax Notation.....	290
Comments.....	291
Constants	292
Data Types	294
Expressions	296
Debug Variables	298
Debug Variable Arrays.....	299
Debug Procedures	301
Control Variables	305
Command Files	307
Filenames	308
Viewpoint Processor and Processor Overrides	309
Symbolic References	310

Qualified Symbol Names	315
Commands and Control Variables	317
aadump	317
abort	318
abs	319
acos	321
advanced	322
asin	323
asm	324
asmmode	332
atan	333
atan2	334
autoconfigure	335
base	336
bell (beep)	339
bits	340
break	342
breakall	343
cachememory	345
cause	347
Character Functions	349
clock	356
continue	357
cos	358
cpubreak, cpuremove, cpudisable, cpuenable	359

cpuid_eax.....	361
cpuid_ebx.....	363
cpuid_ecx.....	365
cpuid_edx.....	367
createprocess	369
cscfg and local_cscfg.....	370
csr	377
ctime.....	379
cwd.....	380
dbgbreak, dbgremove, dbgdisable, dbgenable	382
defaultpath	384
#define	386
define	387
definemacro	389
deviceconfigure.....	391
devicelist	392
devicescan	396
disconnect.....	397
displayflag	398
do while	399
dos	401
dport.....	402
drscan	403
edit	405
editor	406

emubreak, emuremove, emudisable, emuenable	407
emulatorstate	409
encrypt	410
error	411
eval	412
evalprogramsymbols	413
execution point (\$)	415
exit	417
exp	418
fc	419
fclose	420
feof	422
fgetc	424
fgets	426
first_jtag_device	428
flist	429
flush	431
fopen	432
for	434
forward	436
fprintf	438
fputc	439
fputs	440
fread	441
fseek	443

ftell.....	445
fwrite.....	446
getc	448
getchar	449
getnearestprogramsymbol	450
getprogramsymboladdress	452
gets	454
globalsourcepath.....	455
go	457
halt	460
help	461
homepath	462
idcode	463
if	465
include.....	467
invd.....	469
irscan.....	470
isdebugsymbol	472
isem64t	473
isprogramsymbol.....	474
isrunning	476
issleeping	478
issmm.....	480
itpcompatible	482
jtagchain.....	483

jtagconfigure	485
jtagdeviceadd	486
jtagdeviceclear	487
jtagdevices	488
jtagscan	489
jtagtest	490
keys	492
last	493
last_jtag_device	495
left	496
libcall	497
license	500
linear	501
list, nolist	503
load	505
loadbreakpoints	507
loadlayout	508
loadproject	509
loadtarget	510
loadwatches	511
log, nolog	512
log10	513
loge	514
logmessage	515
macropath	516

Memory Access	517
messagebox.....	522
mid	524
msgclose	525
msgdata	526
msgdelete	528
msgdr	529
msgdump	531
msgir	533
msgopen	535
msgreturndatasize	536
msgscan.....	538
msr	540
num_activeprocessors	542
num_all_devices	543
num_devices	544
num_jtag_chains.....	546
num_jtag_devices	547
num_processors	548
num_uncore_devices.....	549
openipc	550
pause	551
physical	553
port.....	555
pow.....	557

print cycles	558
printf	559
proc	562
processorcontrol	563
processorfamily	565
processormode	566
processors	567
procesortype	568
projectpath	569
putchar	570
puts	571
rand	572
readsetting	573
reconnect	574
Register Access	575
reload	577
reloadproject	578
remove	579
reset	581
restart	583
return	584
right	585
runcontroltype	586
safemode	587
save	589

savebreakpoints	590
savelayout	591
savewatches	592
selectdirectory	593
selectfile	594
shell	595
show	596
sin	598
sizeof	599
sleep	600
softbreak, softremove, softdisable, softenable	601
sprintf	603
sqrt	604
srand	606
step	607
stop	609
strcat	610
strchr	611
strcmp	613
strcpy	615
_strdate	617
string [] (index into string)	619
strlen	620
_strlwr	621
strncat	622

strncmp	623
strncpy	625
strpos	626
strstr	628
_strtime	629
strtod	631
strtol	633
strtoul	635
_strupr	637
swbreak	638
switch	640
swremove	642
tabs	643
tan	644
tapdatashift	645
tapstateset	647
targpower	649
targstatus	650
taskattach	652
taskbreak, taskremove, taskdisable, taskenable	653
taskend	655
taskgetpid	656
taskstart	657
tck	658
time	660

uncoreconfigure	661
uncorescan	662
#undef	663
unload	664
unloadproject	666
upload	667
use	669
verify.....	670
verifydeviceconfiguration	672
verifyjtagconfiguration	673
version	674
viewpoint	675
vpalias	676
wait.....	678
wbinvd	679
while	680
windowrefresh	682
wport	683
writesetting	685
yield	687
yieldflag	688
Index.....	689

Contacting ASSET InterTech

Phone: 888-694-6250 toll free in the U.S.
972-437-2800 outside the U.S.

For Sales and Info Contact emails and telephone numbers:
<http://www.asset-intertech.com/About-Us/Contact-Us>

For Support Contact emails and telephone numbers:
<http://www.asset-intertech.com/Support/Contact-Details>

Introduction to SourcePoint

What's New in SourcePoint 7.12

New Project Wizard Improvements - The New Project Wizard has been updated to automatically download newer versions of JTAG XML and target configuration files making it easier to connect to new targets.

Peek Memory - The Register view has a new context menu item call Peek Memory. This allows reading of 32 bits of data using a register as the memory address.

CScripts Support - SourcePoint now uses Python 3.6.9 for running CScripts rather than Python 2.7.

AET LBR Support - AET trace can now optionally include LBR trace for each event type.

What's New in SourcePoint 7.11

Build 76

IceLake Server (ICX) support - Run control support added for ICX. Single package only.

DCI support - Added support for DCI (DbC). Allows for debug through a USB connection (without a debug probe). See [Getting Started with DCI DbC](#) for more information.

MCE Breakpoints - New MCE breakpoint type allows trigger on machine check exceptions

Build 79

IceLake Server (ICX) support - Run control support added for two package ICX.

Build 80

Comet Lake (CML) support - Run control support added for Comet Lake

Tiger Lake (TGL) support - Run control support added for Tiger Lake

Build 89

Comet Lake (CML) G0 support - Run control support added for Comet Lake G0 stepping

Comet Lake (CML) DbC support - Run control support (via DbC) added for Comet Lake

Tiger Lake (TGL) DbC support - Run control support (via DbC) added for Tiger Lake

DbC connection status utility - DbC connection utility now provided with SourcePoint release. This utility is useful for troubleshooting DbC connection issues. It indicates whether the host computer's DCI driver has a valid connection to the target.

IceLake server (ICX) trace support - LBR trace and Intel PT trace are now supported on ICX.

Build 91

Comet Lake (CML) P0 support - Run control support added for Comet Lake P0 stepping

Build 95

IceLake Server (ICX) C0 support - Run control support added for ICX C0 stepping.

NDA processor support - Intel processors still under NDA are now supported in the standard SourcePoint release. It's no longer required to visit the SourcePoint Community to download support files.

Build 96

Jasper Lake support (JSL) - Run control support added for JSL

Build 97

Sapphire Rapids support (SPR) - Run control support added for SPR.

SKL-D support - Run control support added for SKL-D.

Build 99

Elkhart Lake support (EHL) - Run control support added for EHL

ICX AET support - AET trace support added for IceLake Server

Build 100

Rocket Lake (RKL-S) - Run control support added for RKL-S

SPR Trace support - LBR, BTS and Intel PT trace support added for SPR

Build 102

TGL P0 support - Run control support added for TGL P0 stepping

Build 103

ICX-D support - Run control support added for ICX-D Y0/U0 stepping

RKL Trace Hub support - Trace Hub support added for RKL

Eagle Stream CScripts support - Support added for Eagle Stream (SPR) CScripts.

Build 106

TGL R0 support - Run control support added for TGL R0 stepping

EHL RCX issue - Fixed issue with RCX sometimes cleared when running to hardware breakpoint

Build 107

ADL-S support - Run control support added for ADL-S. Only the Core cores are supported at this time. Atom cores are ignored.

Build 108

SPR B0 support - Run control support added for SPR B0 stepping

Build 109

SPR B0 dual package issue - Fixed issue with CS and SS registers first thread in second package sometimes cleared after running to breakpoint.

What's New in SourcePoint 7.10.4

Gemini Lake Support - Added support for GLK processors

Coffee Lake Support - Added support for CFL processors (requires NDA)

Cannon Lake Support - Added support for CNL processors (requires NDA)

IceLake Support - Added support for ICL processors (requires NDA)

CFL Trace - Added support for CFL trace (requires NDA). Includes Intel PT, UEFI and ME trace through the Trace Hub, and AET trace through the Trace Hub.

CNL Trace - Added support for CNL trace (requires NDA). Includes Intel PT, UEFI and ME trace through the Trace Hub, and AET trace through the Trace Hub.

VS 2015 Support - Added symbolic support for VS 2015 compiler.

Improved C-State Handling

ICL Trace - Added support for ICL trace (requires NDA). Includes Intel PT, UEFI and ME trace through the Trace Hub, and AET trace through the Trace Hub.

LBR Trace Bug Fix - Fixed bug that resulted in incorrect LBR trace when executing a go off of a breakpoint.

What's New in Sourcepoint 7.10.3

Trace Hub Support - SourcePoint now supports software / firmware trace through the Intel Trace Hub to system memory.

Intel PT Timestamp - SourcePoint now supports TSC, MTC and TMA packets in Intel PT. This allows Intel PT from multiple cores to be time aligned.

What's New in SourcePoint 7.10.2

Intel Processor Trace Support – Added support for Intel Processor Trace (Intel PT). See the Trace view section for more information.

Trace Search View - The Trace Search view has been added to support Intel PT. This view supports high level views of the trace. The view can be opened from the Trace view context menu.

Trace Statistics View - The Trace Statistics view has been added to support Intel PT. This view supports function profiling of instruction trace data. The view can be opened from the Trace view context menu.

What's New in SourcePoint 7.10.1

IvyTown - Support added for IvyTown including ureg_raw command and Python-CLI device model.

Alternate Processor Numbering - SourcePoint now supports both the Arium and ITP processor numbering schemes. The ITP numbering scheme is selected by setting the ItpCompatible control variable true.

LX-INT - The LX-INT tab in the Emulator Configuration dialog has been reworked to support the new LX-INT rev E-1 adapter.

GD Bit - SourcePoint now supports the GD bit in DR6 to break on target software modifying the DR registers.

Improved Program Load Verification - When verifying a program loaded symbols only, SourcePoint now allows individual sections to be excluded from the verify (e.g., the Init section in a Linux kernel load).

What's New in SourcePoint 7.10

Target Configuration - Target configuration is now handled by a series of new commands executed in the target configuration event macro. See the new [Target Configuration](#) Application Note for more information.

Python Support - SourcePoint now has limited support for the Python Command language.

1024 Processor Support - The number of processors (threads) supported has been increased from 64 to 1024.

Haswell Support - The Haswell processor is now supported including Haswell NI instructions.

Editing of Macro File Errors - The Macro error dialog now allows opening an editor to fix macro file errors, and then resumption of the macro.

SelectFile and SelectDirectory commands - These new commands open dialogs to allow a filename or directory path to be returned to an nstring variable.

Symbol Search Improvement - The Symbol Finder dialog (Edit | Find Symbol) now allows searching for a symbol across multiple programs.

Program Save Improvement - The Program Save dialog now allows for saving a region of memory as an axf file. The Save (Upload) command also supports this.

Viewpoint View Improvements - The Viewpoint window now allows sleeping processors to be automatically hidden from the display. It also allows for individual processors to be hidden.

The following target configuration commands have been added (see [Target Configuration](#) Application Note for more information):

[jtagtest](#)
[jtagscan](#)
[jtagconfigure](#)
[verifyjtagconfiguration](#)
[uncorescan](#)
[uncoreconfigure](#)
[devicescan](#)
[deviceconfigure](#)
[verifydeviceconfiguration](#)
[autoconfigure](#)
[disconnect](#)
[reconnect](#)

What's New in SourcePoint 7.9

Version 7.9.1

Multiple Trace views - Support for up to 10 trace views showing any combination of LBR, BTS and Event Trace.

Time-aligned Trace views - Trace views can be time aligned so scrolling one scrolls the other (BTS and Event trace only).

Advanced mode - Show or hide advanced configuration settings by enabling or disabling advanced mode (Options | Preferences | General).

Search across programs - The Symbol Finder Dialog (Edit | Find Symbol) now allows for searching across all programs loaded in SourcePoint (useful in EFI debug).

Version 7.9

Event Trace - Support for Intel Event Trace (requires special BIOS or processor patch)

BTS Timestamp - BTS trace data is now timestamped (requires special BIOS or processor patch).

What's New in SourcePoint 7.8

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).
 - 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.

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](#)

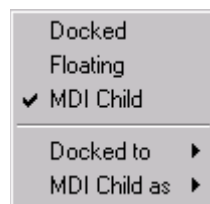
[Menu toolbar](#)

[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



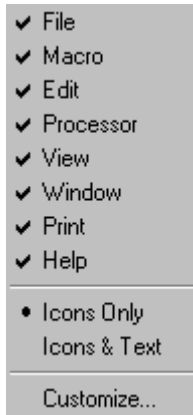
File	Load Project, Reload Project, Save Project, Save Project As, Load Program, Reload Program, Load Macro, Update Emulator Flash, Program Target Flash
Macro	Execute Macro 0, Execute Macro 1, Execute Macro 2, Macro 3*
Edit	Cut, Copy, Paste, Search, Replace
Processor	Go, Stop, Step Into, Step Over, Step 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

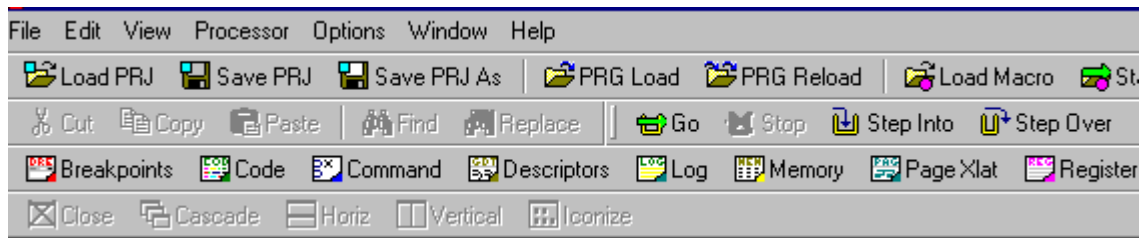
Context Menus

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.

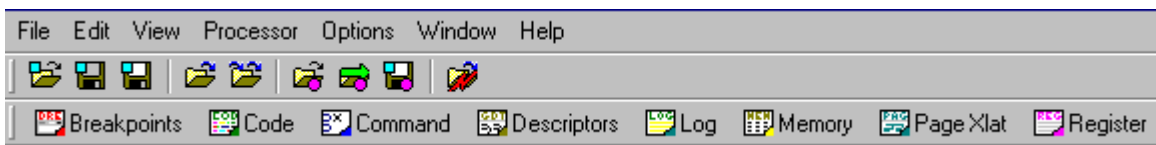


File, Macro, etc. menu items. The menu items in the top section of the menu let you choose which groups of icons you want to display. All icons are displayed by default.

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

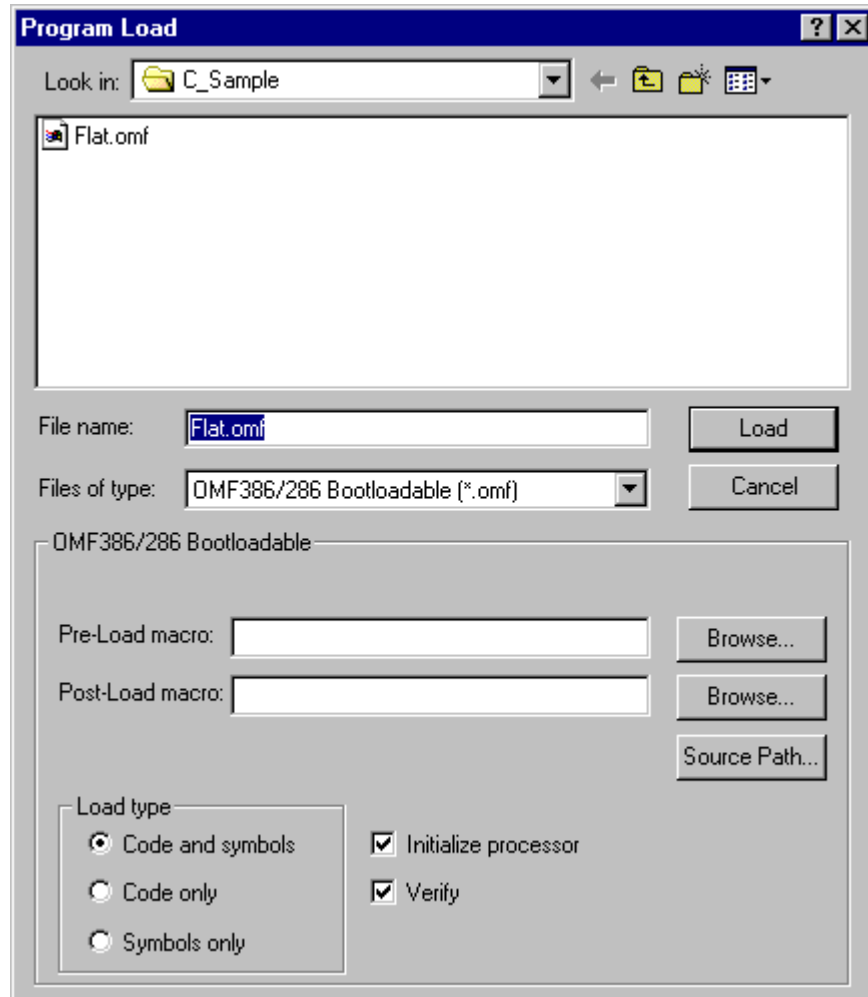
To save a user-generated SourcePoint layout, click on **Save Layout** option in the **File** menu. Select the file you want to save.

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 type	Initialize processor(1)	Macros(2)	Offset(3)	Address(4)

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.

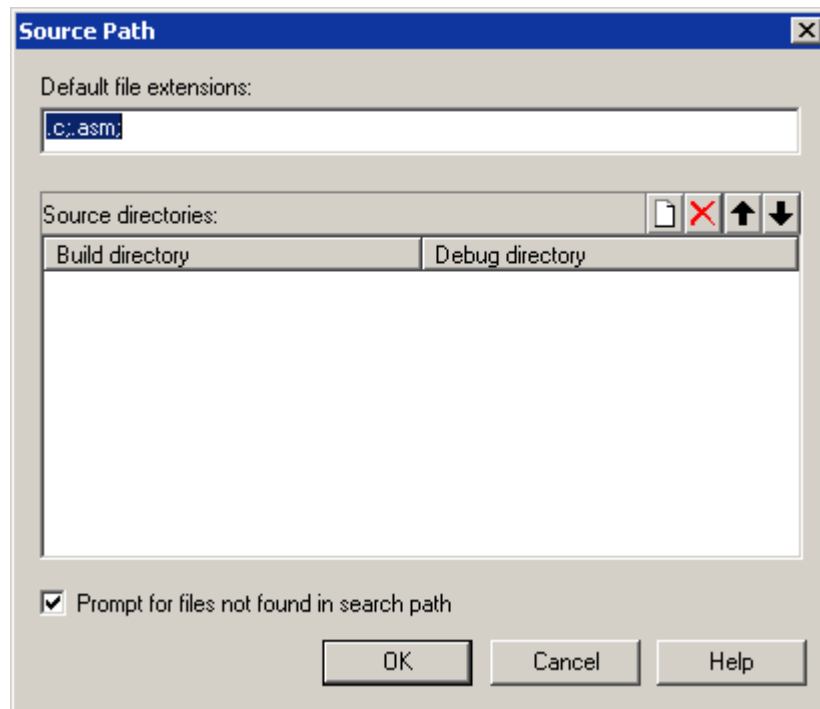
Pre-Load Macro. The **Browse** button allows you to select a macro file that runs prior to the loading of code or symbols. The primary use of this macro is to automate initialization of your target to a known state prior to the actual program load. If this feature is not desired, then leave the field 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.

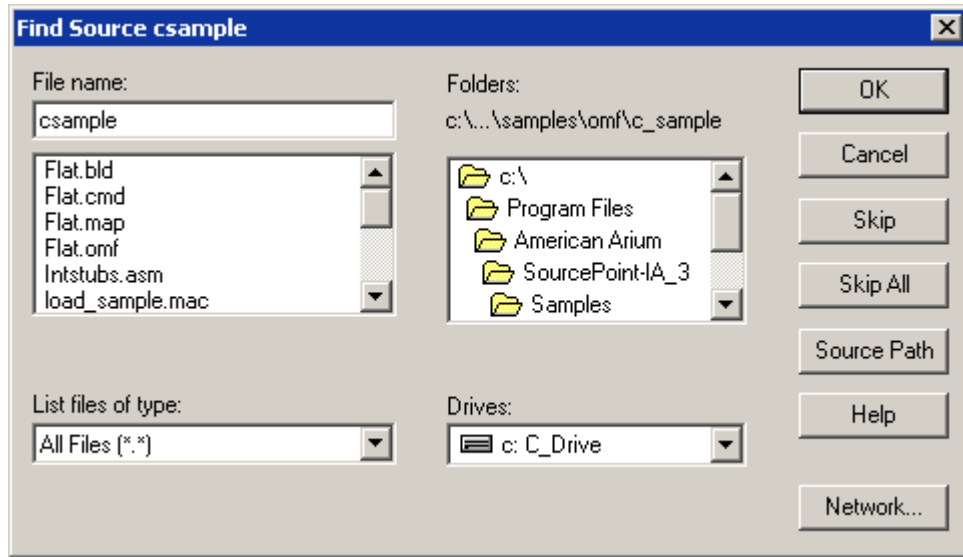
Source path button. Click the **Source path** button to open the **Source Path** dialog box. In it you can specify where the loader should locate source files and what file extensions it should look for. The **Source Path** dialog box now supports both path maps and search paths (for OMF). A path map 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

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 the Macro menu item from the File menu to access the following options: Load Macro and Configure Macros.

Load Macro Option

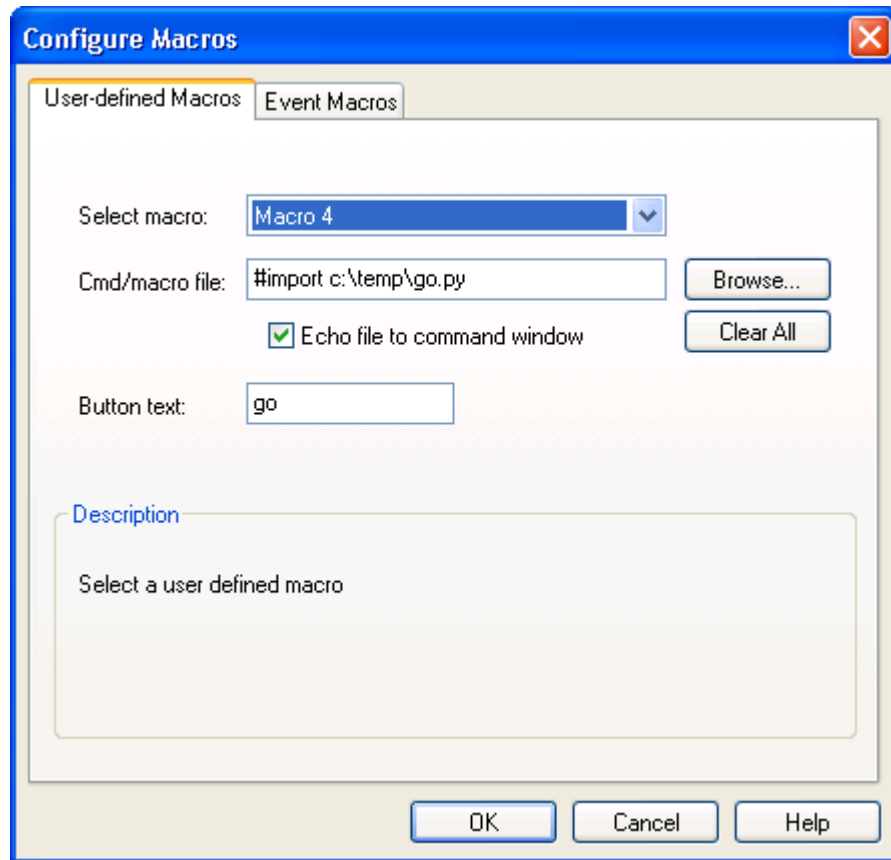
Select the Load Macro option to load an existing macro file.

1. Select the Load Macro option.
2. A standard Open file dialog box displays.
3. Select the desired macro by clicking on it to highlight it or browse for it.
4. Click the Open button. The macro loads, and the Open file dialog box closes.

Configure Macros Option

Select Configure Macros to open the Configure Macros dialog. There are two types of macros: User-defined macros and Event macros. User-defined macros are macros that you create and link to the buttons in the Macro toolbar. Event macros are macros that run when a specific event occurs (e.g., target reset, project load, etc.).

User-Defined Macros Tab



Configure Macros dialog box

Select macro. Use this drop down list to select a user-defined macro number. Up to 20 user-defined macros can be specified.

Note: You must add the macro icons to the icon toolbar prior to adding user-defined macros 4 - 19. If you do not, the select macros drop down text box shows only Macros 0 - 3. To add macros to the toolbar, right-click on the Macro toolbar and select Customize.

Cmd/macro file. Use this box to specify the macro file to execute. Select the Browse button to find the file. Clearing this field removes the macro definition.

Alternatively, you can specify a command to execute. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly. Multiple commands can be separated by semicolons.

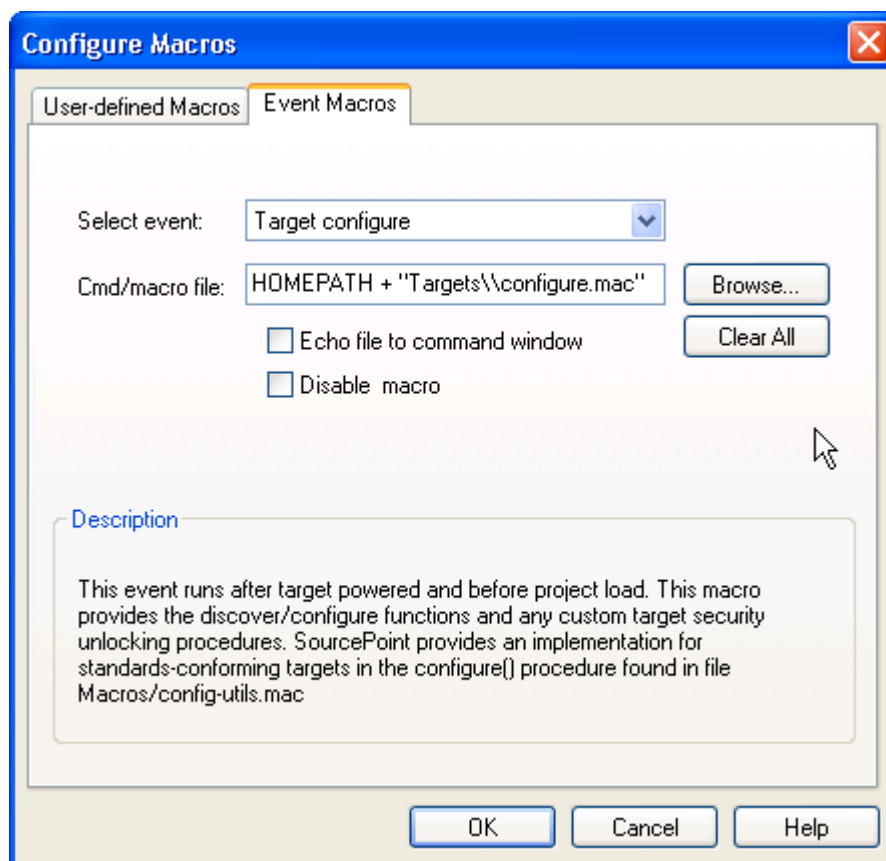
Echo file to command window. Enabling this option causes the contents of the macro file to be echoed to the Command window.

Button text. By default user-defined macro buttons are labeled "Macro 0," "Macro 1," etc. This field can be used to display more descriptive text. The text is visible on the toolbar only when you choose to display both the icon and text.

Clear All. Use this button to remove all macros definitions.

Note: User-defined macros are saved in target configuration files. See [Option Menu - Save Target configuration File](#) Menu Item in SourcePoint Environment for more information.

Event Macros Tab



Event Macros dialog box

Select event. Choose the event to link a macro with. When the event occurs, the macro file will be executed. See Event descriptions below.

Cmd/macro filename. Use this box to specify the macro file to execute. Select the Browse button to find the file.

Alternatively, you can specify a command to execute. If the text entered in this box begins with a "#" character, then it is considered to be a command and is executed directly. Multiple commands can be separated by semicolons.

Echo file to command window. Enabling this option causes the contents of the macro file to be echoed to the Command window.

Disable macro. Enabling this option temporarily disables an event macro from running.

Clear All. Use this button to remove all macros definitions.

Note: Event macros are saved in target configuration files. See [Option Menu - Save Target configuration File Menu](#) Item in SourcePoint Environment for more information.

Event	Description
Breakpoint (any)	Macro executes after any breakpoint is hit. This macro is in addition to any individual breakpoint macros that have been set.
Emulator Connected	Macro executes after SourcePoint connects to the emulator, but before target configuration occurs.
Go	Macro executes just prior to sending the Go command to the emulator.
Project Load	Macro executes immediately after loading or reloading a project.
Project Unload	Macro executes immediately after unloading a project (which includes closing SourcePoint).
Reset (before)	Macro executes just prior to sending the Reset command to the emulator.
Reset detected	Macro executes when the emulator detects a target reset initiated by either SourcePoint or the target. If the target requires security unlocking, this event can be used to run an unlock macro.
Reset complete	Macro executes immediately after target reset has completed.
Reset (after)	Macro executes after a user-initiated reset completes. This event does not occur if Run after reset is enabled.
Startup	Macro executes when SourcePoint starts.
Stop (user)	Macro executes after the Stop button is pressed, and the emulator signals the target has stopped
Target configure	Macro executes after target power on and before project load. This macro provides the discover/configure functions and any custom target security unlocking procedures (when required). See the Target Configuration Technical note for more information.
Target power on detected	Macro executes when the emulator detects a power on transition. If the target requires security unlocking, this event can be used to run an unlock macro.
Target power on complete	Macro executes when the emulator signals that the target power cycle is complete.
Target stop	Macro executes whenever the target stops (either because of a breakpoint or the stop key being pressed). Macro executes before SourcePoint does any automatic memory or register reads to refresh state.

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.

File Menu - 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

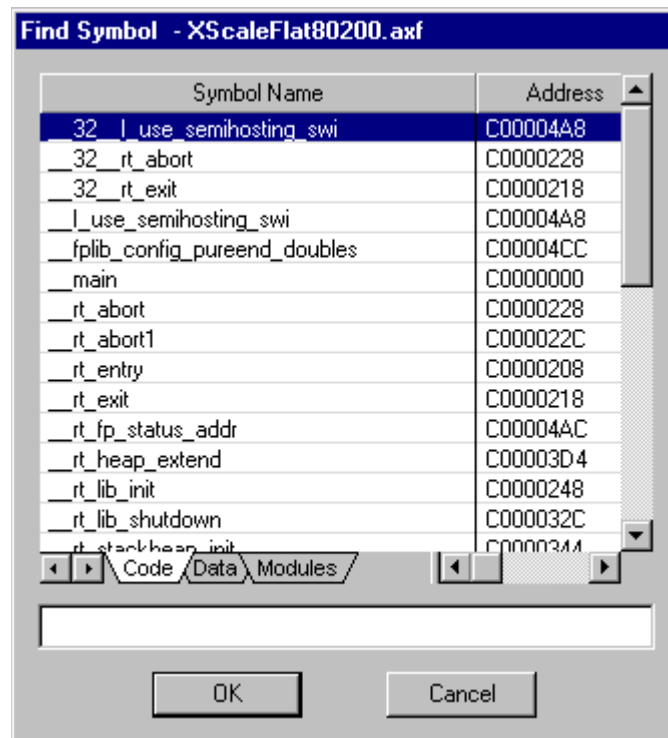
Use the **Find** menu item to enter the text to be found and then clicking the **Find Next** button. Options may be selected prior to the search to set the direction of search and set case sensitive constraints.

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."](#)

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

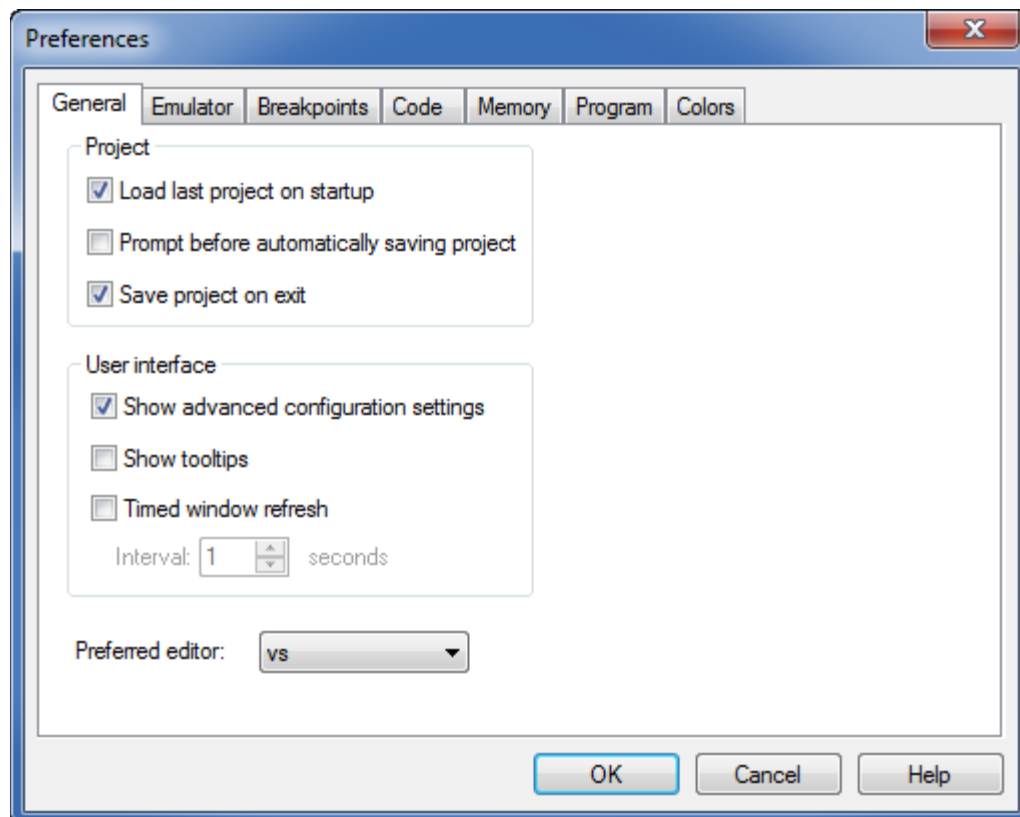
To set, change, or modify SourcePoint preferences, select **Options|Preferences** on the menu bar. The **Preferences** dialog box displays with the following tabs displayed: **General**, **Emulator**, **Breakpoints**, **Code**, **Memory**, **Program**, and **Colors**.

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](#)
[IPC Tab](#)

General Tab

The section under the **General** tab contains options that apply to all of SourcePoint.



General tab under Options|Preferences

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.

Prompt before automatically saving project. When enabled, SourcePoint checks if any key configuration settings have changed prior to writing these settings to the project file. If there are changes, SourcePoint will display a dialog, and prompt the user whether these changes should be saved. The default for this option is enabled.

Save project 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 Project 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 to save the file using the current name and location, or File|Save Project As to save it as another file name or location.*

Show advanced configuration settings. When enabled, all configuration options are displayed including seldom-used advanced options. When disabled, only the most common configuration settings are displayed. The default for this option is enabled.

Settings affected include emulator configuration settings, memory map settings, certain trace configuration settings, and processor control mask settings in the Viewpoint view.

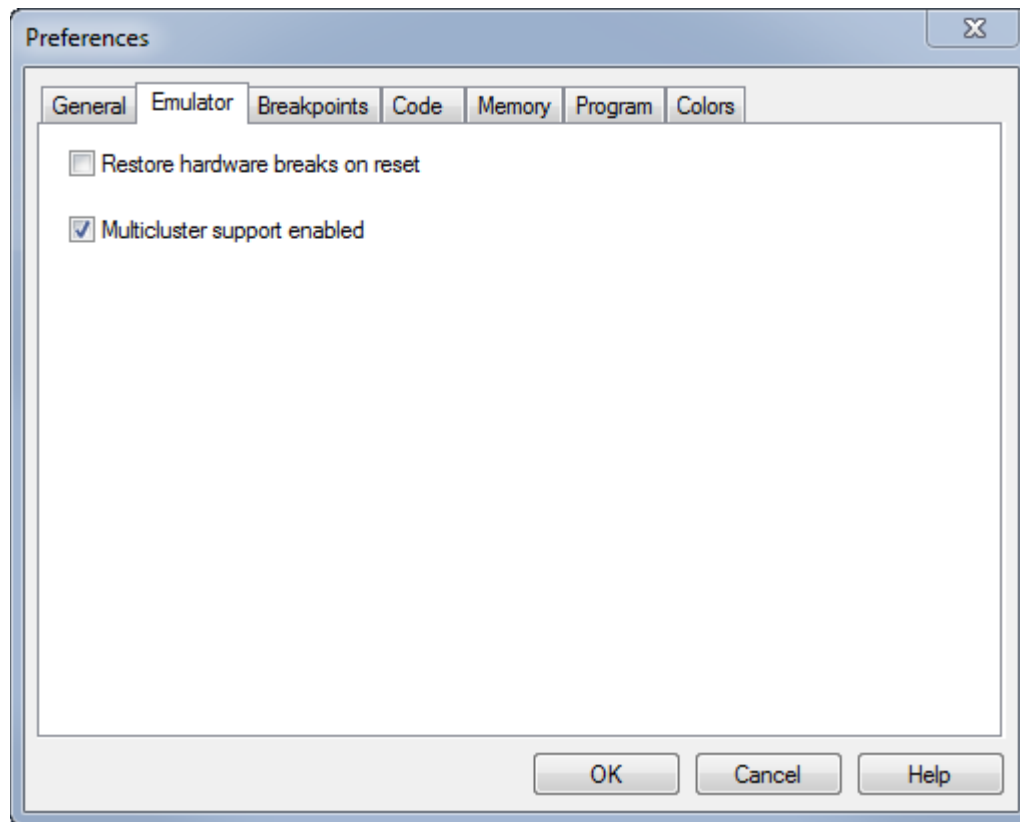
The Advanced control variable can be used to change this setting from the Command window.

Show Tooltips. This option enables flyover help in the Code, Trace and Memory windows and is enabled by default.

Timed window refresh. 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.

Emulator Tab

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



Emulator tab under Options|Preferences

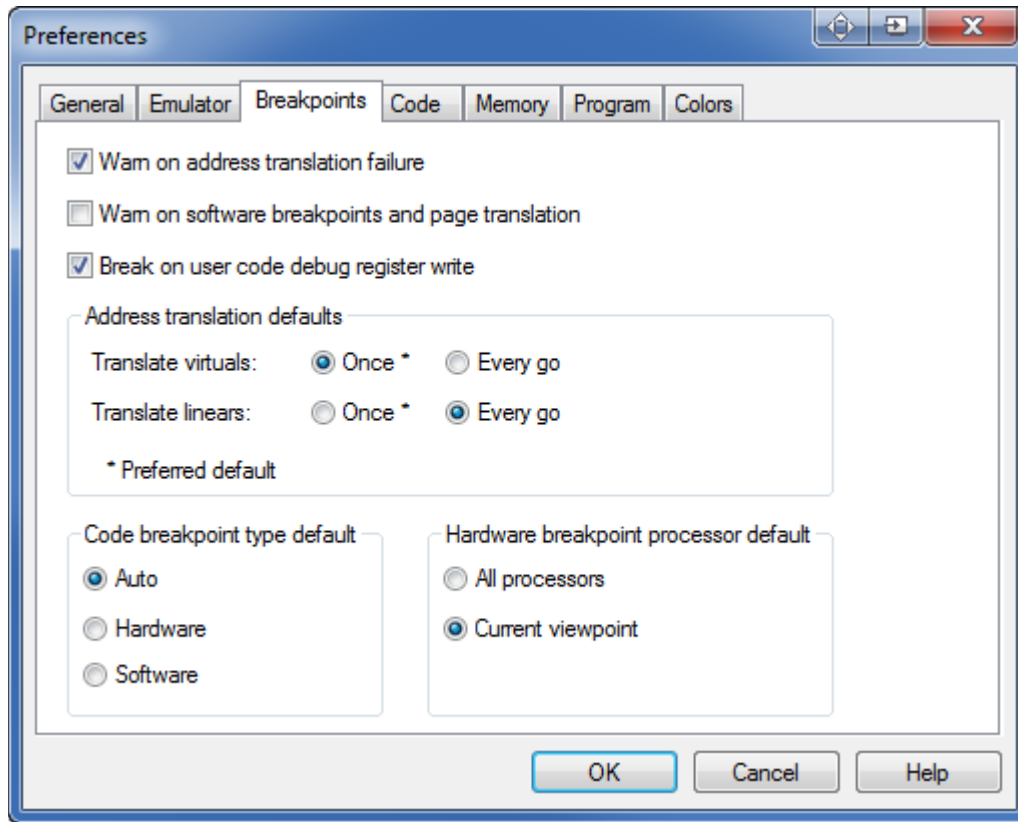
Restore hardware breaks on reset. Select this option to restore, upon target reset, the hardware breakpoints and then start the processor. Clearing this option results in a loss of all hardware breakpoints if a target reset occurs while the processor is running.

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.

Multicuster support enabled. Put a check mark beside this option if you want to enable multicuster support.

Breakpoints Tab

The **Breakpoints** tab displays settings related to breakpoints.



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.

Break on user code debug register write. When this option is enabled, SourcePoint will stop execution whenever user code attempts to modify a processor debug register. This prevents conflicts between user code and SourcePoint use of these registers to implement Hardware breakpoints

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.

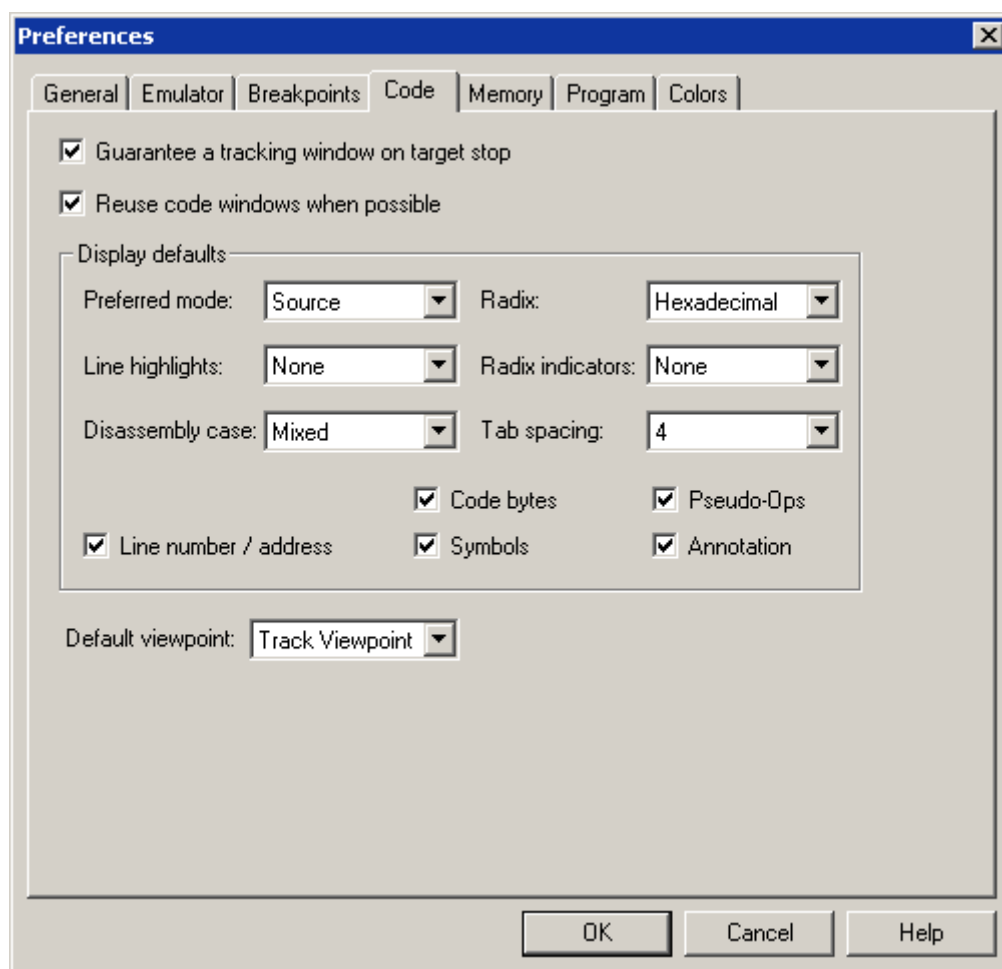
Code breakpoint type default. The default is **Auto**. When **Auto** is selected, for a breakpoint explicitly set in code (such as through the **Code** window), a software breakpoint is used if one is available; otherwise, a processor breakpoint is used if one is available. For temporary breakpoints 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.

Hardware breakpoint processor default. These controls are only visible when connected to a multiprocessor, homogeneous target (all processors are of the same type). When **All processors** is selected, setting a hardware breakpoint in SourcePoint will cause the breakpoint to be set on every processor. When **Current viewpoint** is selected, setting a hardware breakpoint in SourcePoint will cause the breakpoint to be set on a single processor. The **All processors** setting is useful for symmetric multiprocessing environments, where code with the breakpoint set could be dispatched on a different processor at a later time.

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 of 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.

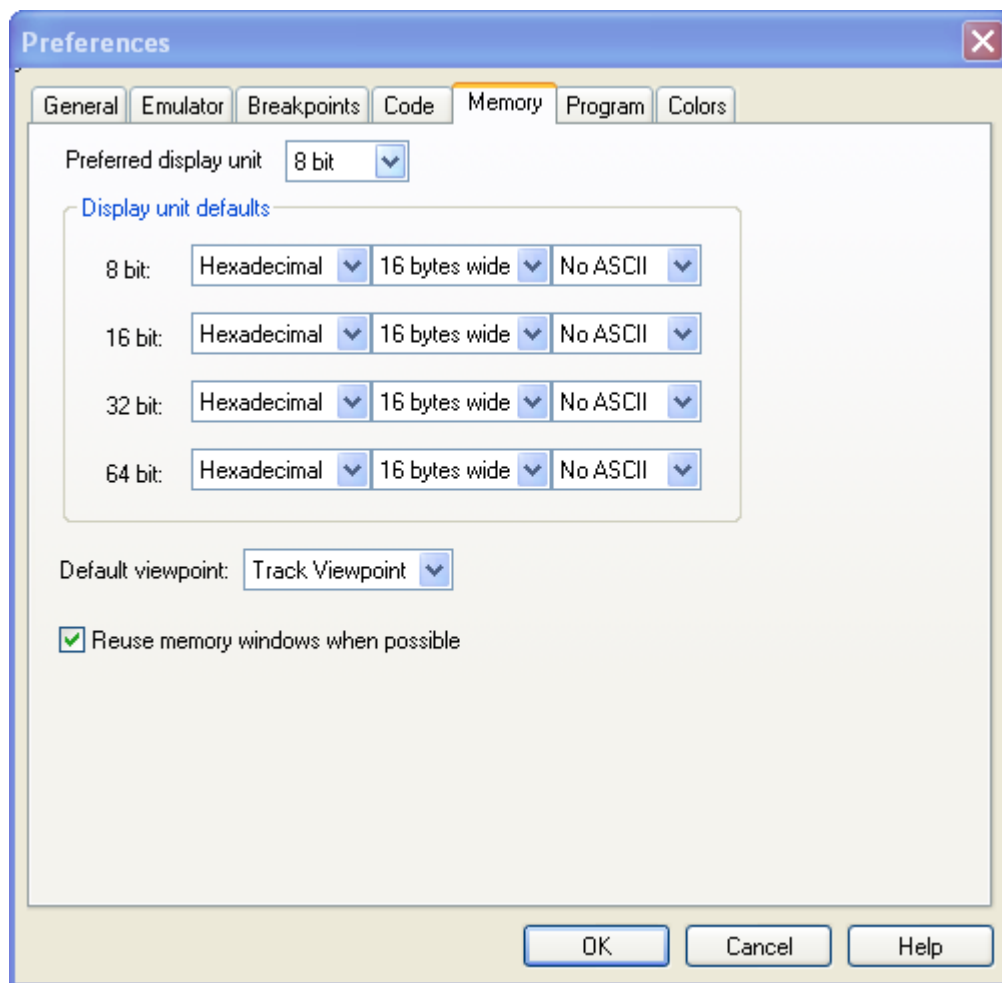
Display Defaults. There are a number of options you can set in this field. They are described briefly below.

- **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**, **Decimal**.
- **Radix indicators.** Options are **Prefix**, **Suffix**, **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).

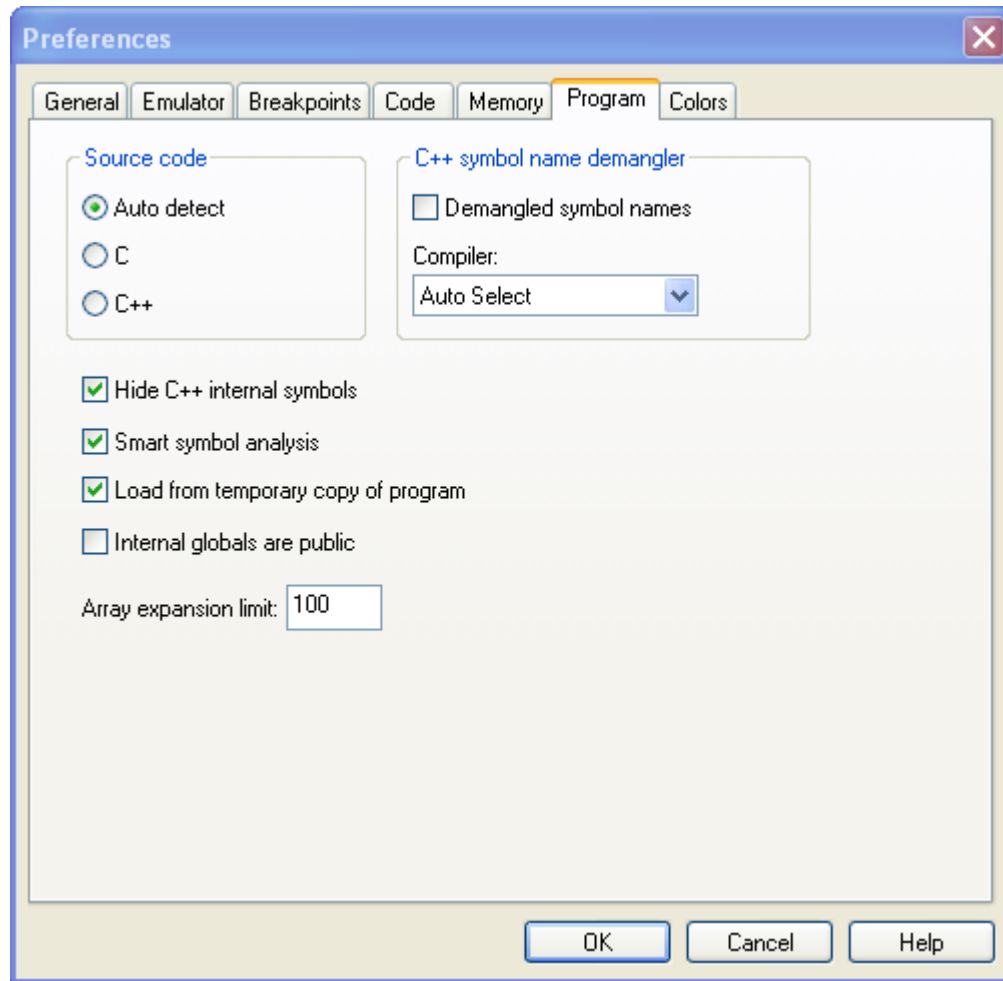
Display unit defaults section. This section allows you to set the preferences for each Display mode. The options for each are for the **Radix (hex, sign, unsign)**, **Display** width (in bytes) and **ASCII/No ASCII**.

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**.

Reuse memory windows when possible. If this option is enabled, SourcePoint attempts to reuse existing **Memory** windows rather than create new ones. This applies to Memory windows that may be opened from the context menus of the **Symbols**, **Trace**, or **Breakpoints** windows.

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 lets you view Code windows, including disassembly and source code, while the target is running. If the option is disabled, the loader uses the original file you specified. This file is held open until the program is removed or the project is unloaded. The tradeoff is the ability for you to rebuild the program while it is loaded in the debugger in exchange for the time and space required to make the copy.

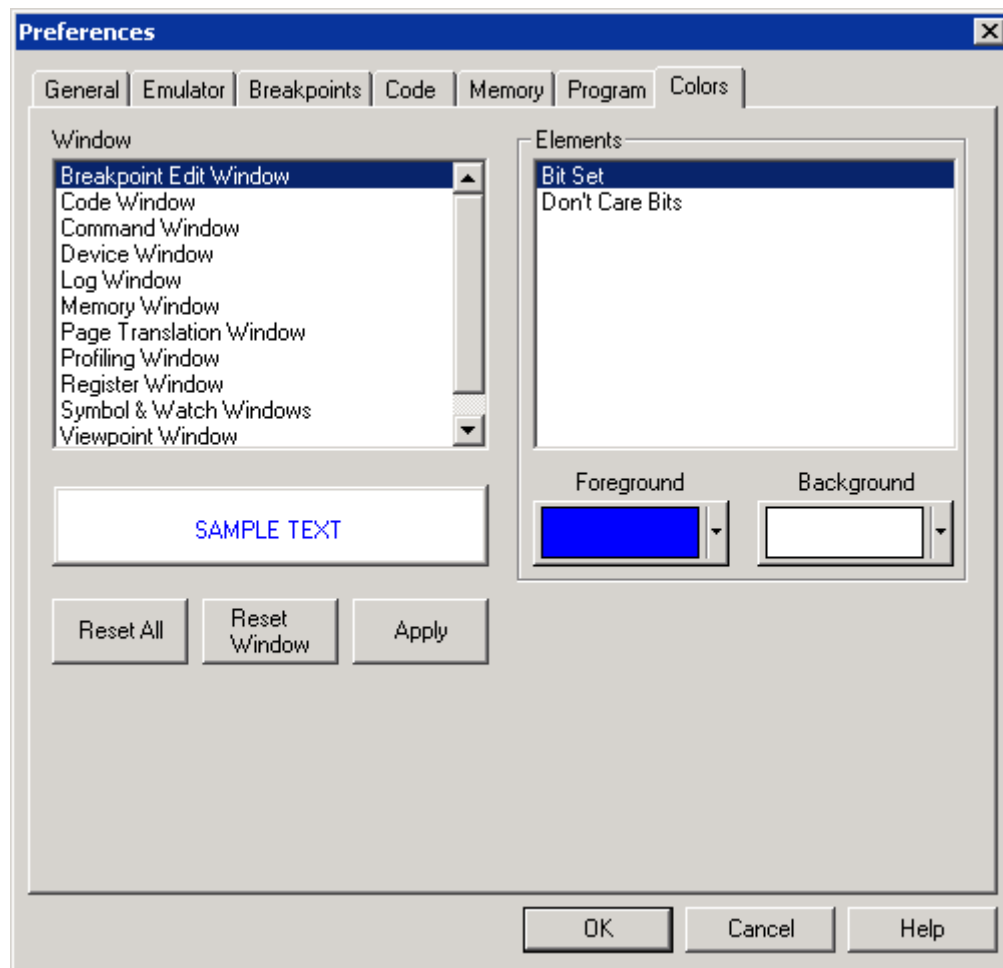
Internal globals are public. C and C++ "static" keyword provides support for function and variable definitions that are not visible outside of the containing module. These symbols are ignored by the linker for resolving external symbol references. Since these symbols are private to a module, their names are not required to be globally unique. Sometimes developers use module symbols as an encapsulation mechanism, but make it a practice to assign unique names to them. In this case, it is safe to enable Internal globals are public so that SourcePoint will publish module symbols in the global space.

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.

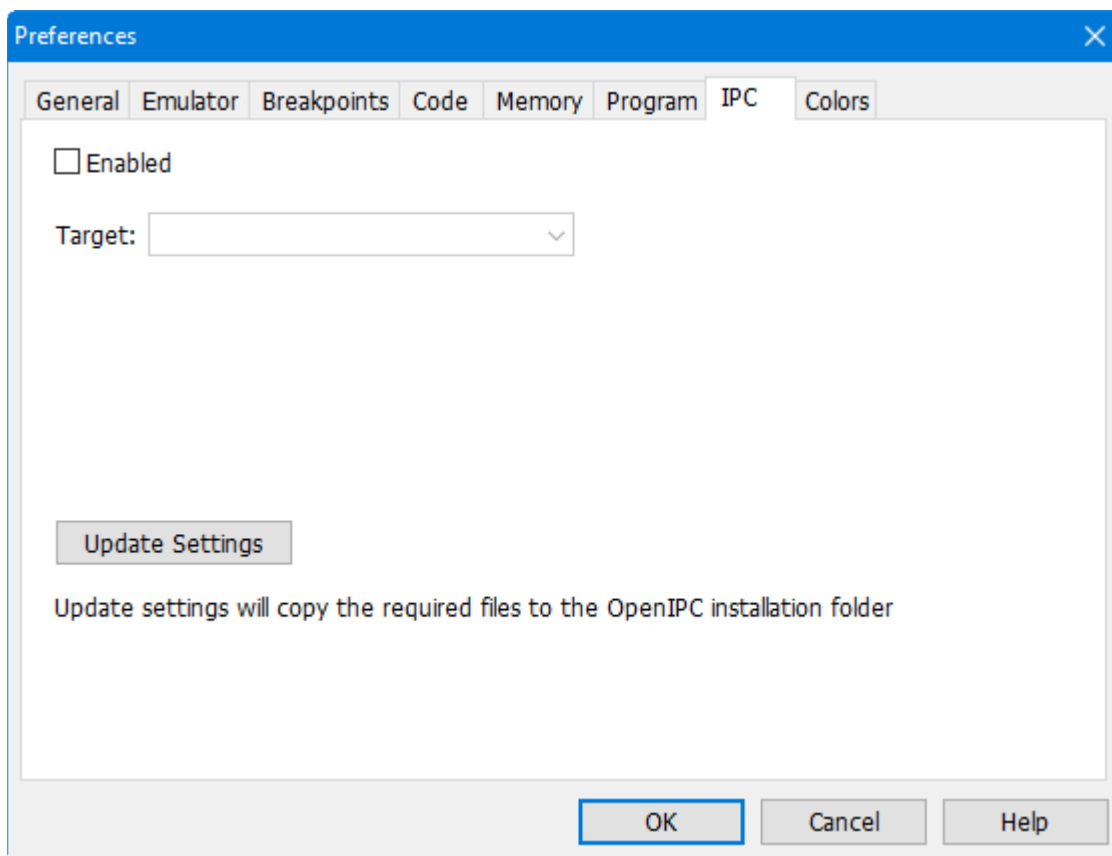
Reset Window button. Allows you to reset all the colors for the currently selected window back to the SourcePoint default colors.

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

IPC Tab

The **IPC** tab is used to configure how SourcePoint interfaces with the Open IPC component within Intel's System Debugger (ISD). This is typically used to run Customer Scripts (CScripts) from a separate Command Line Interface (CLI) window.

Note: ISD is not installed with SourcePoint. It must be obtained separately from Intel and installed before changing any settings in this tab. See the [OpenIPC Application Note](#) for more information.



IPC tab under Options|Preferences

Enabled. Enables Open IPC support. Typing OpenIPC in the Command View will open a CScripts CLI window.

Target. This dropdown lists the targets that are supported. Entries in this list correspond to the XML configuration files found in the PVT\Config folder (located where SourcePoint was installed).

Update Settings. This button is only used when IPC starts SourcePoint. It transfers information to IPC about the location of SourcePoint, and the project file to use. See the [OpenIPC Application Note](#) for more information.

Options Menu - Target Configuration Menu Item

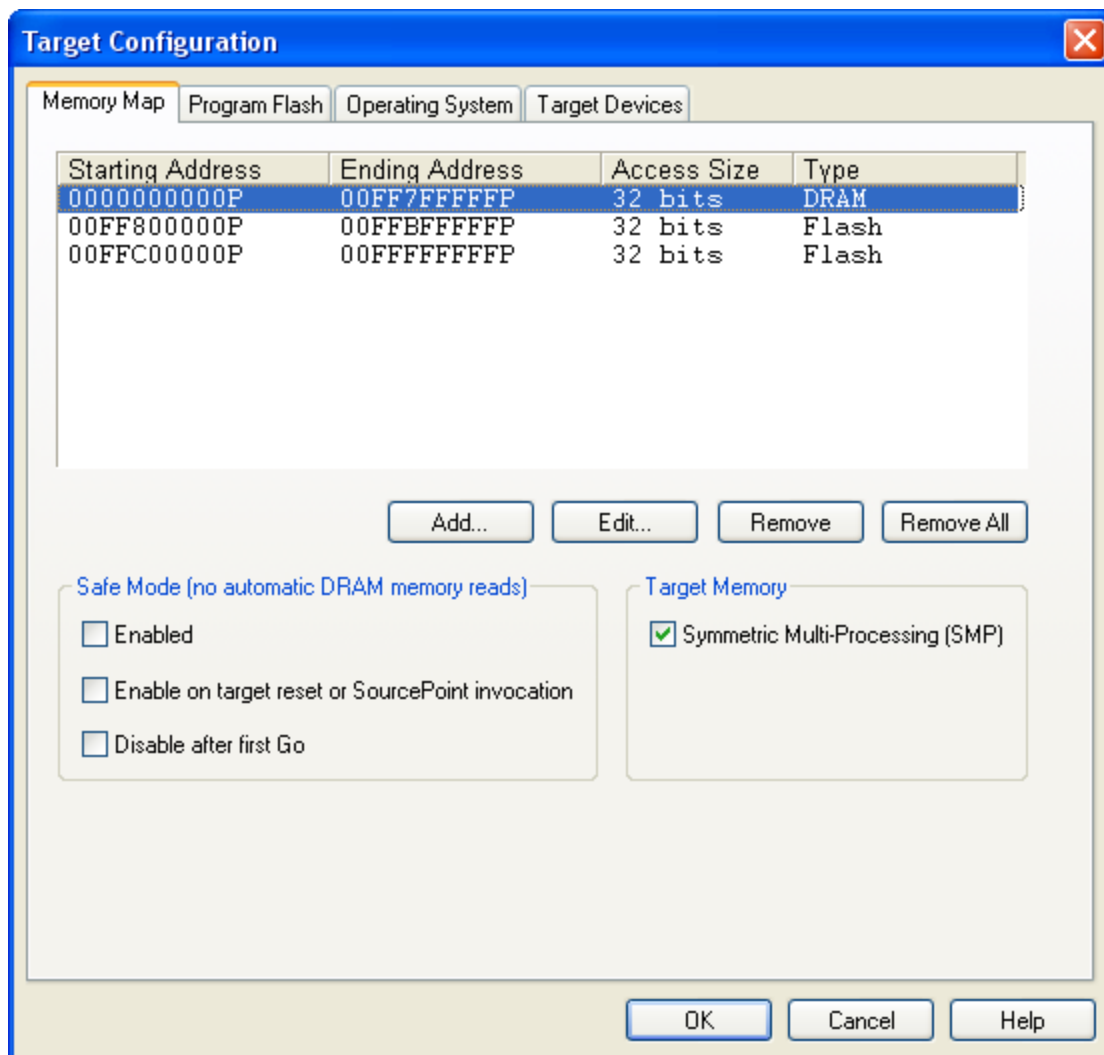
Select **Options|Target Configuration** to open the target configuration dialog.

To move directly to a particular tab, click here:

[Memory Map Tab](#)
[Program Flash Tab](#)
[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.

The screenshot shows the 'Target Configuration' dialog box with the 'Program Flash' tab selected. The dialog has four tabs: 'Memory Map', 'Program Flash', 'Operating System', and 'Target Devices'. The 'Program Flash' tab contains the following sections:

- Flash device(s):**
 - Device address: 00FF800000P (All)
 - Device type: T1: Lower device
 - ☐ Swap endian
- Flash image(s):**
 - Start address: FF800000P (with a 'Define...' button)
 - Filename: C:\Customers\Intel Flash\EFI Trace #1\Rom00B0.bin (with a file selection button)
- Target initialization:**
 - ☒ Run initialization cmd/macro
 - Cmd/macro file: HOMEPATH + "Macros\aa\SPIntelFlash.mac" (with a file selection button)

On the right side of the dialog, there are four buttons: 'Write', 'Verify', 'Erase', and 'Stop'. At the bottom of the dialog are 'OK', 'Cancel', and 'Help' buttons.

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.

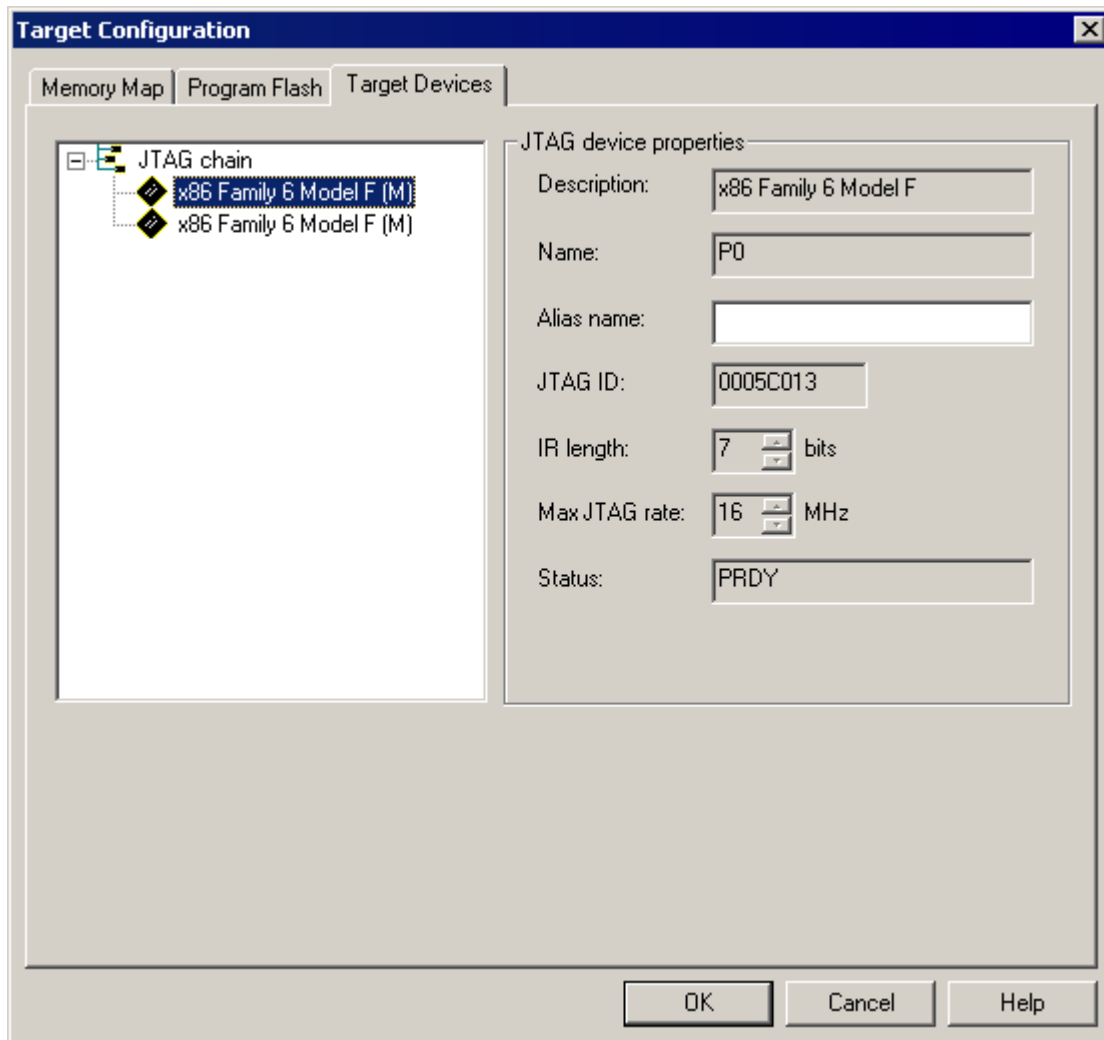
The **Verify** button verifies that the selected flash device is programmed correctly.

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

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

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.

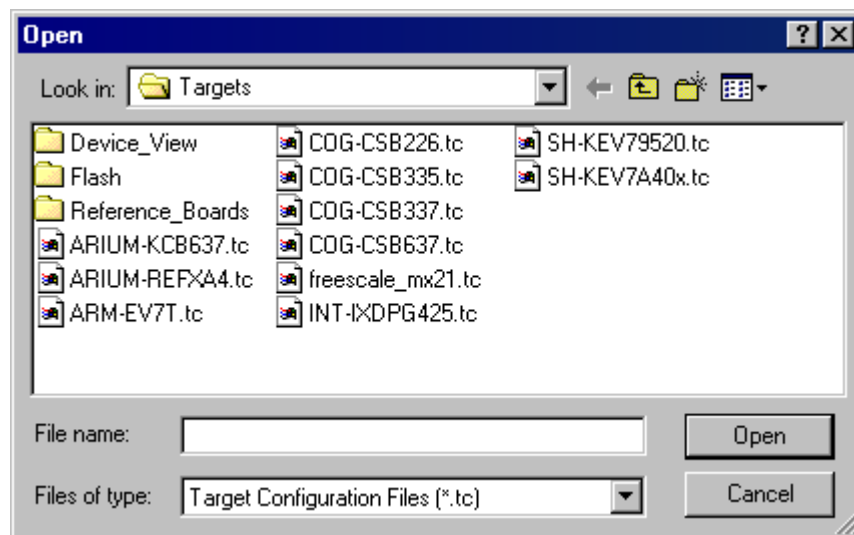
Selecting an item in the tree on the left displays its properties on the right. For the most part, the information is read-only and cannot be modified. However, if SourcePoint does not recognize the JTAG ID of the device, the JTAG properties section includes a **Description** with a drop-down text box 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

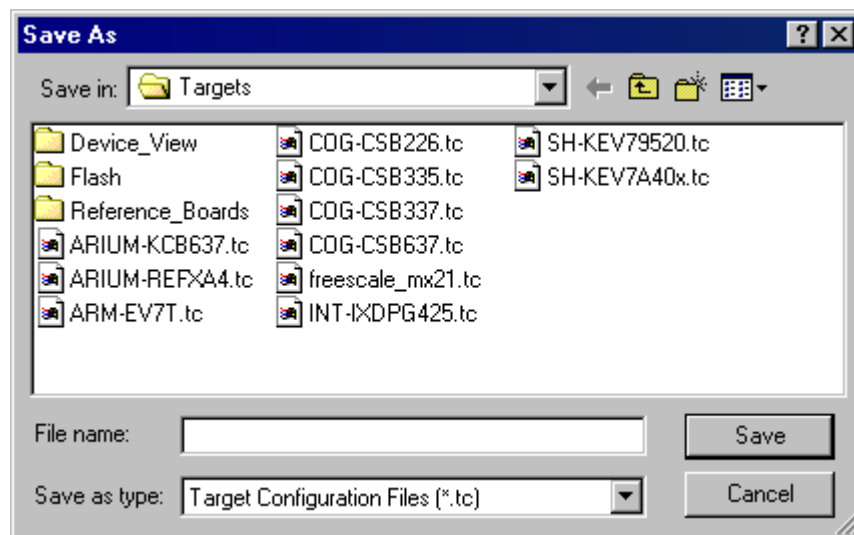
To load a target configuration file, click on the **Load Target Configuration File** menu item in the **Options** menu. Select the file you want to load.



Use this dialog box to load a target configuration file

Options 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.

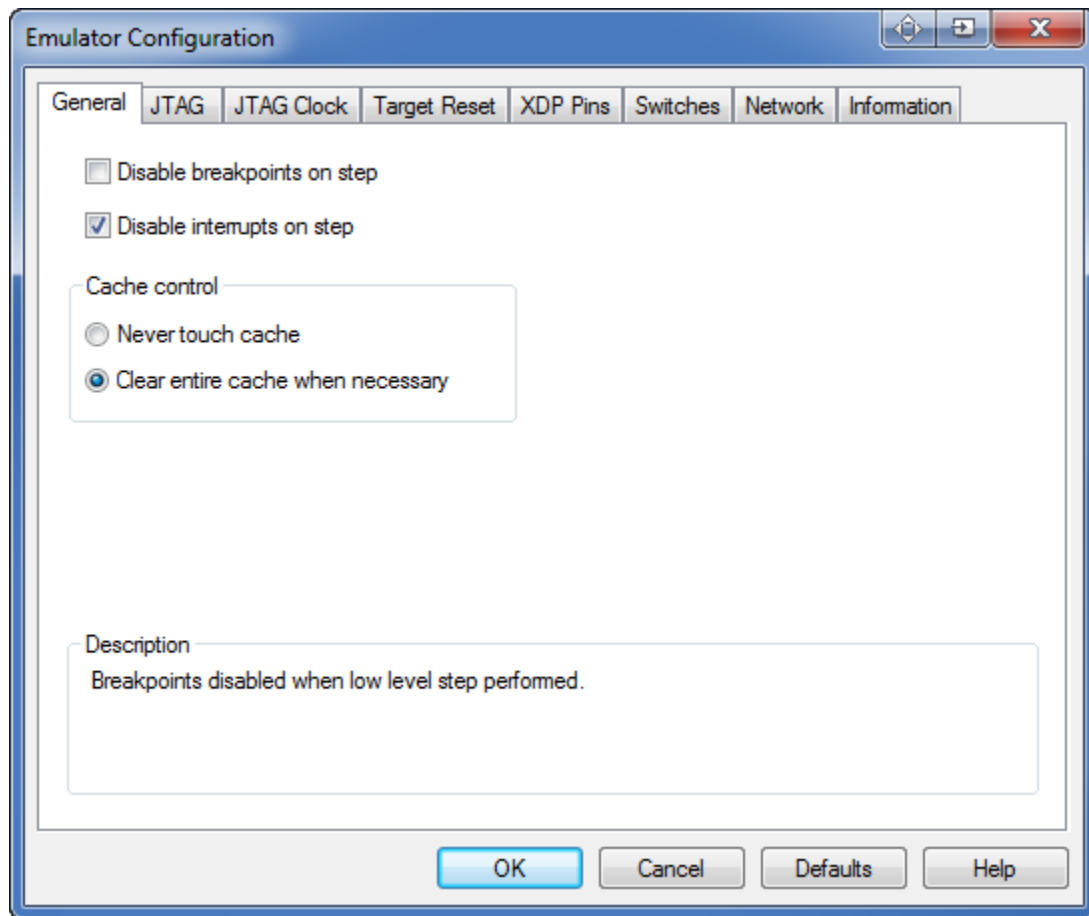
If you are configuring an emulator that supports an ethernet connection, an **Emulator Configuration** dialog box displaying several tabs appears.

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](#)
[XDP Pins Tab](#)
[Switches Tab](#)
[Network Tab](#)
[Information Tab](#)
[JTAG \(DCI\) Tab](#)

General Tab

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



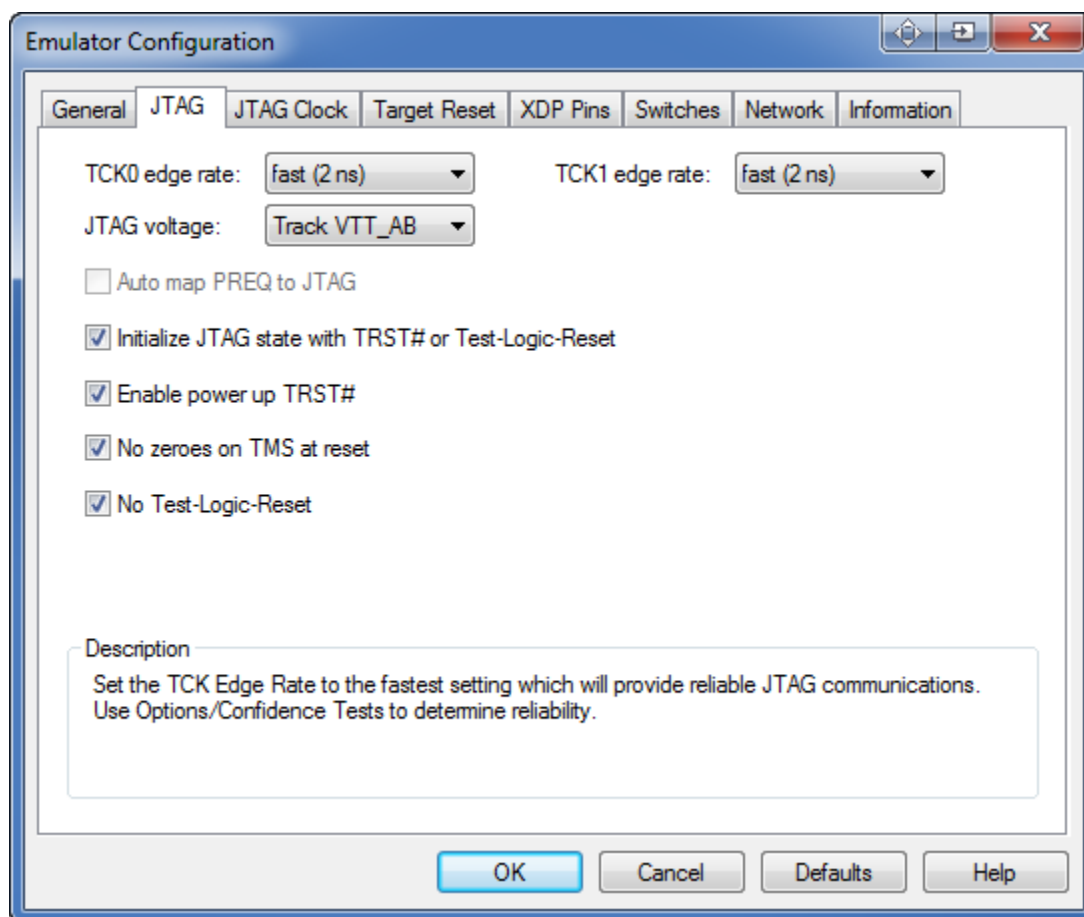
General tab under Options|Emulator Configuration

Disable breakpoints on step. When this option is enabled, breakpoints are disabled when low level stepping is performed. The option is designed to prevent breakpoints in SMM or interrupt code from halting the processor during a single step.

Disable interrupts on step. When this option is enabled, interrupts are disabled when low level stepping is performed. The option is designed to prevent pending interrupt handlers from executing during a single step.

JTAG Tab

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



JTAG Tab dialog box

TCK0 edge rate. 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. 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. 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 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.

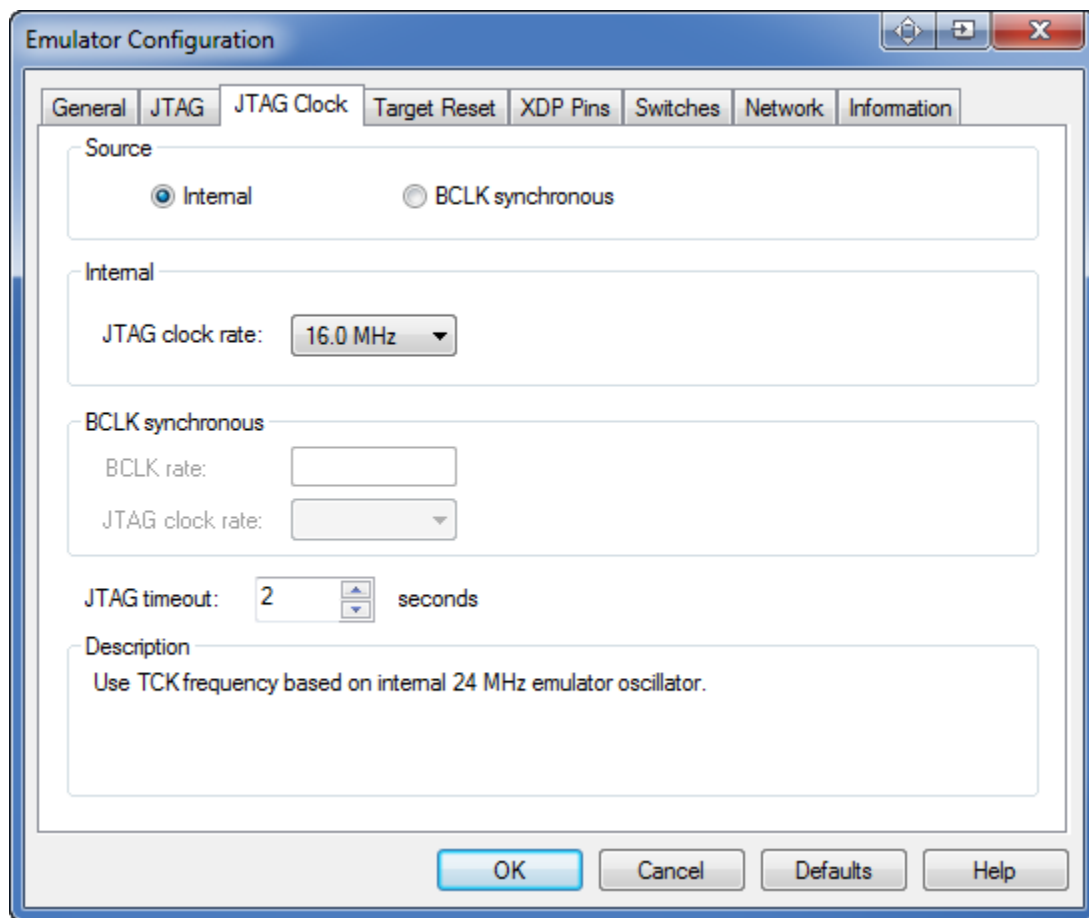
No zeroes on TMS at reset. This option determines whether zeroes are pumped out on TMS at reset.

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.

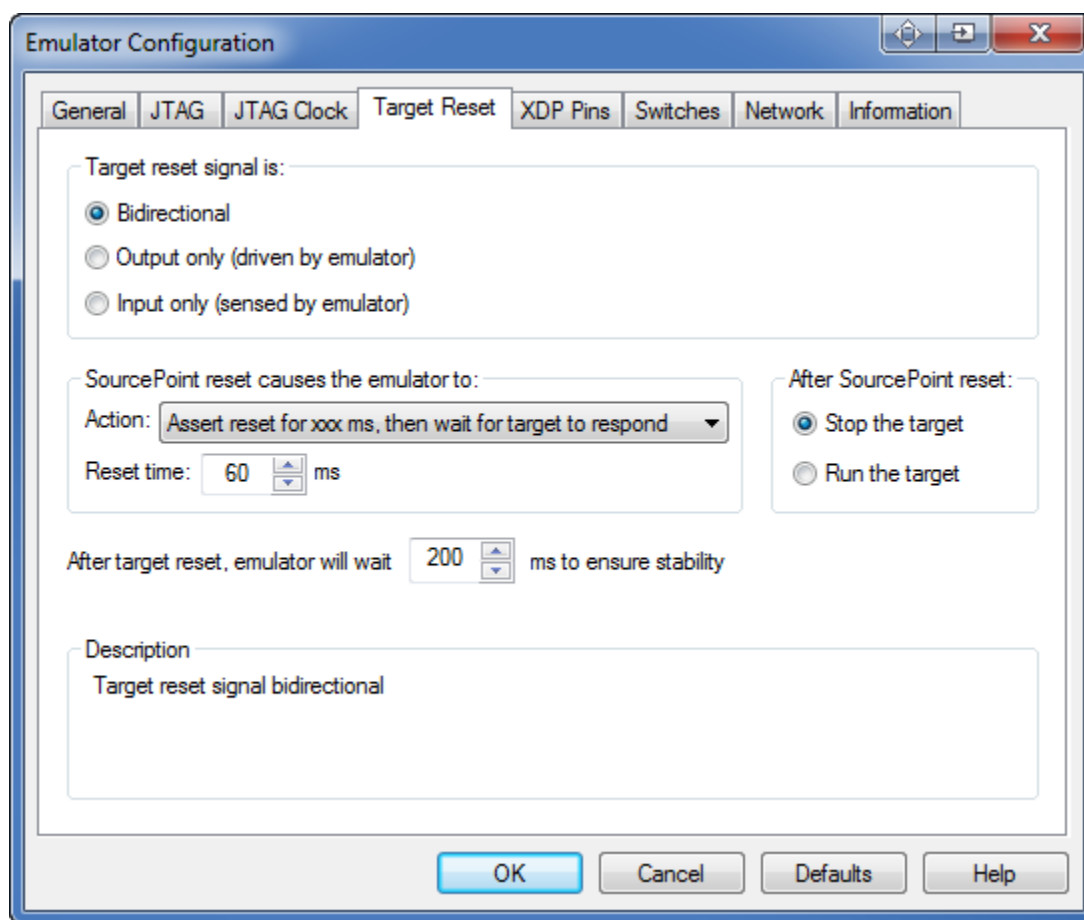
Internal field: JTAG clock rate. This field allows you to specify a clock rate from among the choices provided in the text drop down box.

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.

JTAG timeout. Specifies how long the emulator waits for a response from the target when shifting commands or data on the JTAG chain.

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.

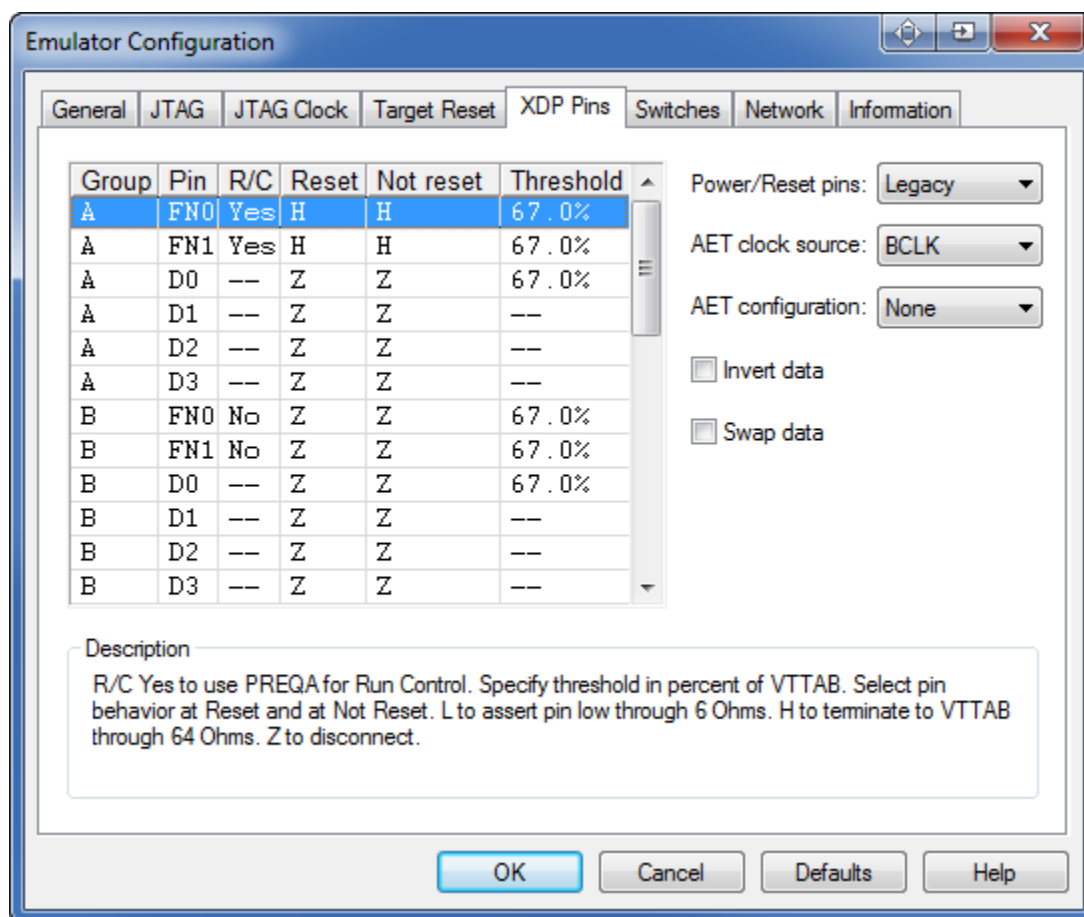
- **Action.** There are four options: **Wait on manual (external) target reset, Assert reset until target responds, Assert reset for xx ms, then wait for target to respond,** or **Assert reset for xx ms, don't wait for target to respond.**
- **Reset time.** Allows you to choose the length of time you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

XDP Pins Tab

This tab displays when you are connected to an ECM-XDP3 or LX-1000 emulator.

***XDP Pins dialog box***

These advanced settings control the hardware-level connection to the target. The target configuration file for a specific target will select the correct settings for this tab. The pins themselves are split into four identical groups (Obs A, B, C, and D), and the Hook pins.

The columns in the pins control identifies each pin's Group and Pin name and allows low level setup. See the Description field for more details.

Power/Reset pins. The Power/Reset pins setting tells the emulator which pins on the debug connector carry the power and reset signals to and from the target.

Legacy - Most server, desktop, and mobile targets require this setting.

3-wire Pwrmode - Intel uses this pinout for some System-On-a-Chip targets aimed at the embedded market.

AET clock source. Select which signal(s) to use as a clock for AET trace.

AET configuration. Select the width and number of AET trace channels emitted by the processor.

A - Use Obs Data A0-A3 as a single 4-bit channel

A/B - Use Obs Data A0-A3 and B0-B3 as 4-bit channels

AB - Use Obs Data A0-A7 as a single 8-bit channel

A/B/C - Use Use Obs Data A0-A3, B0-B3, and C0-C3 as 4-bit channels

A/B/C/D - Use Obs Data A0-A3, B0-B3, C0-C3, and D0-D3 as 4-bit channels

AB/CD - Use Obs Data A0-A3 with B0-B3 as one 8-bit channel and C0-C3 with D0-D3 as another 8-bit channel

ABCD - Use all 16 Obs data pins as a single 16-bit channel.

Invert Data. Interpret low level on Obs data pins a logic 1.

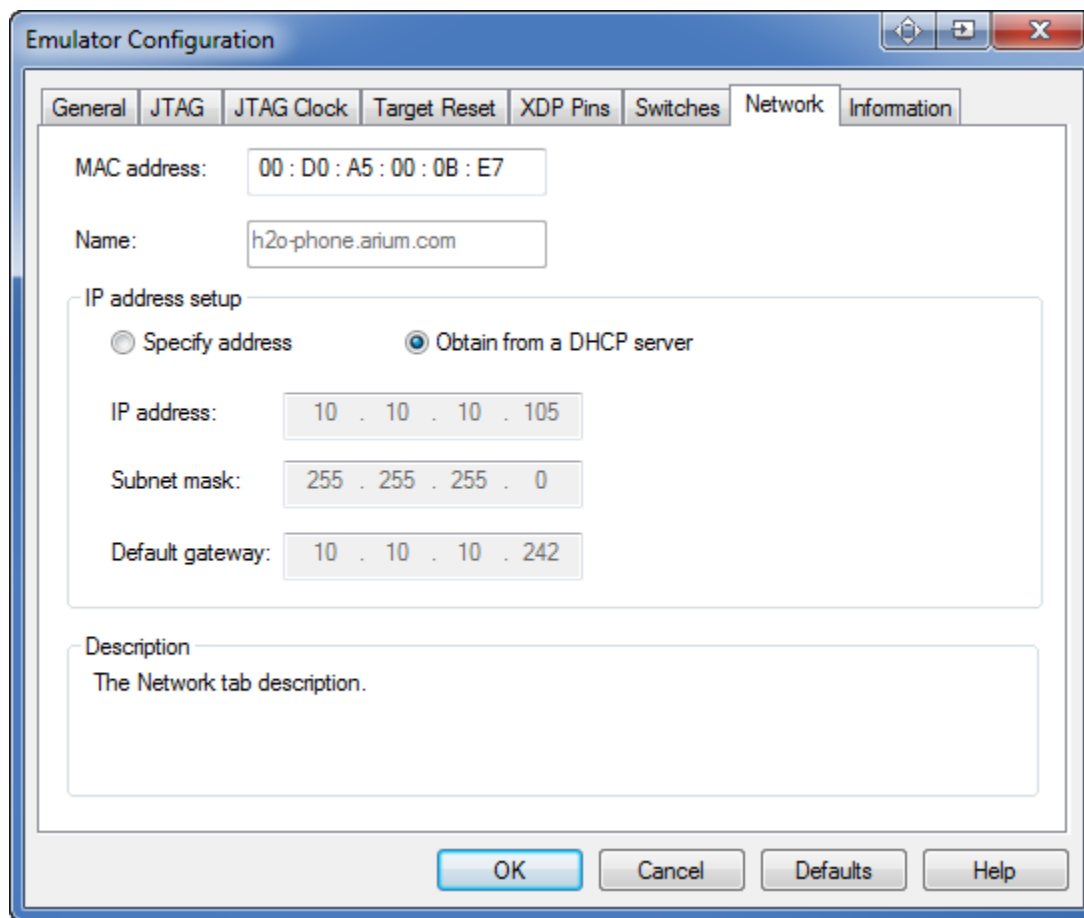
Swap data. Reverse the bytes from first to last in the trace frame. (The last byte becomes the first byte.)

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 image shows the 'Emulator Configuration' dialog box with the 'Network' tab selected. The 'General' tab is also visible. The 'Network' tab contains the following fields and options:

- MAC address:** 00 : D0 : A5 : 00 : 0B : E7
- Name:** h2o-phone.arium.com
- IP address setup:**
 - ☐ Specify address
 - ☒ Obtain from a DHCP server
- IP address:** 10 . 10 . 10 . 105
- Subnet mask:** 255 . 255 . 255 . 0
- Default gateway:** 10 . 10 . 10 . 242
- Description:** The Network tab description.

At the bottom of the dialog box 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.

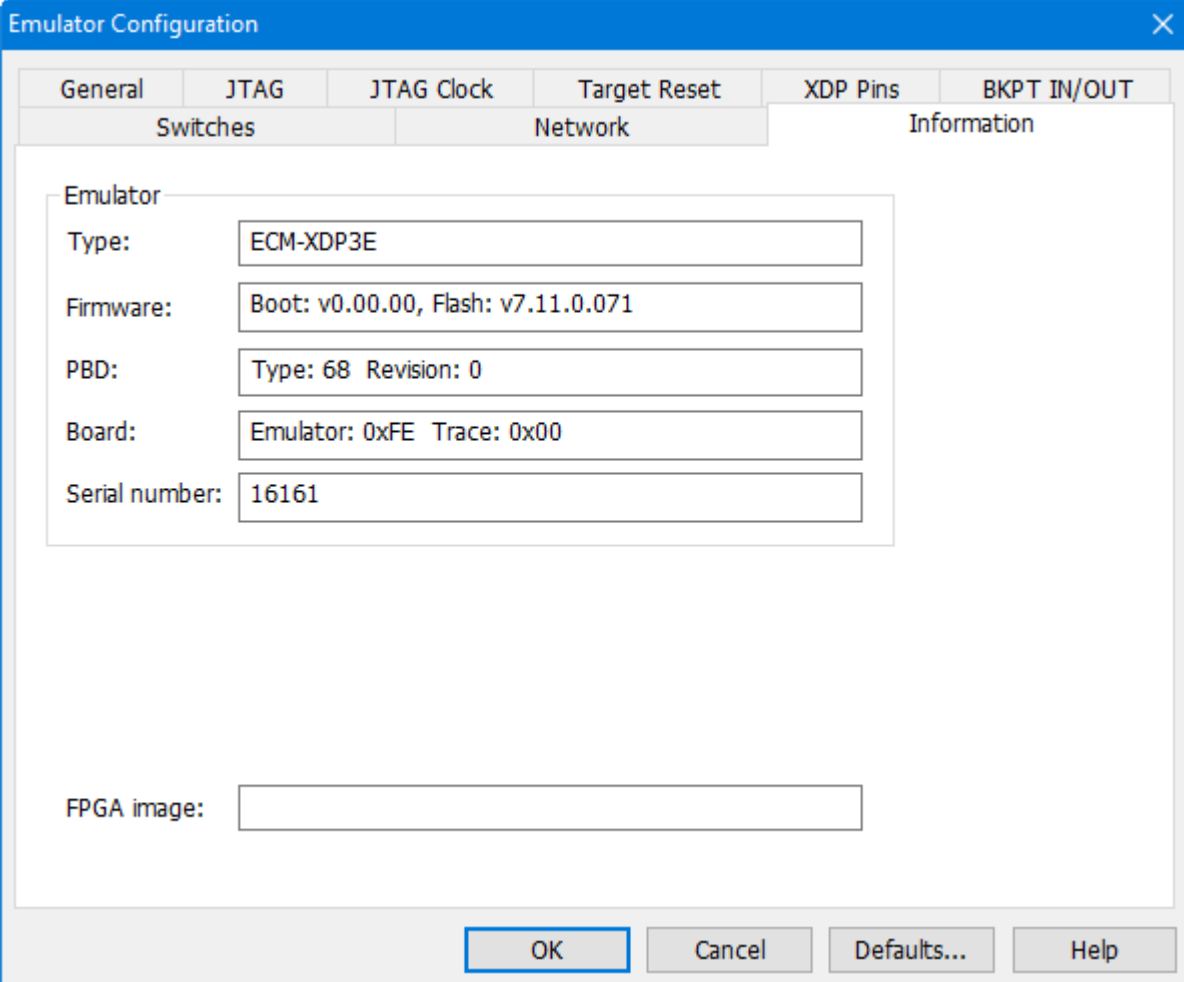
Type. This field gives you the name of the emulator to which you are attached.

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

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

Serial Number. This field gives you the serial number of your emulator.



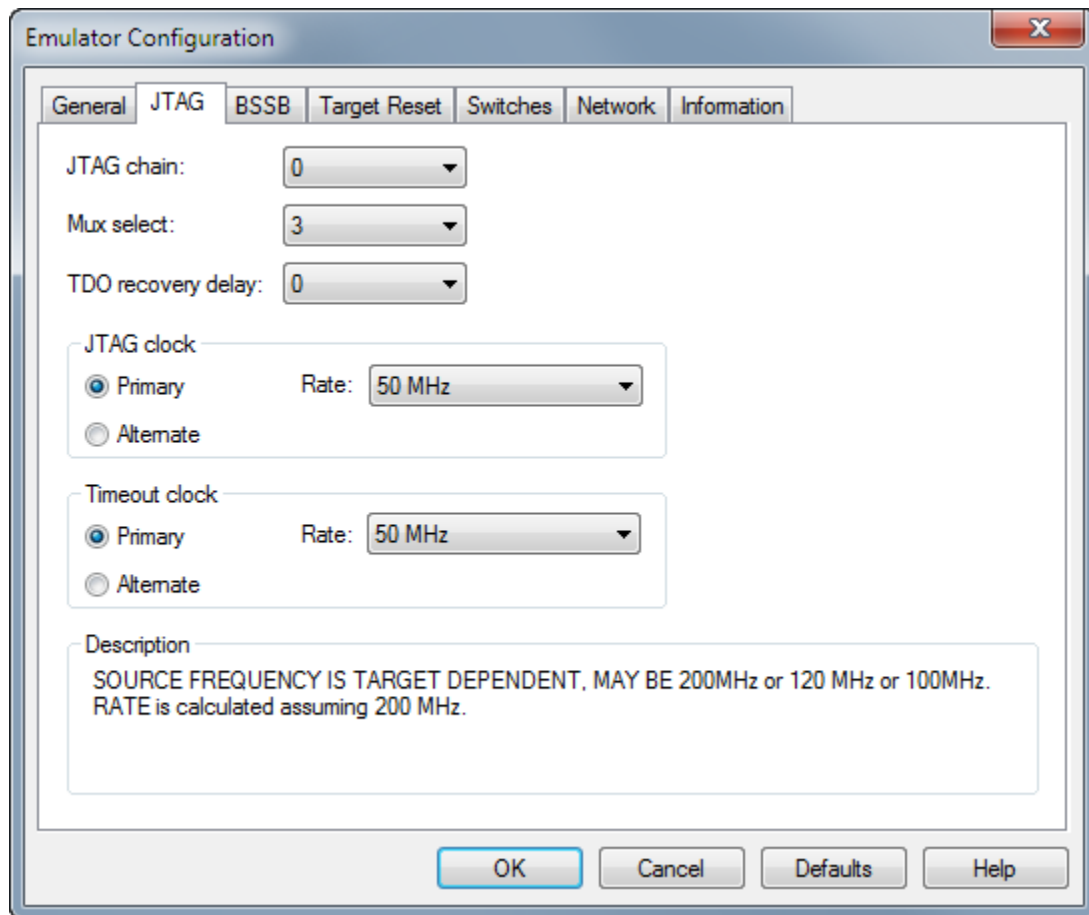
The image shows the 'Emulator Configuration' dialog box with the 'Information' tab selected. The dialog has a blue title bar and a close button. It contains several tabs: 'General', 'JTAG', 'JTAG Clock', 'Target Reset', 'XDP Pins', and 'BKPT IN/OUT'. The 'Information' tab is active, showing a 'Switches' sub-tab. The 'Emulator' section contains five read-only fields: 'Type' (ECM-XDP3E), 'Firmware' (Boot: v0.00.00, Flash: v7.11.0.071), 'PBD' (Type: 68 Revision: 0), 'Board' (Emulator: 0xFE Trace: 0x00), and 'Serial number' (16161). There is also an 'FPGA image' field at the bottom. The bottom of the dialog has 'OK', 'Cancel', 'Defaults...', and 'Help' buttons.

General	JTAG	JTAG Clock	Target Reset	XDP Pins	BKPT IN/OUT
Switches		Network		Information	
<p>Emulator</p> <p>Type: ECM-XDP3E</p> <p>Firmware: Boot: v0.00.00, Flash: v7.11.0.071</p> <p>PBD: Type: 68 Revision: 0</p> <p>Board: Emulator: 0xFE Trace: 0x00</p> <p>Serial number: 16161</p> <p>FPGA image:</p>					
<p>OK Cancel Defaults... Help</p>					

Information Dialog Box

JTAG (DCI) Tab

The JTAG (DCI) tab lets you change options associated with the JTAG scan chain. It is only displayed when connected via DCI (DbC)



JTAG Dialog Box

JTAG chain: Selects the target JTAG chain.

Mux select: Selects the JTAG chain configuration. The possible settings are:

- 0: JIO to CL
- 1: EXI to CL
- 2: EXI to JIO (COM)
- 3: EXI to JIO (SHR)
- 4: EXI to CL to JIO (COM)
- 5: EXI to CL to JIO (SHR)
- 6: EXI to PRI on 2nd
- 7: EXI to SEC on 2nd
- 8: EXI to PRI on 3rd JTAG
- 9: EXI to SEC on 3rd
- 10-15: Reserved.

Where:

COM = "Common JTAG Chain Board Config."

SHR = "Shared JTAG Chain Board Config."
EXI = "EXI-JTAG"
CL = "CLTAP"
JIO = "JTAG-IO"
PRI = "Primary JTAG scan-chain"
SEC = "Secondary JTAG scan-chain"
2nd = "second JTAG debug port"
3rd = "third JTAG debug port".

TDO recovery delay: Sets the number of half-TCK periods to wait before sampling TDO. The possible settings are:

- 0: ½ TCK period (recommended for TCK up to 8 MHz)
- 1: 1 TCK period (8 to 33 MHz TCK)
- 2: 1½ TCK periods (33 MHz to 50 MHz TCK)
- 3: 2 TCK periods (> 50 MHz TCK).

WARNING: Changing this setting may adversely affect JTAG communications.

JTAG clock:

Primary: Source frequency is target dependent. It may be 200 MHz, 120 MHz or 100 MHz. Rate is calculated assuming 200 MHz.

Alternate: Uses BSSB clock. Rate is calculated based upon selected BSSB frequency.

Rate: TCK frequency choices vary depending on whether primary or alternate is selected.

Timeout clock:

Primary: Source frequency is target dependent. It may be 200 MHz, 120 MHz or 100 MHz. Rate is calculated assuming 200 MHz.

Alternate: Uses BSSB clock. Rate is calculated based upon selected BSSB frequency.

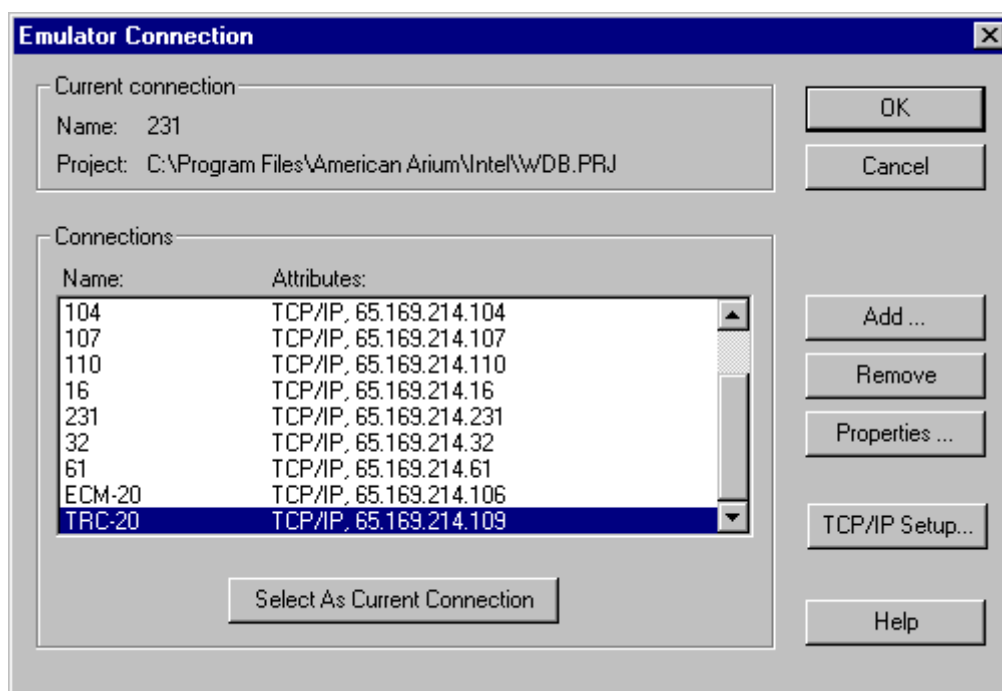
Rate: Timeout period is proportional to inverse of rate from selected source.

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), USB or DCI.

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

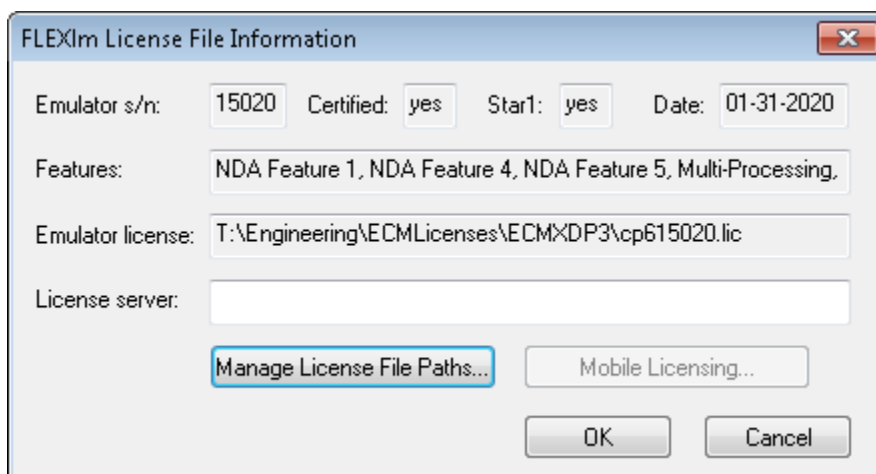
Select **Help|Index** on the menu bar to display an alphabetical list of help topics and related information.

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 the **License File** menu item to display information about the current license file. There are two types of licenses, Perpetual and Subscription. See [SourcePoint Licensing](#) for more information.



FLEXlm Licence File Information dialog

Emulator s/n. Displays the currently connected emulator's serial number. Emulator license files allow SourcePoint to communicate to a particular emulator (by serial number).

Certified: Indicates whether a valid license file was found.

Star1: Indicates whether a valid Star1 service contract is in effect. This field is only valid when a Perpetual license is in use.

Date: If a Perpetual license is in use, indicates the Star1 expiration date. If a Subscription license is in use, indicates the subscription expiration date.

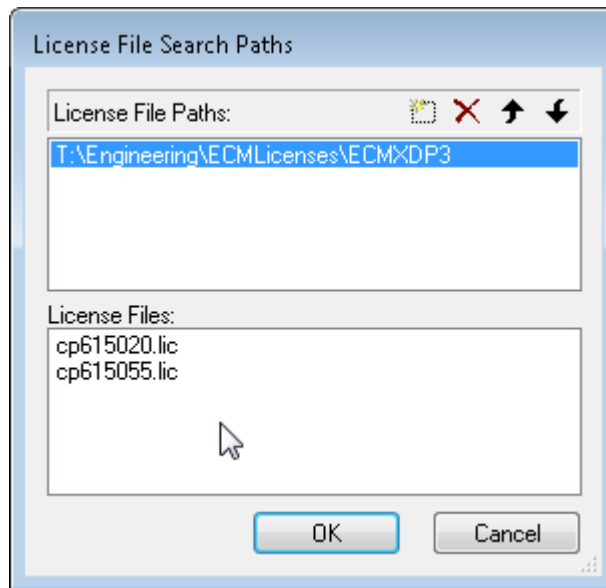
Features: Displays any additional licensed features. These are features that were purchased separately from the emulator and SourcePoint.

Emulator License: Displays the emulator license file that is in use. Emulator license files allow SourcePoint to communicate to a particular emulator (by serial number).

License Server: Used to specify the location of SourcePoint license file server (port@servername). This field is only required when a Subscription license is in use. The location of the license file server can also be specified by adding a key to the registry:

HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ASSET InterTech Inc.\SourcePoint\LicenseServer, Type = REG_SZ, Data = location of server.

Manage License File Paths. Press this button to open the License File Search Paths dialog. This dialog is used to specify where SourcePoint will look for emulator license files. Any changes to the search paths take effect the next time SourcePoint is started.



License File Search Paths Tab

Mobile Licensing. Allows a license to be borrowed from the SourcePoint license file server (typically for offsite usage of a laptop). This button is only enabled when a Subscription license is in use.

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

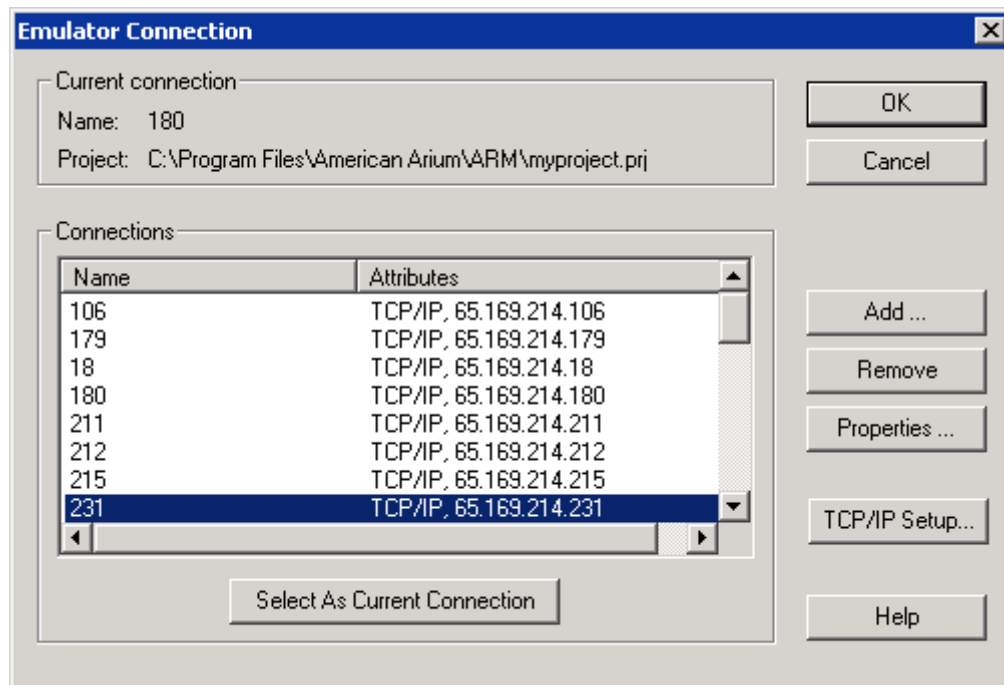
How to Add Emulator Connections

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), you can add emulator connections at any time. Three basic types of connections are described below: USB, TCP/IP, and DCI..

USB Connections

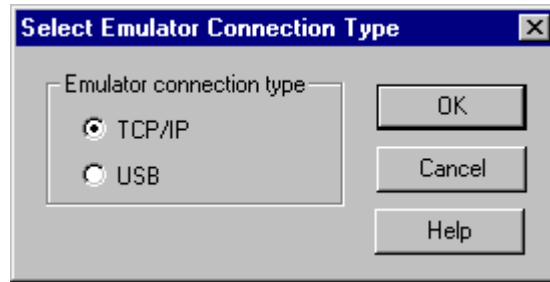
If you are adding a USB connection (assuming you currently have a TCP/IP connection):

1. Attach the USB cable. (This can be hot plugged; you do not need to disconnect your TCP/IP cable.) A standard Microsoft® Windows® hardware installation wizard appear asking you to install the USB driver.
2. Click on the **Browse** button in this dialog box and browse to your SourcePoint working directory.
3. Click on "AriumUsb.inf"
4. Complete the Microsoft installation wizard.
5. Select **Options|Emulator Connection** from the toolbar menu. The **Emulator Connection** dialog box appears, and the USB connection automatically displays in the connection list.
6. Select the USB connection from the list.
7. Click on the button labeled **Select As Current Connection** or double click the connection.
8. Click the **OK** button.



Emulator Connection dialog box

9. Click the **Add** button. The **Select Emulator Connection Type** dialog box appears.



Select Emulator Connection Type dialog box

10. Enable **USB**.
11. Click the **OK** button. The **Emulator USB Connection Properties** dialog box opens with all the text fields filled in.
12. Click the **OK** button. The dialog box closes, and the **Emulator Connection** dialog box displays again.
13. Double click the now highlighted connection entry in the **Connections** text box or click on 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** section.

DCI Connections

A DCI connection connects the host computer running SourcePoint directly to the target (without an emulator). Refer to [Getting Started with DCI DbC](#) for more information.

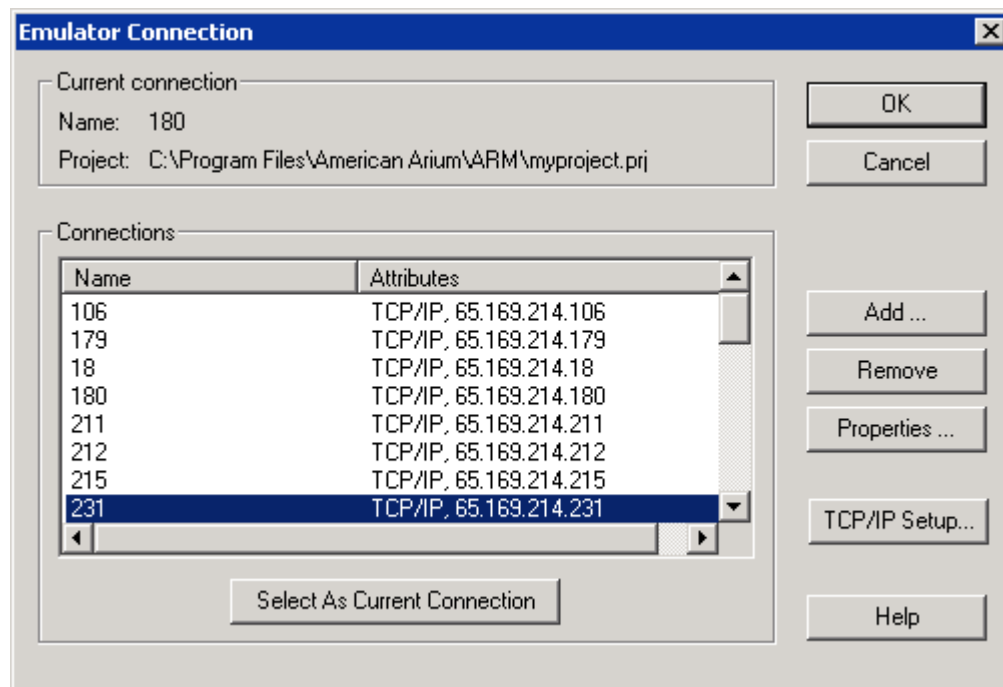
TCP/IP Connections

If you are adding a TCP/IP connection:

1. If necessary, change the hardware cable. A non-network direct connection requires the orange cable. A network connection requires the blue cable.

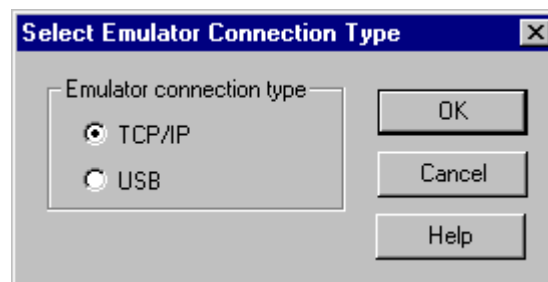
For a direct (non-network) connection between host computer and emulator, a crossover cable is required. (An orange crossover cable is included with new emulators). To connect the emulator to a network, a direct cable is required. (A blue direct cable is included with new emulators.)

2. Go to **Options|Emulator Connection** on the menu bar. The **Emulator Connection** dialog box opens with at least one IP address in the **Connections** section.



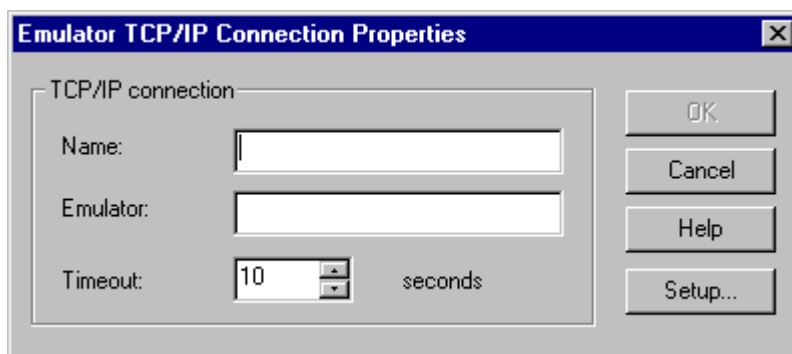
***Emulator Connection** dialog box*

- Press the **Add** button. The **Select Emulator Connection Type** dialog box displays.
- Select **TCP/IP**.



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

- 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 connections, use IP address 192.168.000.001
 - For network IP connections, use the address given to you by your Network Administrator.
 - **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 with the new emulator connection information 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 in the **Current Connection** section.

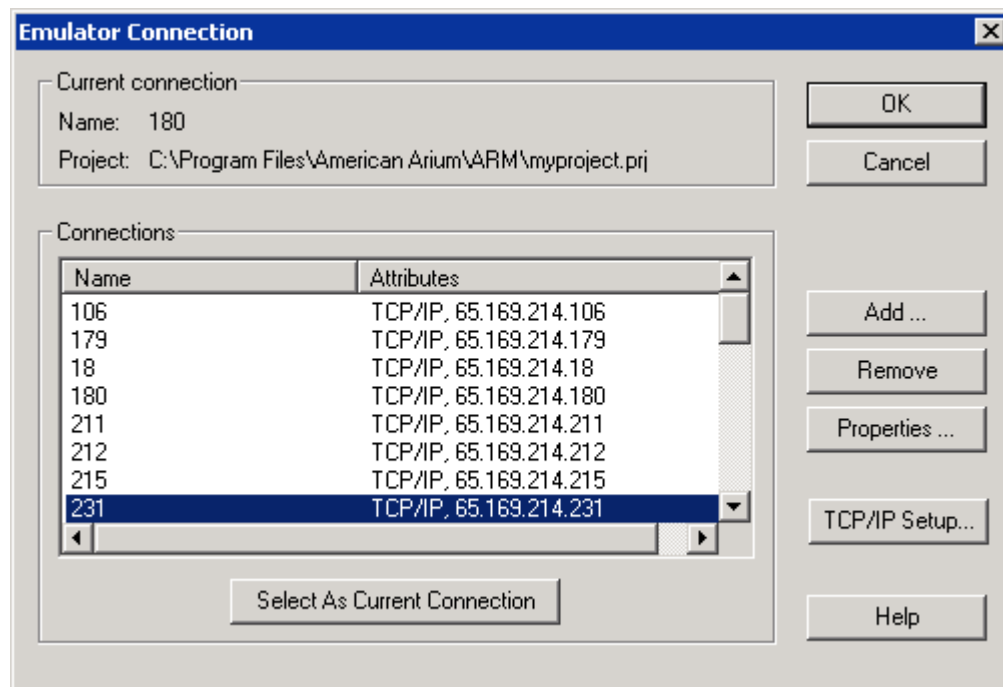
Using Microsoft Windows 2000/2003/2008 DDNS for Addressing Emulators by Hostname

If your network includes a Microsoft Windows 2000/2003/2008 server that provides DHCP and DDNS services, 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.

Note: This procedure works only if the DDNS and DHCP services on your Microsoft Windows 2000/003/2008 Server are configured appropriately.

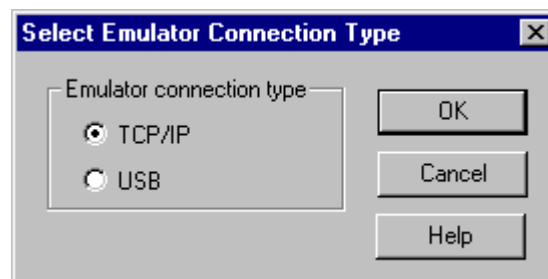
To add a dynamic TCP/IP connection by using a hostname:

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. Go to **Options|Emulator Connection** on the menu bar. The **Emulator Connection** dialog box displays.



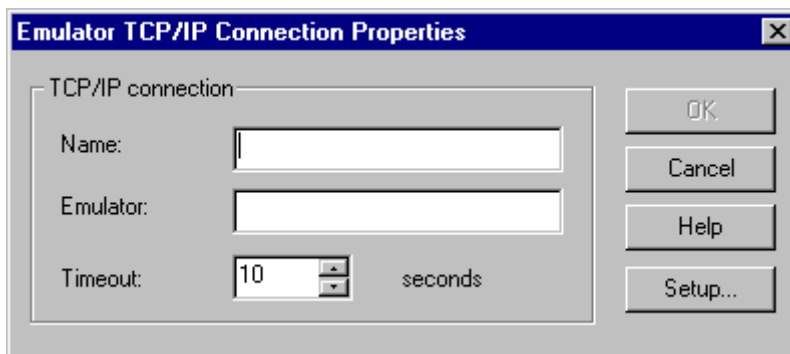
***Emulator Connection** dialog box*

- Press the **Add** button. The **Select Emulator Connection Type** dialog box displays.
- Select **TCP/IP**.



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

- Click the **OK** button. The **Emulator TCP/IP Connection Properties** dialog box displays.



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*).

Depending on your network configuration, you may need either to register a DNS suffix on the workstation where SourcePoint is installed (see "Registering a DNS Suffix" below), or specify the emulator name as a Fully Qualified Domain Name (FQDN), e.g., *ecm-5123.abc.net*. See your Network Administrator for more information.

- If using a standard hostname, type "ecm-**<serial number>**". without quotation marks and inserting the actual serial number of the emulator in place of the italicized words shown (i.e., *ecm-5123*).
 - **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.

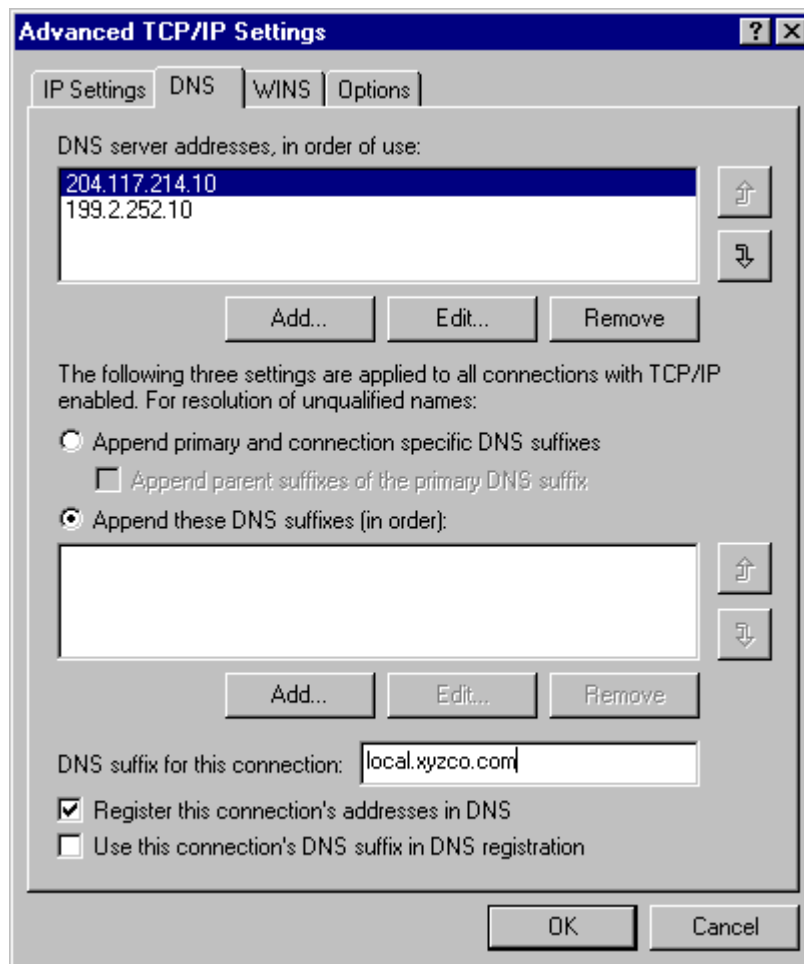
The name of the current connection is displayed at the top of the **Emulator Connection** dialog box under **Current Connection**.

Registering a DNS Suffix

If you want to set up an emulator TCP/IP connection based on DNS name instead of IP address, you may need to set up a DNS suffix on the workstation where SourcePoint is installed.

1. Get the DNS suffix from your Systems Administrator.

2. Go to your Control Panel and double click on **Network and Dial-up Connections**.
3. Double click on **Local Area Connections**. The **Local Area Connections Status** dialog box displays.
4. On the **General** tab, click the **Properties** button. The **Local Area Connection Properties** dialog box displays.
5. In the **This connection uses the following items field**, double-click on the **Internet Protocol (TCP/IP)** option.
6. Click the **Properties** button. The **Internet Protocol (TCP/IP) Properties** dialog box displays.
7. Click the **Advanced** button. The **Advanced TCP/IP Settings** dialog box displays.
8. Click on the DNS tab.
9. Do one of the following:
 - To resolve an unqualified name by appending the primary DNS suffix and the DNS suffix of each connection (if configured), enable **Append primary and connection specific DNS suffixes**. If you also want to search the parent suffixes of the primary DNS suffix up to the second-level domain, enable **Append parent suffixes of the primary DNS suffix**.
 - To resolve an unqualified name by appending the suffixes from the list of configured suffixes, enable **Append these DNS suffixes (in order)**, and then click the **Add** button to add suffixes to the list.
 - To configure a connection-specific DNS suffix, key in the DNS suffix in the **DNS suffix for this connection** text box.
10. Click the OK button.



Advanced TCP/IP Settings dialog box showing the DNS tab

9. In the **DNS suffix for this connection** text box, type the domain suffix supplied by your Systems Administrator.
10. Click the **OK** button.

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 text box in the **User defined macros** section of the dialog box, select a 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. 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.

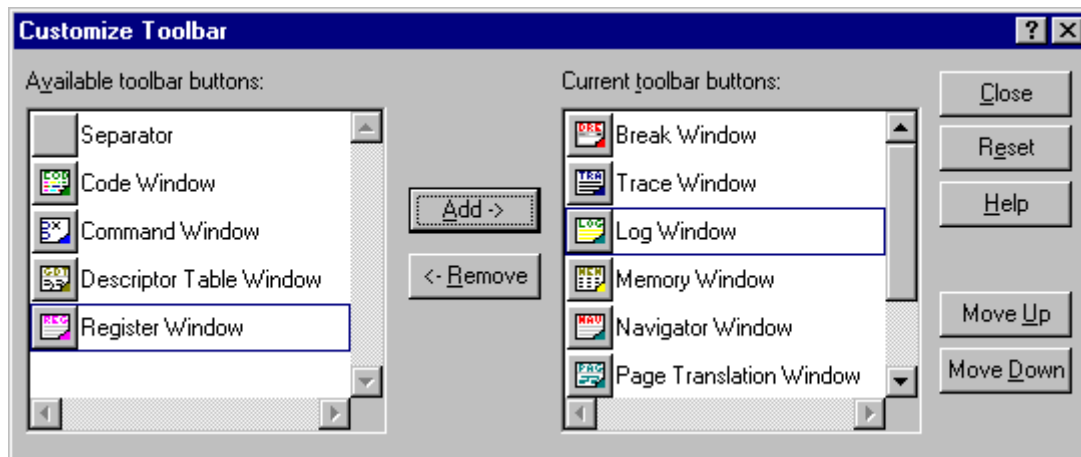
Display Text Next to a Group of Icons

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.



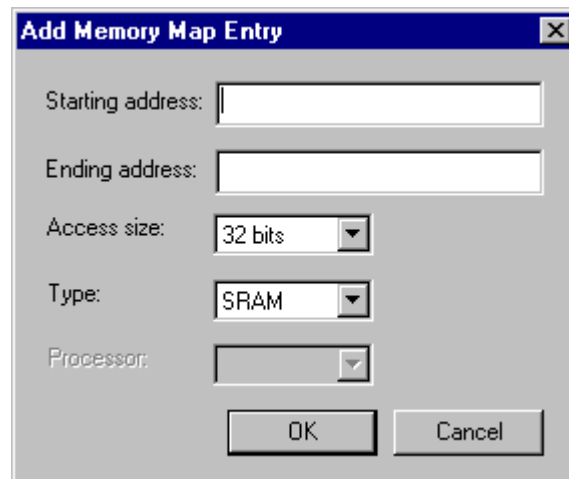
Customize Toolbar window

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 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 Start SourcePoint With Command Line Arguments

Command Line Arguments

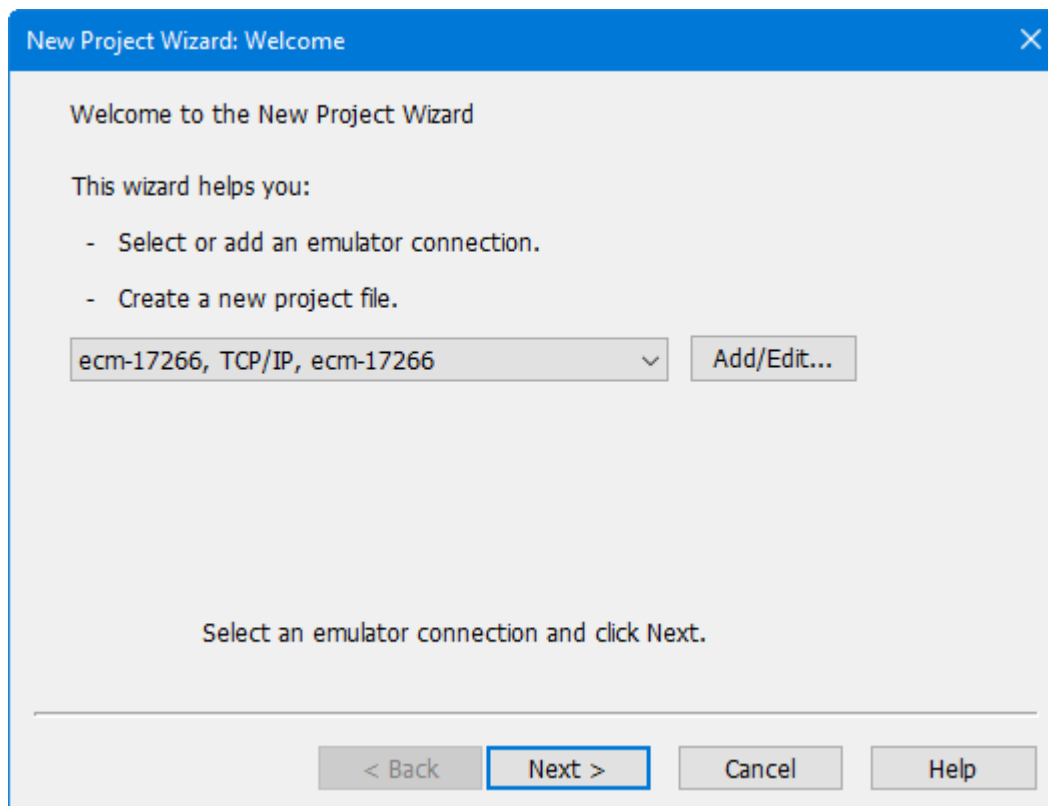
The following command line arguments are available with SourcePoint

-ini file.ini	Use file.ini rather than the default sp.ini
-p file.prj	Load the project file file.prj
-m file.mac	Run the macro file.mac
-mc	Enable multicluster support
-?	Display the SourcePoint command line options
-safe	Start SourcePoint in safe mode. Memory accesses to areas mapped DRAM are restricted
-itp	Causes SourcePoint to run in ITP compatible mode (same functionality as the itpcompatible control variable.

How to Use the New Project Wizard

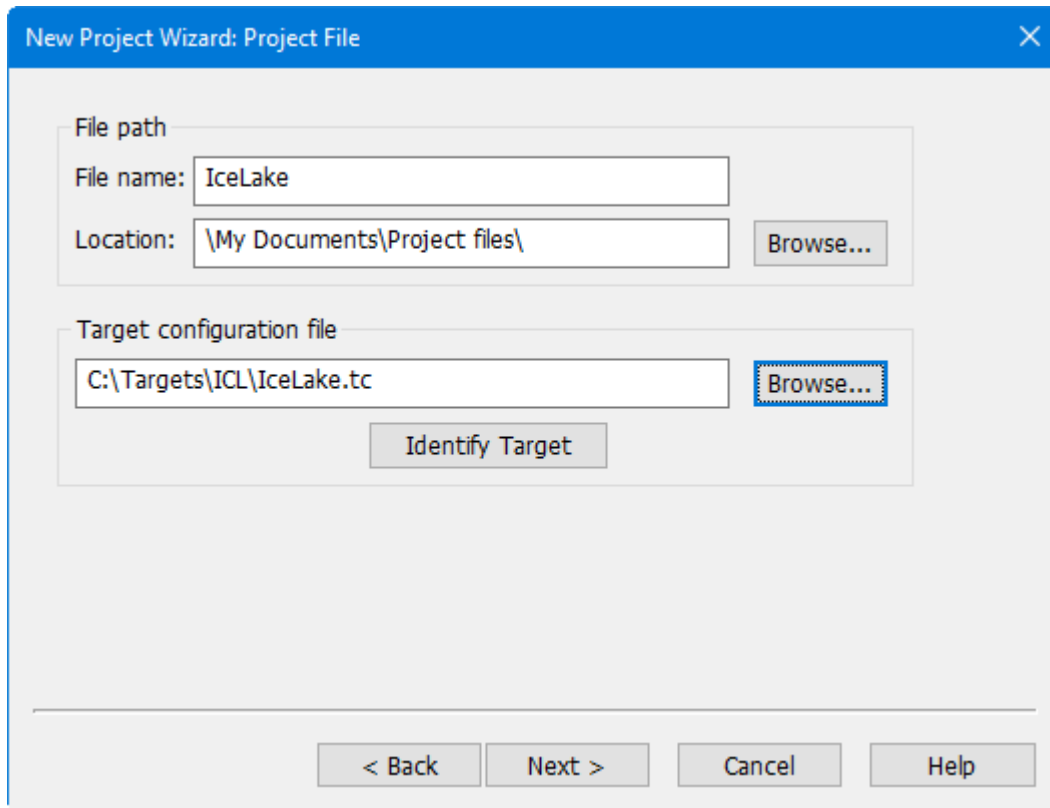
Select File | Project | New Project to open the wizard. The New Project Wizard: Welcome screen displays.

Use the existing emulator connection, or select Add/Edit to create a new emulator connection (USB or TCP/IP).



New Project Wizard: Welcome dialog box.

Click the Next button. The New Project Wizard: Project File screen displays:



The image shows a 'New Project Wizard: Project File' dialog box. It has a blue title bar with a close button. The main area is divided into two sections. The first section, 'File path', contains a 'File name:' field with 'IceLake' and a 'Location:' field with '\My Documents\Project files\'. There is a 'Browse...' button next to the location field. The second section, 'Target configuration file', contains a text field with 'C:\Targets\ICL\IceLake.tc' and a 'Browse...' button. Below these sections is an 'Identify Target' button. At the bottom of the dialog are four buttons: '< Back', 'Next >', 'Cancel', and 'Help'.

New Project Wizard: Project File dialog box

File Path Section

Filename. Enter the name of the project file. Note: This field is filled in automatically if a target configuration file is specified in the next section.

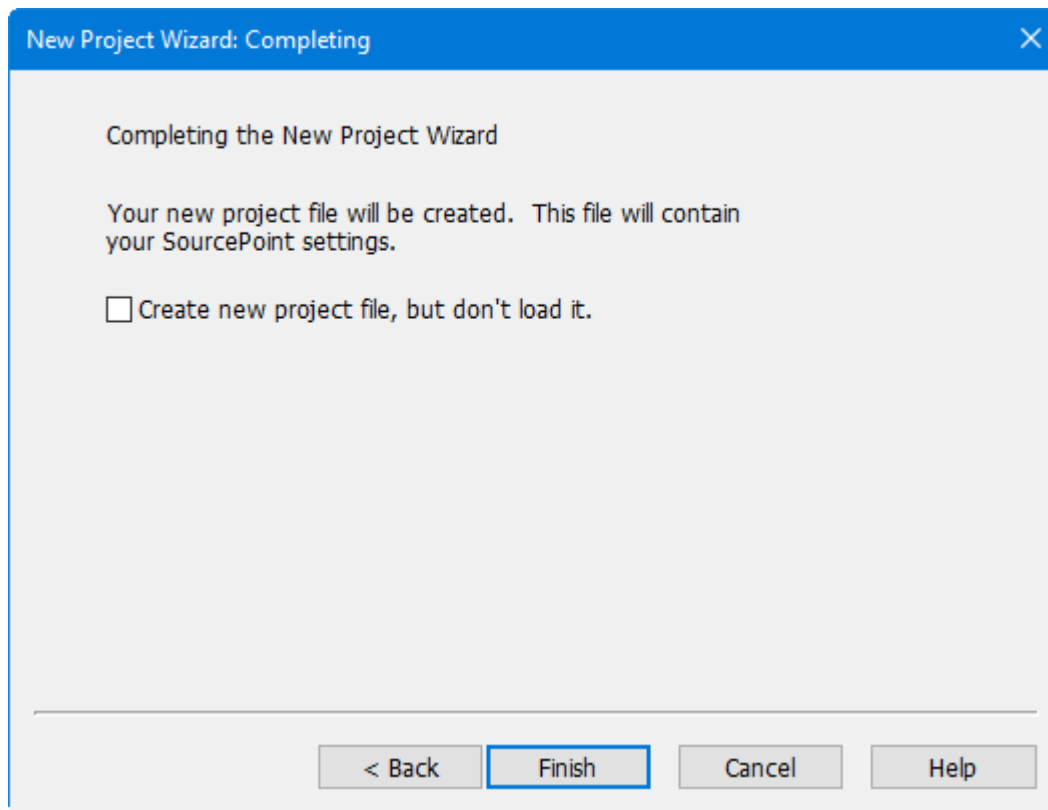
Location. Enter the location for the project file, or press the Browse button to navigate to the location. Note: This field is filled in automatically if a target configuration file is specified in the next section.

Target Configuration File Section

Target Configuration File. Enter the name of a target configuration file, or press the Browse button to navigate and select one.

Identify Target. If you're not sure which target configuration file to use, the Identify Target button can be used to help select the best file. This feature works for most newer processors. It will also check if there is a newer version of the target configuration file available online.

Click the Next button. The New Project Wizard: Completing screen displays:



New Project Wizard: Completing dialog box

To exit the New Project Wizard and open the new project file, select Finish. If you wish to create the new project file without automatically loading it, use the checkbox above.

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

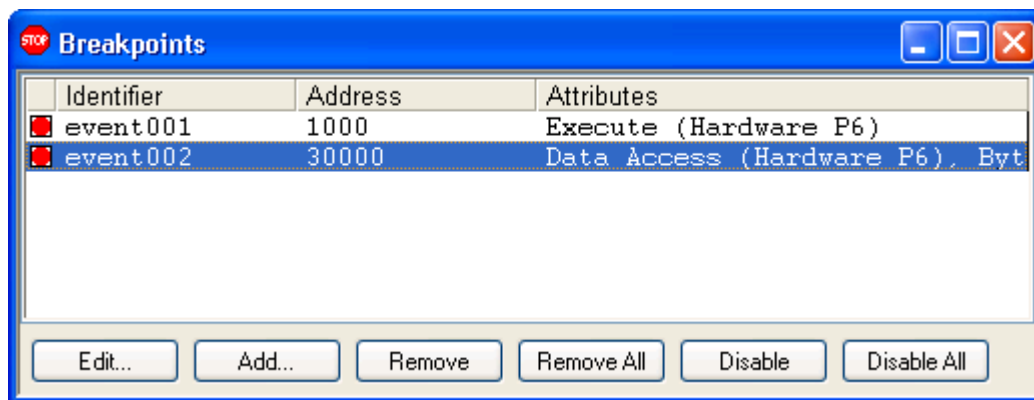
Breakpoints Window Introduction

The Breakpoints window is the central location in SourcePoint for managing breakpoints.

The Breakpoints window is not the only window where breakpoints can be defined. Breakpoints may also be defined in the Code window, the Trace window, the Symbols window, etc. For more information, see [Set Breakpoints From Other SourcePoint Windows](#).

Breakpoints Window

To open the Breakpoints window, select View|Breakpoints on the menu bar or click on the Breakpoints icon on the toolbar.



Breakpoints Window

Breakpoints Window - Breakpoint List Section

The Breakpoints window displays the Breakpoint list, a list of currently defined breakpoints. This includes hardware (processor) breakpoints, software (memory) breakpoints and task debugging breakpoints.

Note: Double clicking in a column heading will sort the Breakpoint list in ascending order by that column. Double clicking the column heading a second time will sort the list in descending order.

Breakpoint List Columns

Enable / Disable (unlabeled). Displays the type of a breakpoint and whether it is enabled. The type of breakpoint is indicated by its icon and color. The same icons used in the Breakpoints window are also displayed in the Breakpoint column in the Code window.

	Breakpoint Resource Type
	Software
	Hardware
	Processor



Breakpoint Icons

Clicking in this column enables or disables a breakpoint. The enable/disable state is indicated by the icon being solid or just an outline.

Identifier. Displays the symbolic name for a breakpoint.

Address. Displays the address of a breakpoint.

Attributes. Displays the other attributes of a breakpoint. The attributes displayed vary by the breakpoint type. A long list of attributes clipped by the right-hand side of the window is available in the tooltip for that field.

Breakpoint List Button Bar

The buttons below the Breakpoint list provide quick shortcuts for managing breakpoints. All of the button actions are also available on the context menu. To reduce the size of the Breakpoints window, the button bar can be hidden by selecting Hide Buttons on the context menu.

Edit. Opens the Edit Breakpoint dialog with the currently selected breakpoint displayed (double left-clicking on the breakpoint performs the same action). See [Add/Edit Dialog](#) for more information about editing breakpoints.

Add. Opens the Add Breakpoint dialog to define a new breakpoint. See [Add/Edit Dialog](#) for more information about adding breakpoints.

If an existing breakpoint is selected when Add is pressed, the dialog will be initialized with that breakpoint's settings. This is useful when creating a breakpoint similar to one already defined.

Remove. Removes the currently selected breakpoint.

Remove All. Removes all breakpoints in the list.

Enable / Disable. Toggles the state of the currently selected breakpoint.

Disable All. Disables all breakpoints in the list.

The number of hardware breakpoints available varies based on processor type. The number of software breakpoints available is always 512.

Breakpoint List Context Menu

The Breakpoint list context menu is available by right clicking anywhere in the Breakpoint list.

Edit. Provides the same functionality as the Edit button (above).

Add. Provides the same functionality as the Add button (above).

Remove. Provides the same functionality as the Remove button (above).

Enable / Disable. Provides the same functionality as the Enable / Disable button (above).

Open Code. Opens a Code window displaying the location of the currently selected breakpoint. This button is only enabled on execution breakpoints.

Open Memory. Opens a Memory window displaying the location of the currently selected breakpoint. This button is only enabled on execution and data breakpoints.

Remove All. Removes all breakpoints from the Breakpoint list.

Enable All. Enables all breakpoints in the Breakpoint list.

Disable All. Disables all breakpoints in the Breakpoint list.

Hide / Show Buttons. Hides or shows the Breakpoint list button bar. Hiding the buttons makes the Breakpoints windows smaller, but requires that all of the button actions be performed via the context menu.

Load. Loads a group of breakpoints from a file. Provides a quick way to switch between different breakpoint environments.

Save. Saves a group of breakpoints to a file. The file saved has a .brk extension by default.

Resources. Opens the Breakpoint Resources dialog. This read-only dialog displays the number of hardware and software breakpoints available and currently in use.

Add/Edit Dialog

Add / Edit Breakpoint Dialog

The Add / Edit Breakpoint dialog opens whenever an add or edit action is requested. The fields in the dialog vary based on the breakpoint resource (Hardware, Software, etc), and also based on the type of breakpoint selected in the Break on field.

The following is a list of the possible breakpoint fields:

Identifier. Displays a user-defined name for the breakpoint. If this field is left blank, SourcePoint will automatically create a name based on the following rules:

1. If a symbolic name was entered in the Address field, then the Identifier field will be set to the symbolic name, and the Address field will be changed to the numeric address value.
2. If a numeric address was entered in the Address field, then the Identifier field will be set to event #, where # is a unique number.

Break On. Defines the breakpoint type. The breakpoint type and resource are closely related. If you select a new breakpoint type that is not available for a particular breakpoint resource, then the resource type will change automatically. For more on breakpoint types and resources see [Breakpoint Types and Resources](#).

Resource. Defines the breakpoint resource (Hardware, Software, etc.). The choices in this dropdown list vary based on the breakpoint type selected in the Break on field above. If only one breakpoint resource is available for a given breakpoint type, then that resource is automatically selected.

Processor. In multiprocessor systems, selects the processor associated with the breakpoint. For task breakpoints, this field is labeled Task, and contains the name of the task the breakpoint is set in.

Address. Defines the breakpoint address. Both symbolic address expressions and numeric addresses are supported.

If a symbolic address expression is entered without specifying a breakpoint identifier (name), then the symbolic expression is copied to the Identifier field, and the numeric address is entered in the Address field.

Find Symbol Button. Opens the Find Symbol dialog. This dialog allows quickly locating any program symbol and its memory address.

For detailed information on the Find Symbol dialog box, see the [Edit Menu](#) topic under "SourcePoint Overview," part of SourcePoint Environment.

Binary (1010) Buttons. Binary buttons are available for the Address and Data fields. These buttons open a dialog that allows editing a field value in binary and, if the breakpoint type supports it, specifying "Don't Cares" for individual bits.

Translate. Specifies the address translation type of the breakpoint. This controls when virtual addresses are translated to physical addresses. This control is only enabled for processors that have an MMU, and only for breakpoint resource types where a physical address is specified (e.g., software breakpoints).

When Translate Once is selected, the virtual address is translated when the OK button is pressed to dismiss the dialog.

When Translate Every Go is selected, the virtual address is re-translated prior to every Go operation.

Cmd/macro. Specifies a macro to run when the breakpoint hits. There are two ways to specify a macro:

1. Use the Browse button to find a macro file to be executed.
2. Enter a single command to execute prefaced by the '#' character.

This field is only available for hardware execution breakpoints, software breakpoints, task breakpoints and power cycle breakpoints.

Breakpoint Types and Resources

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

Breakpoint Type	Break Resource
Data Access	Hardware
Data Access in SMM	Hardware
Data Write	Hardware
Data Write in SMM	Hardware
Execute	Hardware Software
Execute in SMM	Hardware
I/O Access	Hardware
I/O Access in SMM	Hardware
Reset	Emulator
Init	Emulator
SMM Entry	Processor
SMM Exit	Processor
Power Cycle	Processor
BKPT IN	Emulator
Machine Check	Processor

Hardware (Debug Register) Breakpoints

Hardware breakpoints rely on processor-specific registers to recognize events, such as instruction execution or data reads/writes at a memory or I/O address. Hardware 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 hardware breakpoints. Data values are not part of a breakpoint condition.

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

Note: Hardware 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.

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

The Reset breakpoint uses a signal on the debug port to detect entry and exit from the reset state. When the exit from reset state is detected the emulator will halt the target.

The Init breakpoint stops the target when the processor encounters an Init event.

The BKPT IN breakpoint utilizes an input signal on the emulator itself to allow stopping the target via an external trigger signal. There may be a delay, or “slide”, from the BKPT IN signal edge to the time the target is stopped by the emulator.

Processor Breakpoints

SMM Entry and SMM Exit breakpoints stop the target when the processor is entering or exiting System Management Mode, respectively.

The Power Cycle breakpoint stops the target when it detects the target has gone from the power off state to the power on state.

The Machine Check breakpoint stops the target when a Machine Check exception occurs on the target.

How To - Breakpoints

Set Breakpoints From Other SourcePoint Windows

The Breakpoints window is the central location for managing breakpoints in SourcePoint, but there are many shortcuts that provide easy ways to define breakpoints in other Windows.

Code Window

The Code window allows breakpoints to be manipulated from either the Breakpoint column, the context menu, or with keyboard shortcuts.

Breakpoint Column

The Breakpoint column is the blank column at the far left of the Code window. Its primary purpose is to show existing breakpoints, but it can also be used to set and clear breakpoints.

A left click in the Breakpoint column sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab).

A double left click in the Breakpoint column sets an alternate breakpoint type. For instance, if the default code break type is set to hardware, then a software breakpoint will be set. If the default code break type is set to software, then a hardware breakpoint is set. This is convenient when the default code break type is set to software, but a hardware breakpoint needs to be set in ROM or Flash.

A double left click on a breakpoint icon in the Breakpoint column toggles the type from hardware to software or vice-versa.

A left click on a breakpoint icon removes the breakpoint.

Context Menu

Set Breakpoint. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The F9 shortcut key performs the same function.

Clear Breakpoint. Clears an existing breakpoint.

Set Alternate Breakpoint. Performs the same action as a double left click in the Breakpoint column (see above). Pressing Shift+F9 performs the same function.

Toggle Breakpoint Type. Toggles an existing breakpoint from hardware to software or vice versa. Pressing Shift+F9 performs the same function.

Disable Breakpoint. Disables a currently enabled breakpoint. CTRL+F9 performs the same function.

Enable Breakpoint. Enables a currently disabled breakpoint. CTRL+F9 performs the same function.

Add Breakpoint. Opens the Add Breakpoint dialog to add a breakpoint. This is used to add something other than a hardware or software breakpoint (e.g., an emulator breakpoint).

Edit Breakpoint. Opens the Edit Breakpoint dialog to edit the current breakpoint. This is used to change an existing breakpoint.

Go Until Cursor. Sets either a hardware or software breakpoint depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The breakpoint is temporary and is automatically cleared after it hits. The F7 shortcut key performs the same function.

Trace Window

The following context menu items are available:

Set Breakpoint. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The F9 shortcut key performs the same function.

Add Breakpoint. Opens the Add Breakpoint dialog to add a breakpoint. This is used to add something other than a Hardware or Software breakpoint (e.g., an emulator breakpoint).

Symbols Window

The following context menu items are available:

Set Breakpoint. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The F9 shortcut key performs the same function.

Go Until Cursor. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The breakpoint is temporary and is automatically cleared after it hits. The F7 shortcut key performs the same function.

Find Symbol Dialog

The following context menu items are available:

Set Breakpoint. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The F9 shortcut key performs the same function.

Go Until Cursor. Sets either a hardware or software breakpoint, depending on the current default code break setting ([Options|Preference|Breakpoint](#) tab). The breakpoint is temporary and is automatically cleared after it hits. The F7 shortcut key performs the same function.

Command Window

Breakpoints can be set, cleared, enabled and disabled from the Command window and from macro files.

See [dbgbreak commands](#) for setting debug (Hardware) breakpoints.

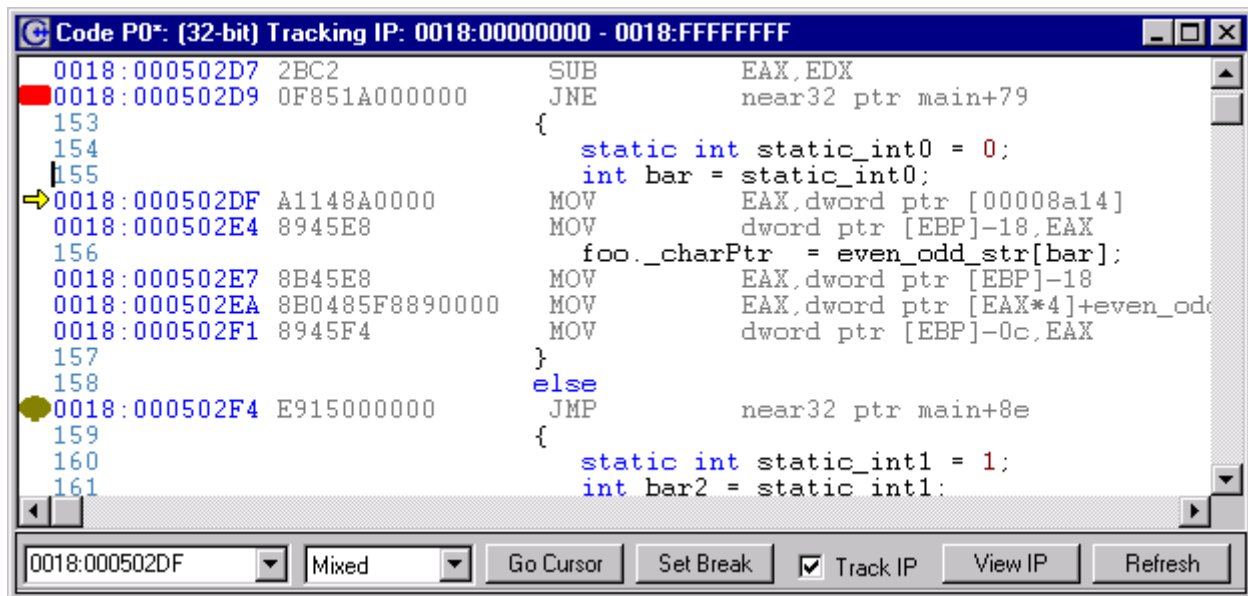
See [softbreak commands](#) for setting software breakpoints.

See [cpubreak commands](#) for setting processor breakpoints.

Code Window

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.

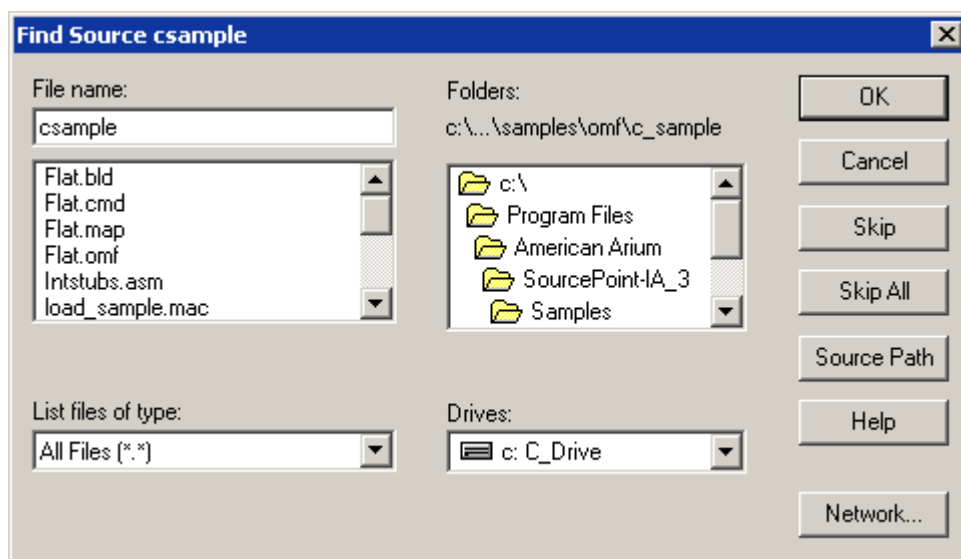
Track IP check box. If **Track IP** is checked, the **Code** window always displays code at the processor instruction pointer (IP). If the processor stops beyond the address range currently visible in the **Code** window, the **Code** window is redrawn showing the code at the new IP. If **Track IP** 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**.

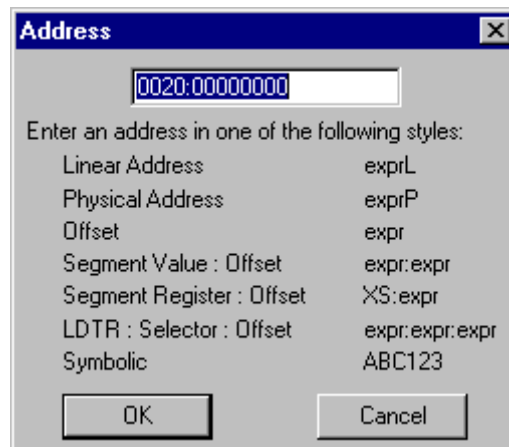
Disassembly Uses menu item. Use this menu item to determine whether you want to view target memory or cached memory associated with disassembly mode. Options are **Target Memory** and **Cached Program**.

- **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.

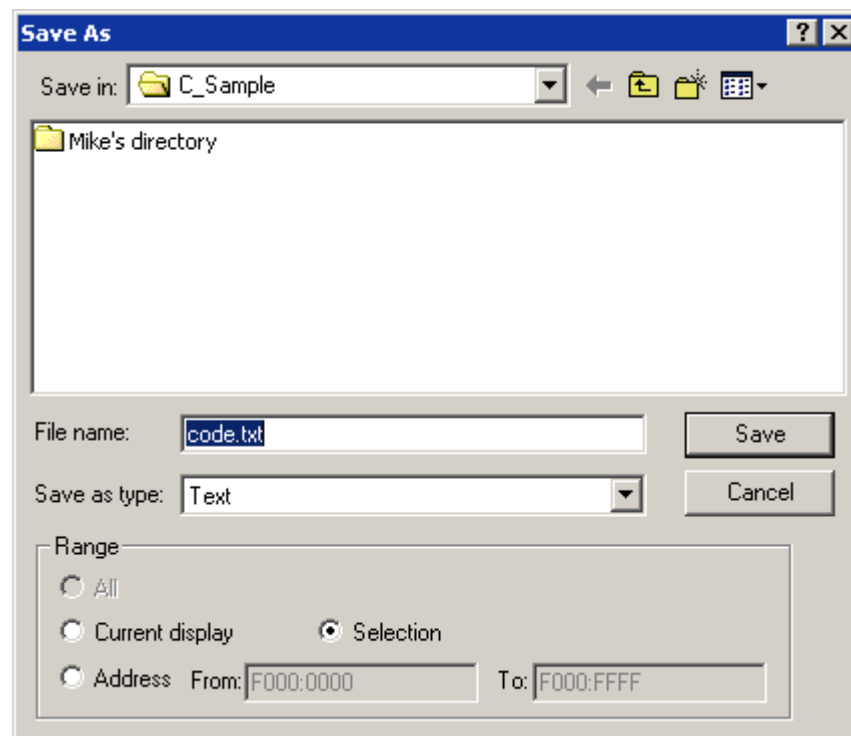
2. Enter a file name with a ".prj" extension
3. Click the **OK** button.

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 Introduction

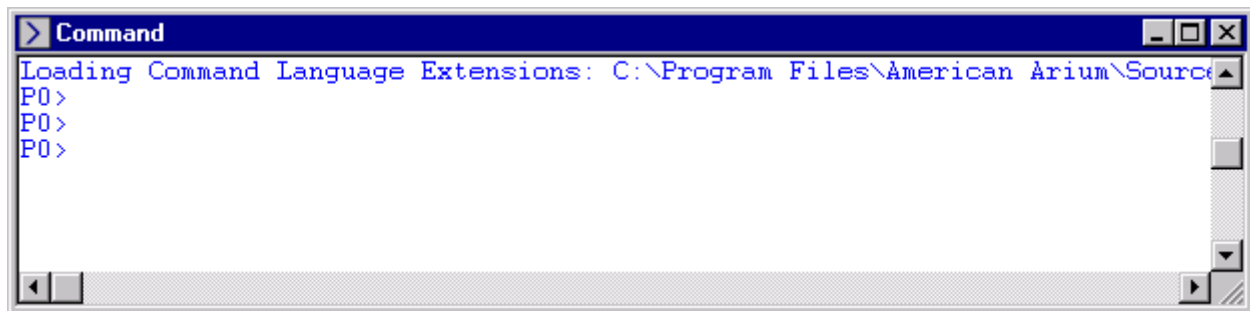
The Command window provides a command line interface to SourcePoint. Commands may be typed one at a time, or multiple commands can be executed from a command file.

The Command window displays a history of previously executed commands and their responses.

This section describes the Command window itself. For a detailed description of the SourcePoint command language, refer to SourcePoint Command Language in the Table of Contents.

The Command Window

To open the Command window, select View | Command on the menu bar or click the Command window icon on the toolbar.



Command window

Entering Commands

Commands are entered at the prompt. In single processor systems the prompt is the '>' character. In multi-processor systems the prompt is the current viewpoint processor name followed by the '>' character (e.g., P0>).

To execute a new command, type the command at the prompt and press Enter. If the command generates a response (e.g., a memory read operation) it will be displayed on the next line.

Commands are colored black, response data is colored blue, and errors are colored red. These colors can be changed by selecting Options | Preferences | Color.

Command History

The Command window displays a history of previously executed commands and their responses. There are two ways to execute a previously executed command:

1. Scroll to the command, left click anywhere in it, and press Enter.
2. While at the command prompt, use the up and down arrow keys to scroll through the command history. Pressing Enter will re-execute the displayed command.

Previously executed commands can also be edited to create new commands. Simply find the command, edit it, and press Enter. Command responses are not editable.

The Command window history can be cleared by selecting Clear Command Window from the context menu.

Editing Commands

Following is a list of keys that can be used for editing or recalling commands.

Up Arrow	At the command prompt moves back one command in the command history. Anywhere else in the window, moves the caret up one line.
Down Arrow	At the command prompt moves ahead one command in the command history. Anywhere else in the window, moves the caret down one line.
Page Up	At the command prompt recalls the oldest command in the command history. Anywhere else in the window, scrolls back one page.
Page Down	At the command prompt recalls the newest command in the command history. Anywhere else in the window, scrolls forward one page.
Right Arrow	Move one character to the right.
Left Arrow	Move one character to the left
Ctrl+Right	Move one word to the right
Ctrl+Left	Move one word to the left
Home	Move to the beginning of the current command
End	Move to the end of the current command
Ctrl+Home	Move to the beginning of the Command window
Ctrl+End	Move to the end of the Command window
Esc	Erases the current command
Enter	Execute the current command
Backspace	Erase the character prior to the caret
Del	Erase the character at the caret
Ctrl+C	Copy the currently selected text to the clipboard
Ctrl+X	Delete the currently selected text and copy it to the clipboard
Ctrl+V	Paste the contents of the clipboard at the caret location
Ctrl+Break	Exit line continuation mode, or cancel a currently running command file

Line Continuation

Line continuation means that a command spans multiple lines. When in line continuation mode the prompt shows '>>' rather than '>'. There are two types of line continuation:

1. When SourcePoint detects that a partial command has been entered, it will automatically enter line continuation mode (e.g., typing if (x), and then pressing Enter). As additional lines are typed, SourcePoint will determine if the command is complete, and then will automatically exit line continuation mode, and execute the command.

2. Typing '\ ' at the end of a line will force SourcePoint into line continuation mode. This is rarely used while typing commands, but is useful for long printf statements in command files.

Line continuation mode can be forced off by pressing Ctrl+Break.

Entering Multiple Commands as a Single Command

Multiple commands can be entered as a single command by using ';' as a delimiter (e.g., stop; ord4 0x1000; go).

Copy / Paste

The Command window supports cut, copy, and paste operations. These operations can be selected from the Edit menu, the Command window context menu, or by pressing Ctrl+X, Ctrl+C or Ctrl+V.

Text can be selected by left clicking and moving the mouse, or by double left clicking to select a word.

Pasting a command into the Command window causes it to be executed immediately. Pasting multiple commands executes all of the commands immediately.

Drag / Drop

The Command window supports two types of drag and drop operations:

1. If a command file is dragged from Windows Explorer, and dropped into the Command window, SourcePoint will execute the command file.
2. If a program file is dragged from Windows Explorer, and dropped into the Command window, SourcePoint will load the file as if the Load command had been used.

Command Files

Command files are text files containing multiple commands. Creating command files helps to automate oft-repeated operations. Command files are also referred to as macro files, script files and include files. There are several ways to execute a command file:

1. Use the [include](#) command in the Command window (see Commands manual).
2. Drag and drop a command file from Windows Explorer to the Command window.
3. Select File | Macro | Load Macro from the main menu.
4. Select File | Macro | Configure Macros to attach a command file to a user-defined toolbar button, and then press the button.
5. Select File | Macro | Configure Macros to attach a command file to an event. Examples of events include: go, stop, project load, power cycle, etc. When the event occurs the macro will automatically execute.
6. Define a breakpoint and specify a command file to execute when the breakpoint hits.

Recently executed macro files are shown in File | Recent Macros. Selecting a command file from this list will re-execute the file. Breakpoint and event macros are excluded from this list.

When a command file is executing, the name of the file is shown in the SourcePoint Status bar (at the bottom of the SourcePoint window).

Aborting a Command File

Press Ctrl+Break to terminate a running command file.

Logging Commands and Responses to a File

Commands and their associated responses can be logged to a file. The [log](#) command begins logging while the [nolog](#) command ends logging. See the Commands manual for more information.

There are two other ways to copy Command window commands and responses to a text file:

1. Use the mouse to select text in the Command window and then paste it into a text editor.
2. Select File | Save As from the main menu to save all or a portion of the Command window to a file.

Printing the Command Window

All or a portion of the Command window can be printed. Select File | Print from the main menu or press Ctrl+P in the Command window.

Searching the Command Window

The Command window supports searching for old commands or response data. Select Edit | Find from the main menu, or press Ctrl+F in the Command window.

Executing an Operating System Command

The Shell command is used to execute commands outside of SourcePoint. There are two ways to use the command:

1. Type Shell without any arguments to open an operating system command window. When the window closes, the focus switches back to the SourcePoint Command window.
2. Type Shell with a command to execute the command and return immediately to SourcePoint.

Refer to [shell](#) in the Commands manual for more information.

Getting Help

There are two ways to access help from the Command window:

1. Type Help or press F1 to open the SourcePoint Help window.
2. Type Help "command name" to open the SourcePoint Help window with the command help topic already displayed.

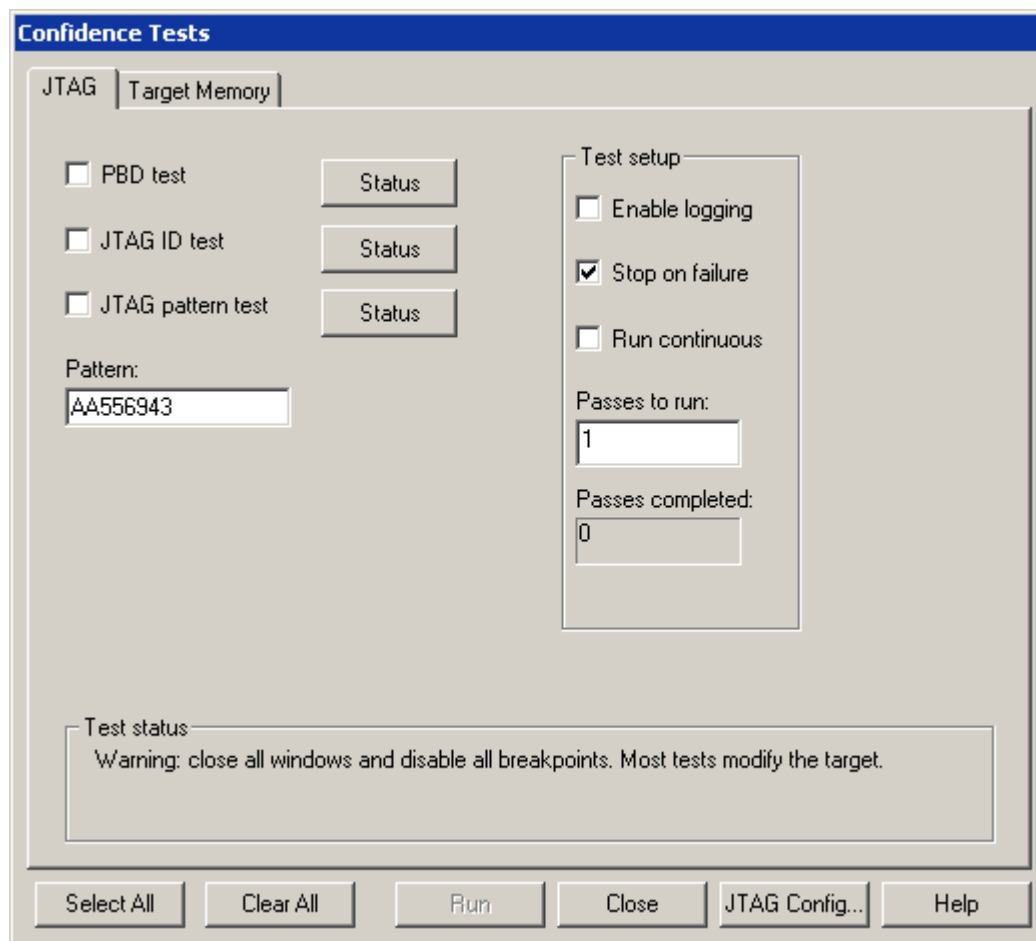
Confidence Tests Window

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 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.

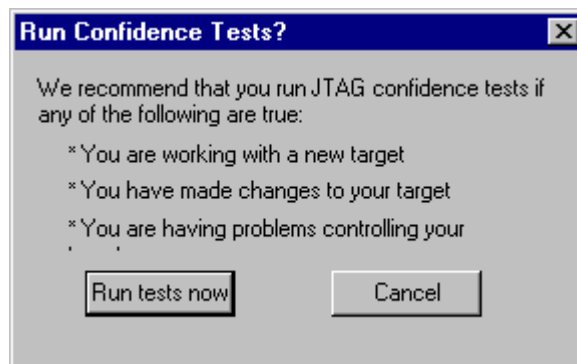
Passes completed. This is not an option but a counter. As the name implies, the text box displays the number of passes the test has completed.

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

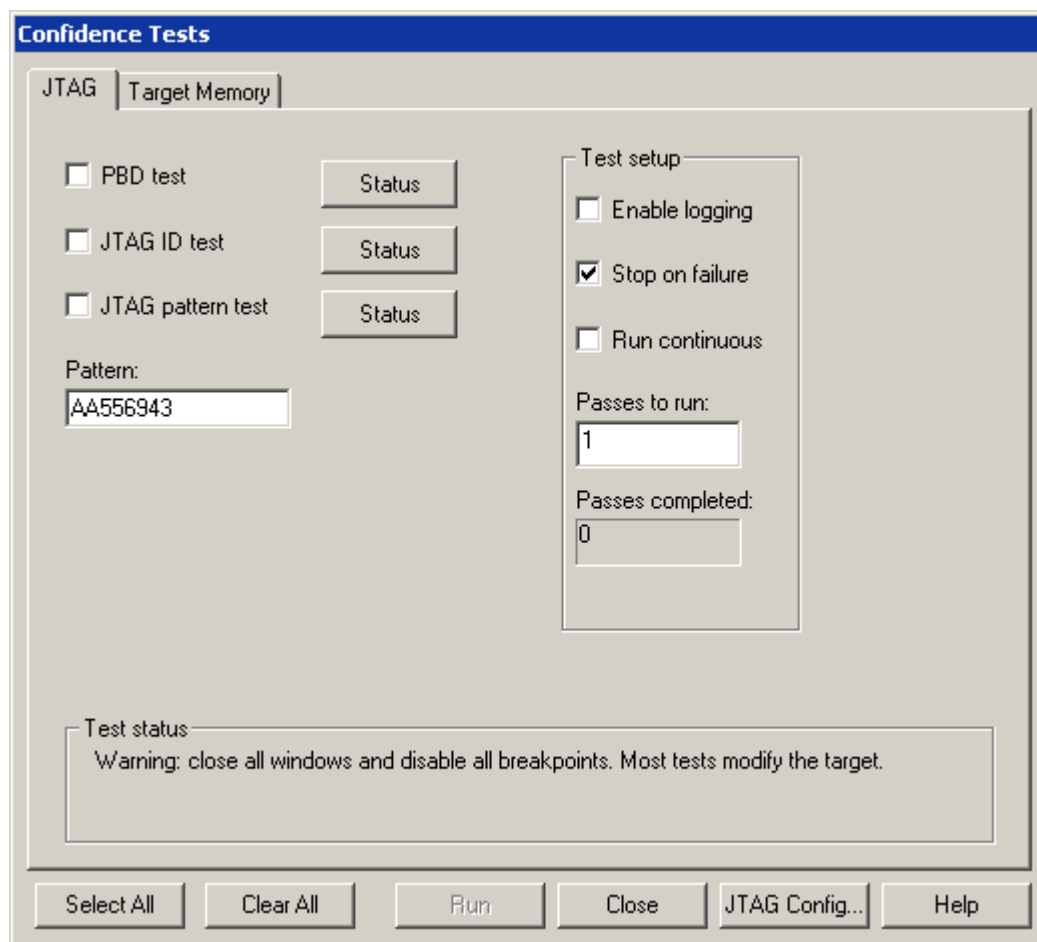
JTAG Tab

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, 'Read target memory' and 'Write target memory', each followed by a 'Status' button.
- Address Range:** Two text boxes labeled 'Start address:' and 'End address:'.
- Write data pattern:** A group box containing three radio buttons: 'Address as data' (selected), 'Checkerboard', and 'Fill with:' followed by a text box.
- Test setup:** A group box containing:
 - Checkboxes for 'Enable logging' and 'Run continuous'.
 - A checked checkbox for 'Stop on failure'.
 - A 'Passes to run:' text box with the value '1'.
 - A 'Passes completed:' text box with the value '0'.
 - A checkbox for 'Reset target first'.
- Test status:** A text area displaying 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

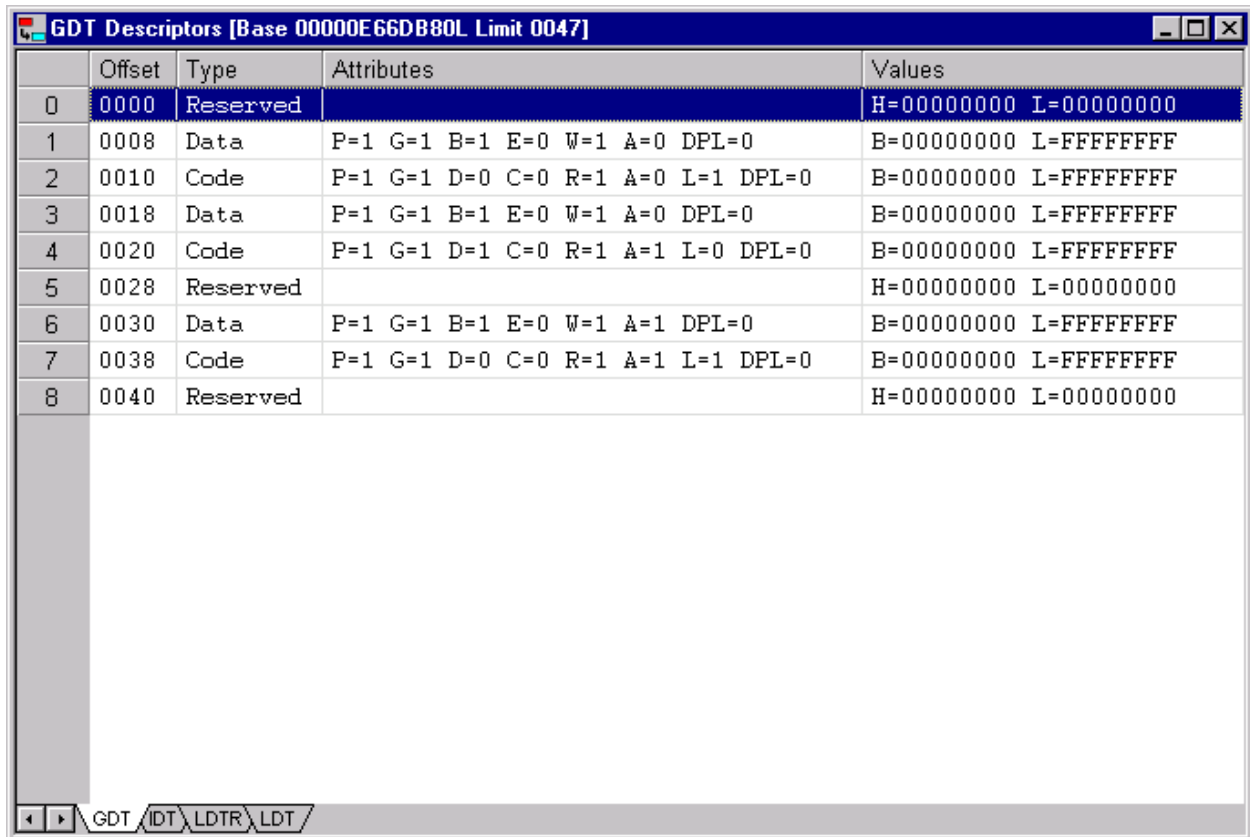
Table of Confidence Test Failures and Symptoms

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 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.



The screenshot shows a window titled "GDT Descriptors [Base 00000E66DB80L Limit 0047]". It contains a table with four columns: Index, Offset, Type, Attributes, and Values. The table lists 9 entries (index 0 to 8). Entries 0, 5, and 8 are "Reserved". Entries 1, 2, 3, 4, 6, and 7 are "Data" or "Code" descriptors with various attributes (P, G, B, E, W, A, DPL) and base/limit values (H and L).

	Offset	Type	Attributes	Values
0	0000	Reserved		H=00000000 L=00000000
1	0008	Data	P=1 G=1 B=1 E=0 W=1 A=0 DPL=0	B=00000000 L=FFFFFFFF
2	0010	Code	P=1 G=1 D=0 C=0 R=1 A=0 L=1 DPL=0	B=00000000 L=FFFFFFFF
3	0018	Data	P=1 G=1 B=1 E=0 W=1 A=0 DPL=0	B=00000000 L=FFFFFFFF
4	0020	Code	P=1 G=1 D=1 C=0 R=1 A=1 L=0 DPL=0	B=00000000 L=FFFFFFFF
5	0028	Reserved		H=00000000 L=00000000
6	0030	Data	P=1 G=1 B=1 E=0 W=1 A=1 DPL=0	B=00000000 L=FFFFFFFF
7	0038	Code	P=1 G=1 D=0 C=0 R=1 A=1 L=1 DPL=0	B=00000000 L=FFFFFFFF
8	0040	Reserved		H=00000000 L=00000000

At the bottom of the window, there are navigation buttons: GDT, IDT, LDTR, and LDT. The "GDT" button is currently selected.

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 (Doubleword) 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.

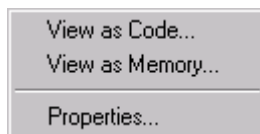
- Select the **GDT** tab to view the status of or modify the contents of a Global Descriptor Table (GDT) entry.
- Select the **IDT** tab to view the status of or modify the contents of an Interrupt Descriptor Table (IDT) entry.

- 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.



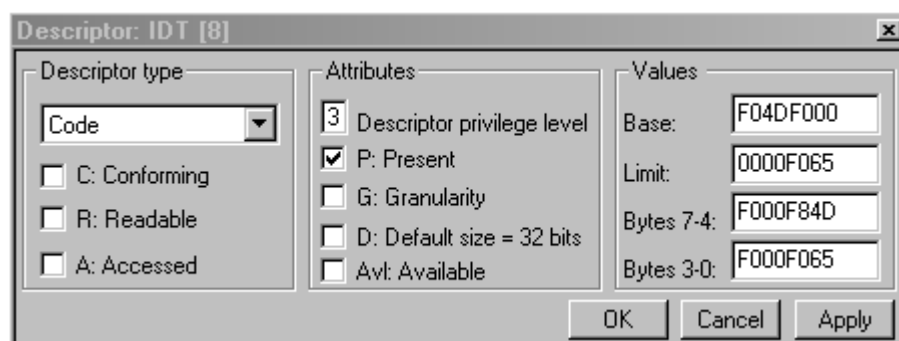
Descriptors menu

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.

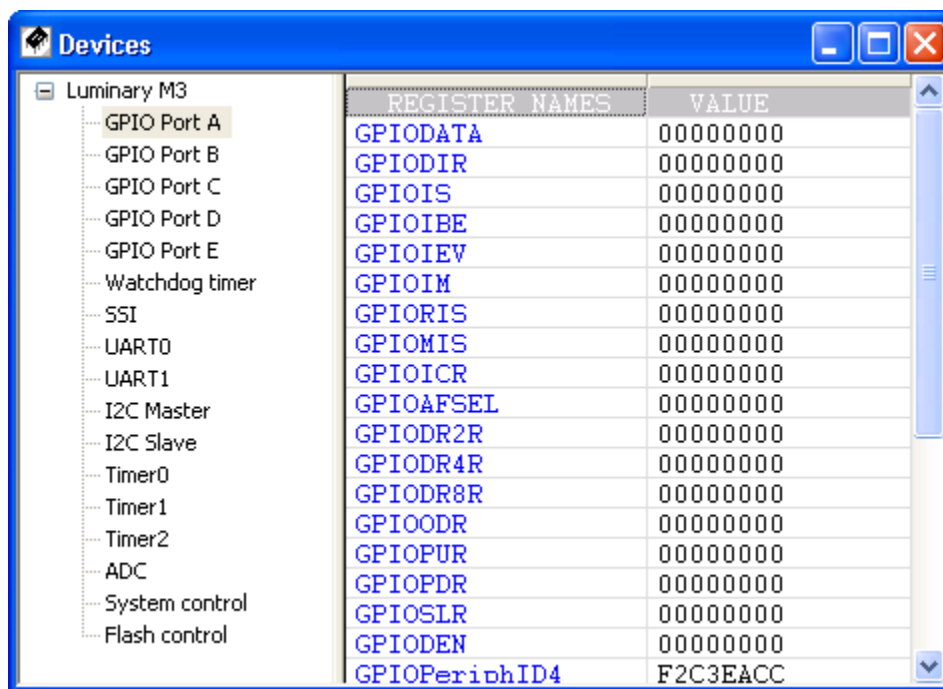
4. In the **Descriptor Type** dialog box, select the check boxes that provide the desired status.
5. In the **Attributes** section of the dialog box, type in the desired **Descriptor privilege level**.
6. Select the check boxes that provide the desired attributes.
7. In the **Values** section of the 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 Introduction

To open the Devices window, select View|Devices or click on the Devices icon on the toolbar. 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 | MSR | MEM | IO | SIO | USER | CHILD}
```

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 and change the contents of a memory location. The memory location is limited to lengths of 1, 2, 4, or 8 bytes.

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. The **Devices** window is not limited to displaying memory. Register cells allow you to display and change the contents of a register.

Syntax:

```
CELL#=<row#>,<column#>,REG,<register name>
```

Example:

```
CELL0=0,1,REG,EIP
```

The above example creates a register cell in the second column of the first row and displays the value of the EIP register in the cell.

MSR cells. MSR cells allow you to display and change the contents of an MSR.

Syntax:

```
CELL#=<row#>,<column#>,MSR,<MSR address>
```

Example:

```
CELL0=0,1,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. I/O cells allow you to display and change the contents of an I/O port.

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.

Indirect I/O cells (IA-32 processors only). Indirect I/O cells allow you to display and change the contents of an indirect I/O location.

When reading or writing a data value, the index value is first written to the port specified by the [IndexPort](#) directive. The data value is then either written to, or read from, the port specified by the [DataPort](#) directive.

Syntax:

```
CELL#=<row#>,<column#>,SIO,<index>
```

Note: An error is generated if an indirect I/O cell is defined with missing IndexPort or DataPort directives.

Example:

```
IndexPort=70h,1
DataPort=71h,1
CELL0=0,0,SIO, 30h
```

The above example creates an indirect I/O cell in the first column of the first row and displays an 8-bit value read from index 30 of indirect I/O port pair 70/71. The value is re-read every time the target stops, or when refresh is selected from the context menu.

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. User-defined cells are read-only.

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 or memory cell (within the same device) and extract a variable number of bits at a particular offset.

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. They are also shown when the parent cell is expanded. 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.

Directives

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. Processors can be specified numerically (e.g., 0, 1, 2, 3), alpha-numerically (e.g., P0, P1, P2, P3), or alpha only (e.g., AHB, APB). If this directive is not specified, then the current viewpoint processor active when the Device view file is 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.

Syntax:

```
RepeatDevice = <device#>
```

Example:

```
[Device1]
Name = Uart1
Base = E0000000
< cell definitions>
[Device2]
Name = Uart2
RepeatDevice = 1          // get 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 **AddSymbols**.

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 the register 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 in within a [Device] section, in which case it applies to only that device.

This directive increases the load time of Device view files.

Syntax:

```
AddSymbols = [true | false]
```

Example:

```
[Device1]
Name = Uart1
AddSymbols = true
Cell10 = 0, 0, text, "ctrlReg"
Cell11 = 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.

IndexPort directive. This directive specifies the index port to be used for indirect I/O cells. 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 in within a [Device] section, in which case it applies to only that device.

Syntax:

```
IndexPort = <port #>, <port size>
```

DataPort directive. This directive specifies the data port to be used for indirect I/O cells. 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 in within a [Device] section, in which case it applies to only that device.

Syntax:

```
DataPort = <port #>, <port size>
```

Example:

```
IndexPort=70h,1
DataPort=71h,1
CELL0=0,0,SIO, 30h
```

The above example creates an indirect I/O cell in the first column of the first row and displays an 8-bit value read from index 30 of indirect I/O port pair 70/71. The value is re-read every time the target stops, or when refresh is selected from the context menu

Enumerations

Enumerations can be defined in a device file and referred to 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"
```

Example:

```
[Enum1]
name="version"
key0=0,"Version 1.0 Rom"
key1=1,"Version 1.1 Rom"
key2=2,"Version 1.2 Rom"
```

These enumerations can be referenced by a cell definition using the enum keyword.

Example:

```
Cell10=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. This can be done by adding NAME="Group Name" into the group section. 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.

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 to a specific cell definition to affect just that cell.

Background Color. Background color can be set using a simple 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 // all cells to green
```

Example:

```
BACK=green // all cells to green
```

Example:

```
Cell10=0,0,text,"Name",back=white // single cell only
```

Text Color. Text color can be set using a simple 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:

```
TEXT=<Hex Value for Red><Hex Value for Green><Hex Value for Blue>
```

or

TEXT=<Color Keyword>

Example:

TEXT=FFFFFF // all cells to white

Example:

TEXT=white // all cells to white

Example:

Cell10=0,0,text,"Name",text=white // single 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 // all cells in a device

Example:

Cell10=0,0,justify,"Name",justify=center // single 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 // 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" // all cells in device
```

Example:

```
cell0=0, MEM, 1000p, 1, FORMAT="%d" // single cell only
```

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 how you can put various syntax options together.

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

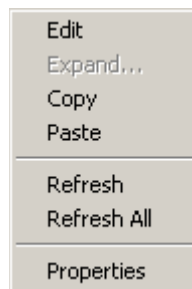
Add Device Menu Item. Prompts for a device view file to load. More than one file can be loaded at a time. Each file can contain one or more device definitions. Currently displayed devices are saved in the project file, so when you exit and restart SourcePoint the devices displayed are remembered.

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 or memory cell into binary. This view also displays any bit fields (child cells) 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 0x40000000 and display 4 bytes of memory read from address 0x40000000.

How To - Devices Window

How to Create a Simple Devices Window

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. Review the information found in "Devices Window," under Devices Overview," part of *Devices Window* for specifics on each type. 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,col,type,string,bkgd color,text color,tooltip
//-----
cell0=0,0,text," REGISTER NAMES ",back=gray,text=white
cell1=0,1,text," VALUE ",back=gray,text=white,

//-----
// Register Names
// format is shown as:
// cell#,row,col,type,string,text color,tooltip
//-----
cell2 =1,0,text,"BOOT_CR",text=blue,tooltip="boot control"
cell3 =2,0,text,"RESET_SR",text=blue,tooltip="reset status"
cell4 =3,0,text,"IDCODE",text=blue,tooltip="identity and ver"
```

```

cell15 =4,0,text,"SRAM0_SR",text=blue,tooltip="SP SRAM0 size"
cell16 =5,0,text,"SRAM1_SR",text=blue,tooltip="SP SRAM1 size"
cell17 =6,0,text,"DPSRAM0_SR",text=blue,tooltip="DP SRAM0 size"
cell18 =7,0,text,"DPSRAM0_LCR",text=blue,tooltip="DP SRAM0 lock"
cell19 =8,0,text,"DPSRAM1_SR",text=blue,tooltip="DP SRAM1 size"
cell110=9,0,text,"DPSRAM1_LCR",text=blue,tooltip="DP SRAM1 lock"

//-----
// Memory Mapped locations for registers listed above
// format is shown as:
// cell#,row,col,type,memory location,access,tooltip
//-----
cell111=1,1,mem,0x7fffc000,4,access=rw,tooltip="base+000H BUS 2"
cell112=2,1,mem,0x7fffc004,4,access=rw,tooltip="base+004H BUS 2"
cell113=3,1,mem,0x7fffc008,4,access=r,tooltip="base+008H BUS 2"
cell114=4,1,mem,0x7fffc020,4,access=r,tooltip="base+020H BUS 2"
cell115=5,1,mem,0x7fffc024,4,access=r,tooltip="base+024H BUS 2"
cell116=6,1,mem,0x7fffc030,4,access=r,tooltip="base+030H BUS 2"
cell117=7,1,mem,0x7fffc034,4,access=rw,tooltip="base+034H BUS 2"
cell118=8,1,mem,0x7fffc038,4,access=r,tooltip="base+038H BUS 2"
cell119=9,1,mem,0x7fffc03C,4,access=rw,tooltip="base+03CH BUS 2"

```

5. Save the file with the extension ".dev".

Loading the Devices File

1. Select the **Devices** window icon, or from the **View** menu select **Devices**. A file dialog displays asking for the file.
2. Select your device.dev file and click on **Open**.

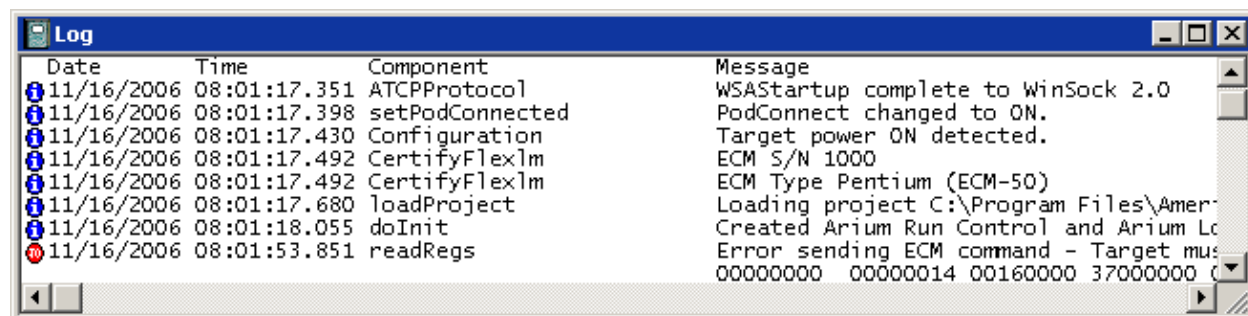
Note: You can add additional views by right clicking in the devices side (left) of the **Devices** window pane.

Sample device files may be found in the \Samples\Devices directory of your SourcePoint installation.

Log Window

Log Window 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.

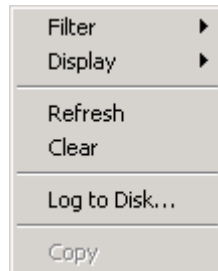
Log Window Icon Definitions

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.

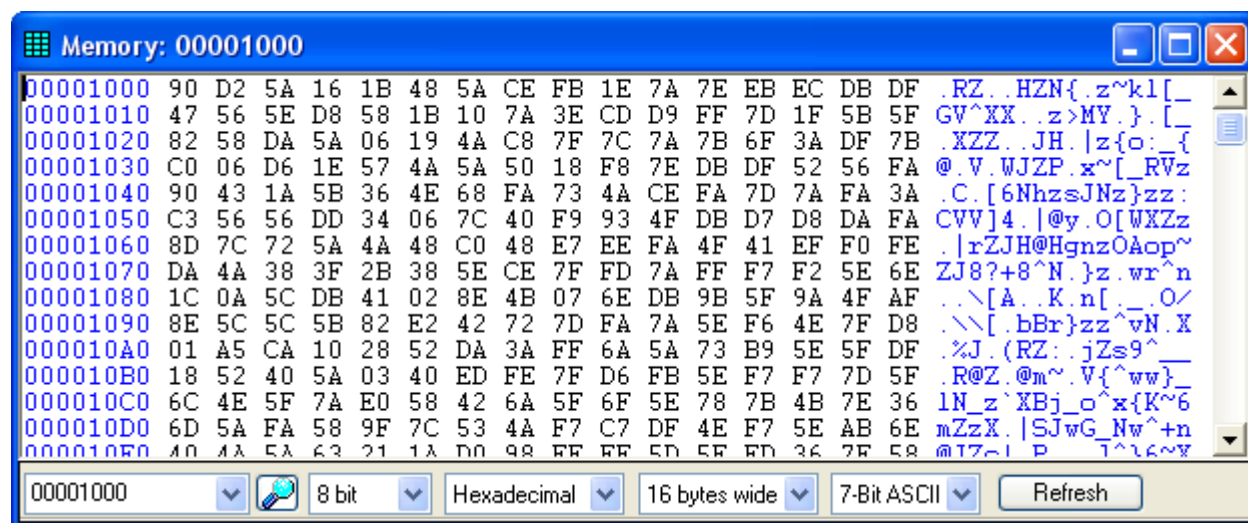
Copy menu item. The **Copy** menu item allows you to copy the **Log** window. It is accessible only in the context menu (available by right-clicking on the **Log** window).

Memory Window

Memory Window 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 drop down 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

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.

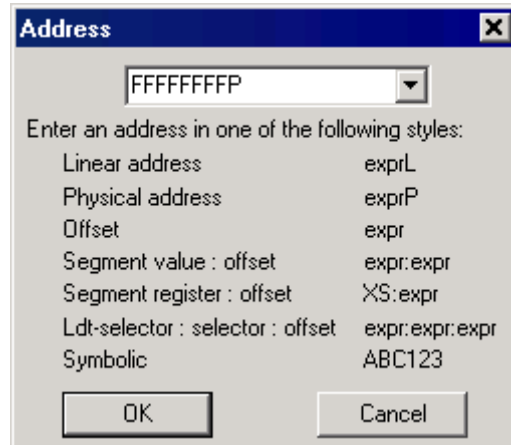
Memory Window Preferences

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.

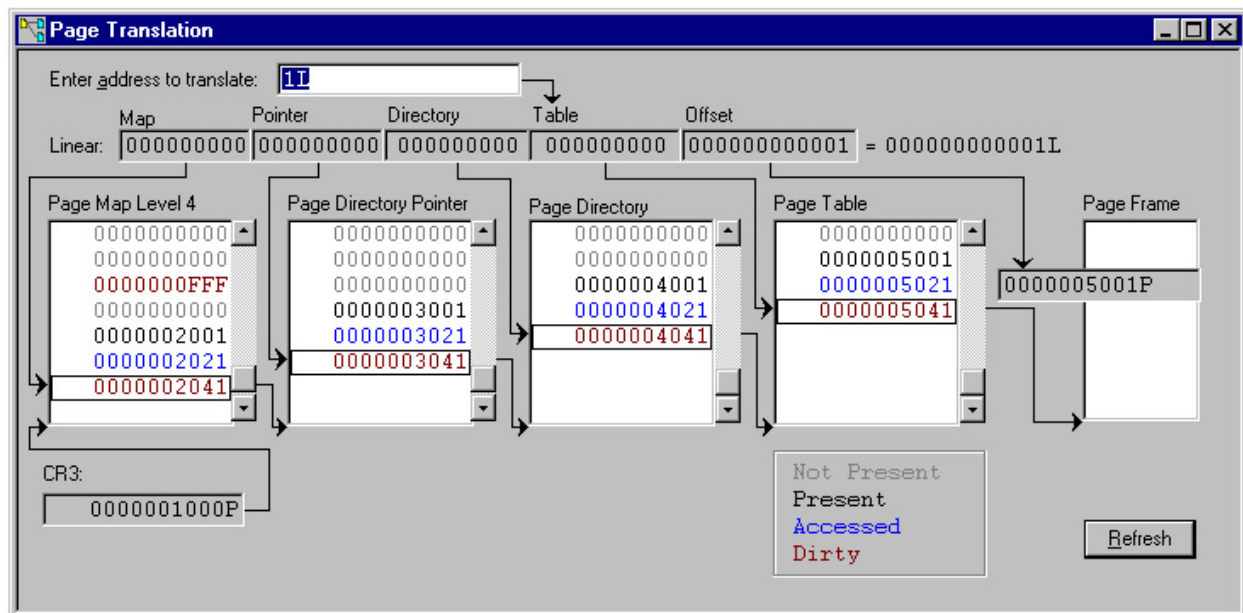
5. Press the Enter key or click on a different address string to activate the new values.

The changed field is displayed in bright green.

Page Translation Window

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.

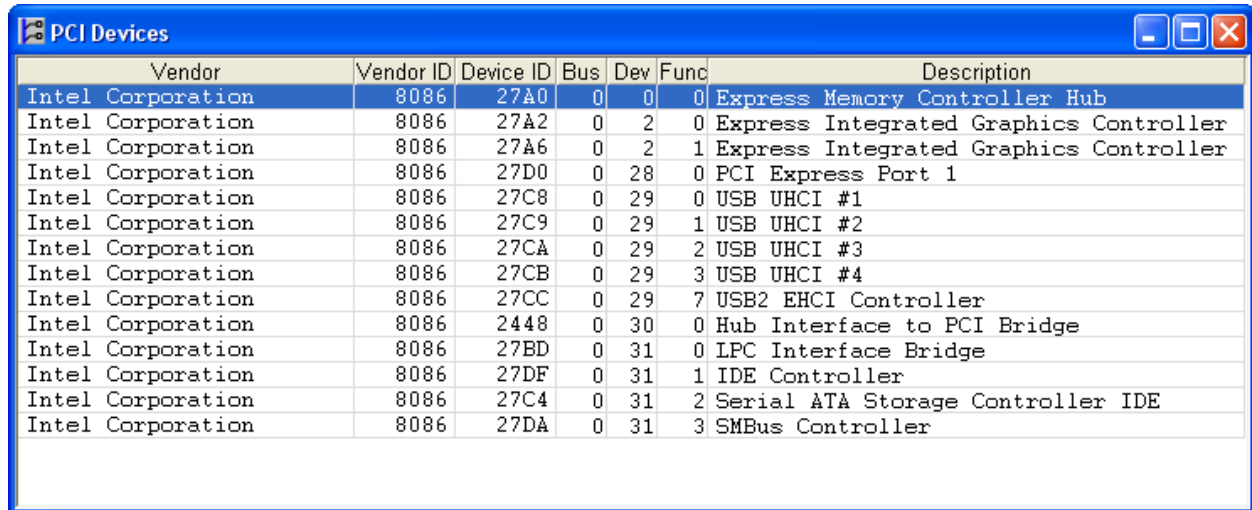
Page Frame

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 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.

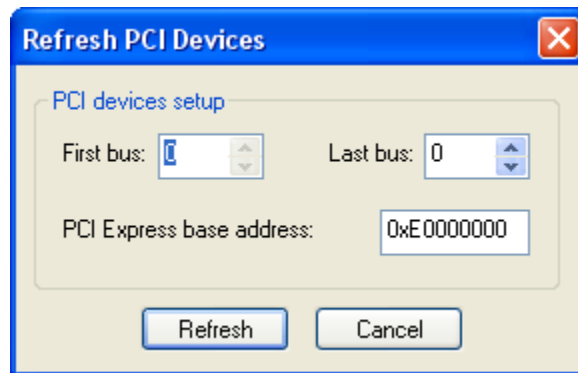
- The Description column displays the device's description string.

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

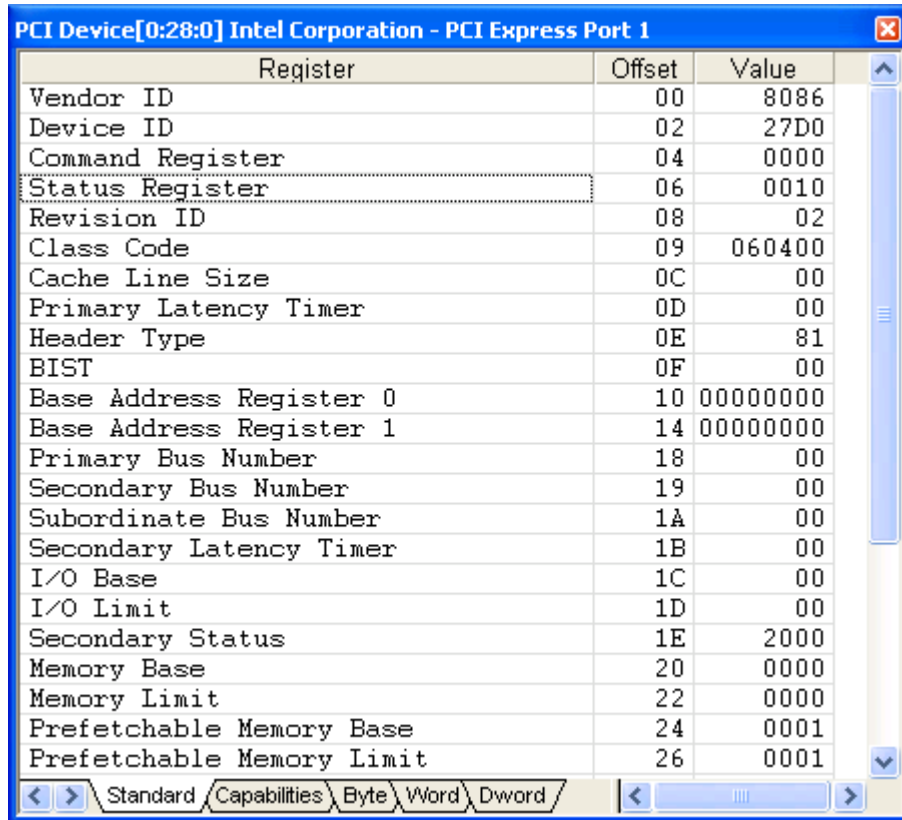
The context menu contains two items, Refresh and View Registers.

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

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.

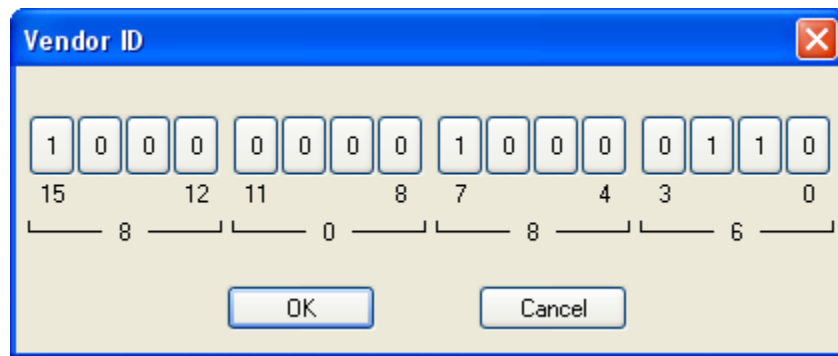
This is a resizable modeless dialog box, meaning that it stays on top of other windows and allows you to switch to other windows while staying active until you close it.

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.



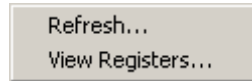
Expand Dialog

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.

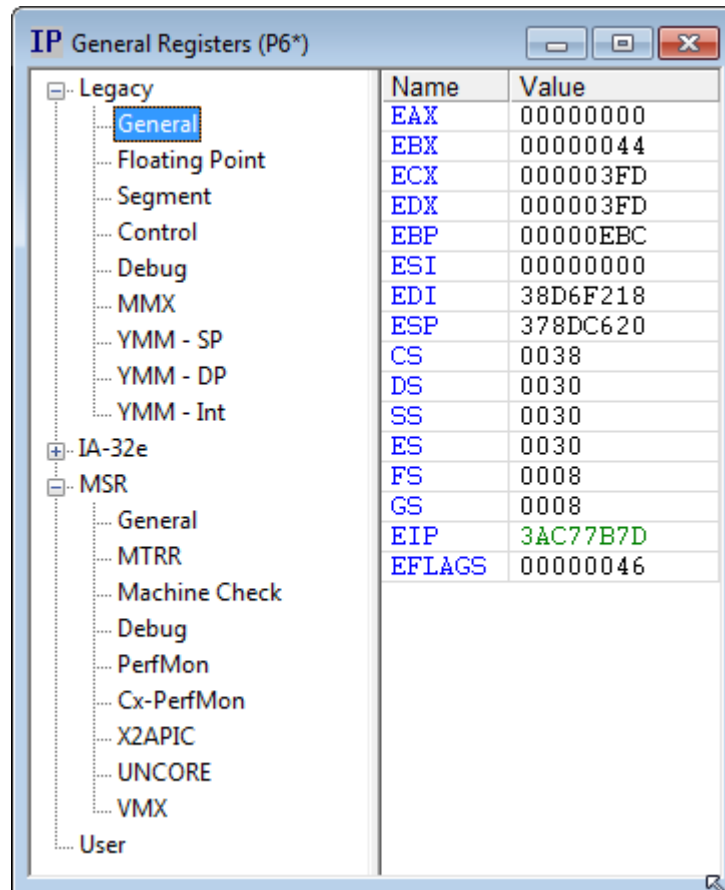
4. Click the **Refresh** button to begin scanning.

Registers Window

Registers Window Introduction

The Registers window displays registers from a selected processor. Multiple Registers windows may be opened.

To open the Registers window, select View|Registers on the menu bar or click on the Registers icon on the toolbar.



Registers window

Register Groups

The left side of the Registers window contains groups of registers that are available for viewing.

Register List

The right side of the Registers window displays the list of processor registers contained in the selected register group. The number of columns displayed varies based on the register group selected. Columns are resizable.

Name. Displays the register name. The tooltip for a register name gives a description of the register.

Value. Displays the register value. Register values are displayed in hexadecimal except for floating point registers which are displayed in scientific notation.

The tooltip 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 tooltip for EFLAGS displays all the processor flags and their values. The tooltip for floating point registers displays the value in hex. The tooltip for segment registers (CS, SS, etc.) displays the base, limit, and access rights associated with that register

Number. This column is only present when an MSR register group is displayed. It displays the MSR number.

Description. This column is only present when an MSR register group is displayed. It displays a description of the MSR.

Registers Window Menu

The Registers menu displays on the menu bar after a Registers window has been opened. It is also displayed when right-clicking in the Register List area of the Registers window

Edit. Edits a register value. Double left-clicking has the same effect. See Editing a Register Value below.

Expand. Displays a register value in binary. See Expanding a Register Value below.

Remove. Removes the currently selected register from the Register list. The register can be restored by selecting Restore Defaults.

Open Code Window. Opens a Code Window to the address specified by the selected register value.

Open Memory Window. Opens a Memory Window to the address specified by the selected register value.

Peek Memory. Read 32 bits of memory using the register value as a memory address.

Show Description. Shows / hides the Description columns.

Copy to User Page. Copies the currently selected register to the User register list. This is useful for creating a custom register list.

Restore Defaults. Restores the default register lists.

Viewpoint. Selects which processor's registers are displayed. This menu item is only present for multi-processor targets. See Processor Selection below.

Editing a Register Value

To edit a register value, either select Edit from the menu, or double click on a register value. Enter the new value, and either press Enter, or cursor from the field to write the value to the processor. Press Esc to cancel an edit

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. Register values cannot be edited while the processor is running.

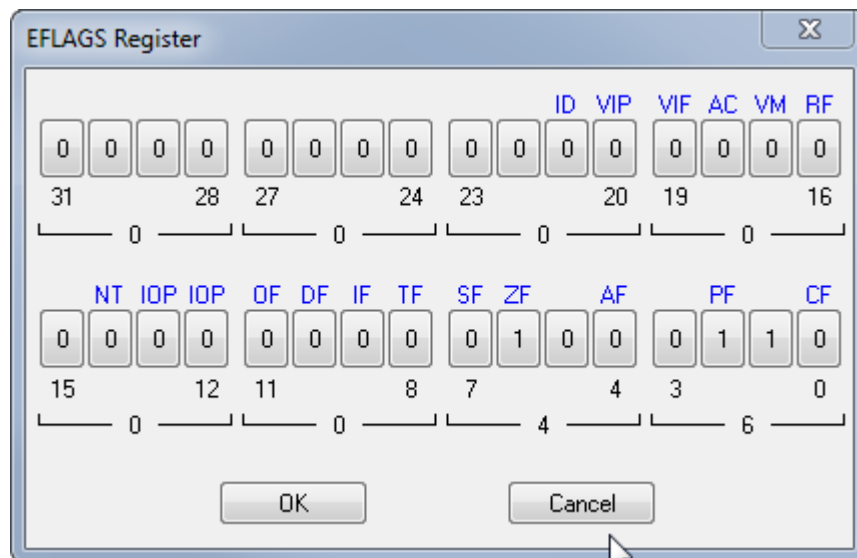
Read-only register values are displayed in gray. These values may not be edited. Write-only register values are displayed as a string of asterisks. To edit a write-only value, type over the asterisks. When the edit is terminated, the value is written to the processor and the register value changes back 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.

Expanding a Register Value

To expand a register to view individual bits, select Expand from the menu. The Expand dialog will open.



Expand dialog

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.

Note: To display a register's bits and have them remain always viewable, open the Watch 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.

Processor Selection

The processor is selected by selecting Viewpoint in the menu, and then specifying a processor. On single processor targets, this menu item is not present.

The default processor selection is Track Viewpoint. When selected, the Registers window automatically switches processors when the viewpoint processor is changed (in the Viewpoint window).

The currently selected processor is displayed in the Title bar. If Track Viewpoint is enabled, then a '*' character is displayed.

Register Value Coloring

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.

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.

Register coloring can be customized by selecting Options|Preferences|Colors.

How To - Registers

Customize 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 the number and location of windows. The currently selected register list and processor selection are also saved.

Adding or Reordering Registers

Registers lists can be reordered. Simply drag and drop a register name to a new position in the register list.

Registers can be added to a list. Either drag the register name to the left-hand pane and drop it on another register group, 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 User register window is no different than any other register list 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.

Print a Register List

Select File|Print to print the currently displayed register list. All registers in the list are printed regardless of the number displayed.

Symbols Windows

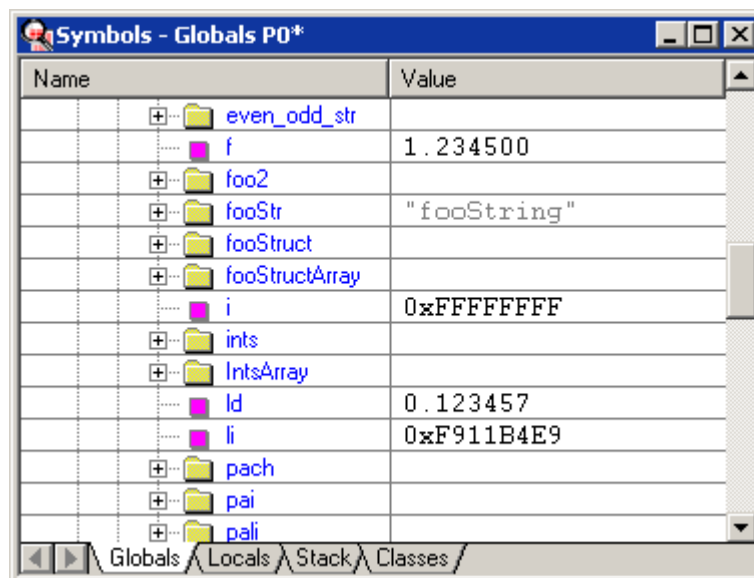
Symbols Window Introduction

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.

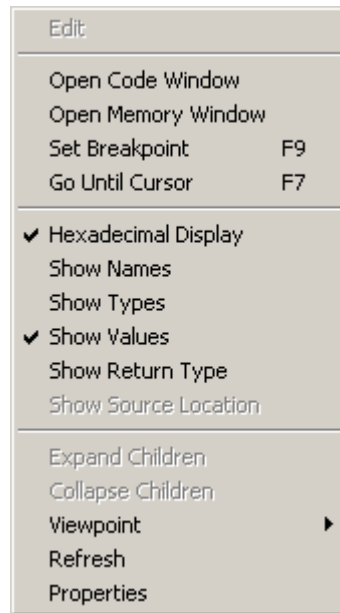
Symbols Window Icon Definitions

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 Hexidecimal.

Show Names/Show Types/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 Introduction

The Trace window displays a history of executed instructions and/or events. The following types of trace are supported:

LBR Trace

LBR trace displays a history of executed instructions. It does this by reading Branch Trace Messages (BTMs) from the Last Branch Record MSRs in the processors.

The advantage of LBR trace is it is non intrusive. 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.

BTS Trace

BTS trace displays a history of executed instructions. It does this by reading Branch Trace Messages (BTMs) from the Branch Trace Store (BTS), an area of system memory set aside for trace.

The BTS can be much larger, and store many more BTMs than LBRs. The disadvantage of using the BTS is that writing BTMs to memory takes 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.

BTS trace can also provide timestamp data to determine how long a section of code took to execute. This timestamp capability requires the same patch mechanism as Event trace (see below).

See [BTS Trace Configuration](#) for a more detailed description of BTS trace.

AET Trace

Architectural Event trace (AET) captures information about certain types of events (e.g., I/O accesses, MSR reads and writes, interrupts, exceptions, BTMs, etc.). AET trace requires either an emulator with trace hardware (e.g., LX-1000) or a target with trace hub capability.

AET trace also provides timestamp data to determine how long a section of code took to execute.

See [AET Trace Configuration](#) for a more detailed description of AET trace.

Intel Processor Trace (Intel PT)

Intel PT is similar to BTS trace in that a region of system memory is set aside to record trace data. The advantage of Intel PT trace over BTS trace is there is very little slowdown of the processor. As with BTS trace, tracing out of reset (before system memory is initialized) is not supported.

Note: Early versions of Intel PT trace were referred to as RTIT (Real Time Instruction Trace).

Note: On some processors, LBR trace and Intel PT are mutually exclusive. SourcePoint will display an error message if both trace types are enabled.

See [Intel PT Configuration](#) for a more detailed description of Intel PT trace.

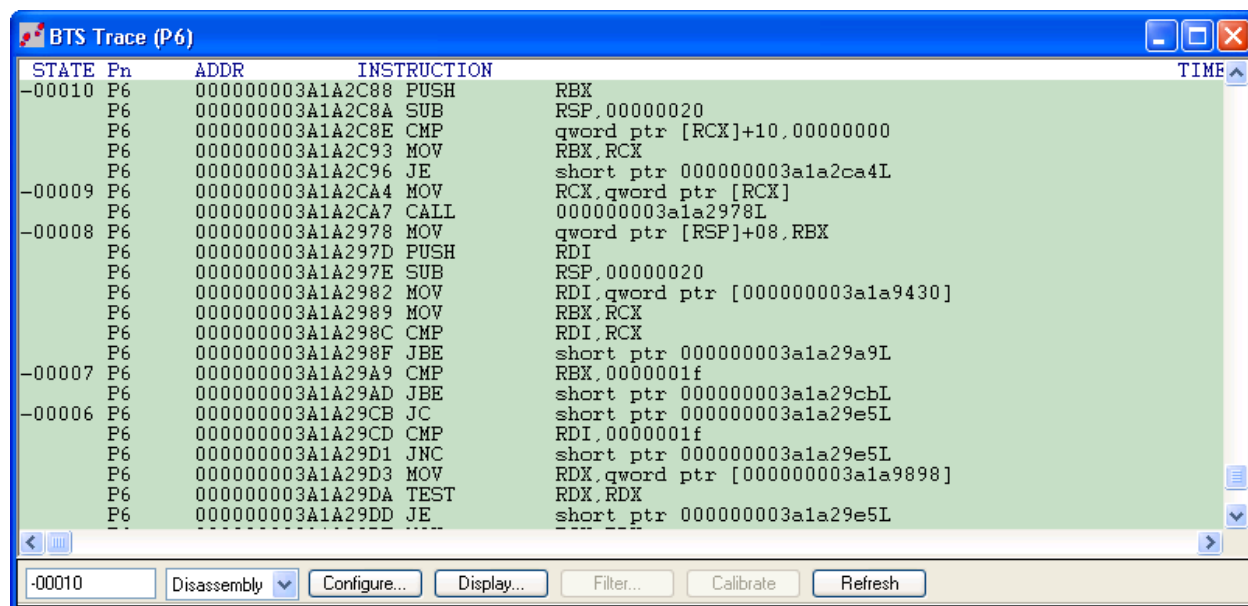
System Trace

System trace refers to software / firmware trace routed through the Trace Hub. The Trace Hub is a new Intel trace architecture available on some targets. If this form of trace is available, then a Trace Hub tab will be displayed in the Trace view's Trace Configuration dialog.

System trace is typically software-generated ASCII messages, but can also include other forms of trace data. See Trace Hub Configuration for a more detailed description of system trace.

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)

More than one trace window can be opened. A common use case is to have one Trace window display BTS trace and another Trace window display Event trace, and to enable automatic time alignment between windows. Scrolling in one Trace window will automatically scroll the other Trace window so that the same area of trace (based on timestamp) is displayed.

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).
- **TimeStamp (all but LBR).** Displays timestamp information. This can include accumulated time (time from a given point in trace), or delta time (time between trace cycles).

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** BTM source address
- **TimeStamp (all but LBR).** Displays timestamp information. This can include accumulated time (time from a given point in trace), or delta time (time between trace cycles). Note: BTS trace has two formats: 32-bit (12 byte BTMs) and 64-bit (24 byte BTMs). Timestamp is only supported with the 64-bit format.
- **STS (Event trace only).** Displays the type of event
- **LIP (Event trace only).** Displays the last instruction pointer value. This is usually the instruction that caused the event.
- **TSC (all but LBR).** Display the raw timestamp counter value. This is the value that timestamps are calculated from.
- **AUX1 (Event trace only).** Displays auxiliary data packet #1. Format varies based on event type.
- **AUX2 (Event trace only).** Displays auxiliary data packet #2. Format varies based on event type.

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.

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.

Filter Button (Event trace only)

The **Filter** button opens the Trace Display Filtering dialog. This dialog is used to select which processor's trace is displayed in the Trace view. This is post capture filtering. The Trace Configuration dialog can be used to select which processors to trace.

Calibrate Button

The **Calibrate** button calibrates the emulator's trace capture hardware. When used with AET trace, this calibrates the LX-1000 trace hardware. When used with SW/FW trace, this forces detection of the presence of the Trace Hub hardware.

Refresh Button

The **Refresh** button causes SourcePoint to re-read memory and trace data, and to re-disassemble trace. It also exits safe mode (DRAM accesses restricted by memory map) if currently enabled. This button is useful if memory was previously unavailable because of safe mode being enabled.

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.

Trace Source Selection. The first few entries in this menu display the available trace sources (e.g., LBR, BTS, etc). Only trace sources that are enabled in the Trace Configuration dialog are displayed. When you enable a trace source, the Trace view automatically switches to that source.

State, Disassembly, Mixed. Selects the current display mode. Provides the same functionality as the Display Mode drop down list in the dialog bar.

Open Tracking 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.

Open Trace Search Window. Opens the Trace Search window. This view provides a fast method for searching for code and data accesses, and also provides high level methods of viewing trace as function calls.

Note: Even though multiple Trace Windows can be opened, there is only one Trace Search window. The Trace Search window is associated with the Trace window it was opened from. If the associated Trace window is displaying IPT trace, then the IPT trace will be searched. If the Trace window is displaying

emulator trace, then emulator trace will be searched. You can associate the Trace Search window with a different Trace window by selecting Open Trace Search Window in another Trace window.

Open Trace Statistics Window. Opens the Trace Statistics window. This view provides code and data profiling.

Note: Even though multiple Trace Windows can be opened, there is only one Trace Statistics window. The Trace Statistics window is associated with the Trace window it was opened from. If the associated Trace window is displaying IPT trace, then the IPT trace will be analyzed. If the Trace window is displaying emulator trace, then emulator trace will be analyzed. You can associate the Trace Statistics window with a different Trace window by selecting Open Trace Statistics Window in another Trace window.

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.

Go To 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.

Zero Timestamp (all but LBR). Zeros accumulated timestamp at the cycle indicated by the caret.

Viewpoint (all but LBR). Selects which processor's trace is to be displayed. If Track Viewpoint is selected, the trace displayed is automatically changed whenever the processor viewpoint is changed.

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.

Filter. Opens the Trace Display Filtering dialog. Provides the same functionality as the Filter button in the dialog bar.

Calibrate. Calibrates the trace capture hardware. Provides the same functionality as the Calibrate button in the dialog bar. When used with AET trace, this calibrates the LX-1000 trace hardware. When used with SW/FW trace, this forces detection of the presence of the Trace Hub hardware.

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.

Load Trace File. Displays disassembly of a binary trace file rather than emulator trace. In order to display disassembly, you must be either connected to the target where the original trace was captured, or the Trace window must be set to use Cached Program, and a program must be loaded.

Save Trace File. Saves all or a portion of the emulator trace as a binary trace file. During the save operation, a progress dialog box with a cancel button is displayed

Close Trace File. This command switches from displaying a binary trace file back to displaying normal trace (from the LX-1000).

Hide Field/Restore Fields. These settings apply to State display mode only. Individual fields can be hidden by positioning the caret in the field to be hidden and selecting the Hide Field menu item from the Trace menu. To re-enable the display of all fields, select Restore Fields. Note: When a new trace source is selected (LBR, BTS or Event), all fields are automatically re-enabled.

Refresh. Refresh trace disassembly. Provides the same functionality as the Refresh button in the dialog bar.

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. Changes in this dialog take effect on the next trace capture.

[Emulator Tab](#)

[LBR Tab](#)

[BTS Tab](#)

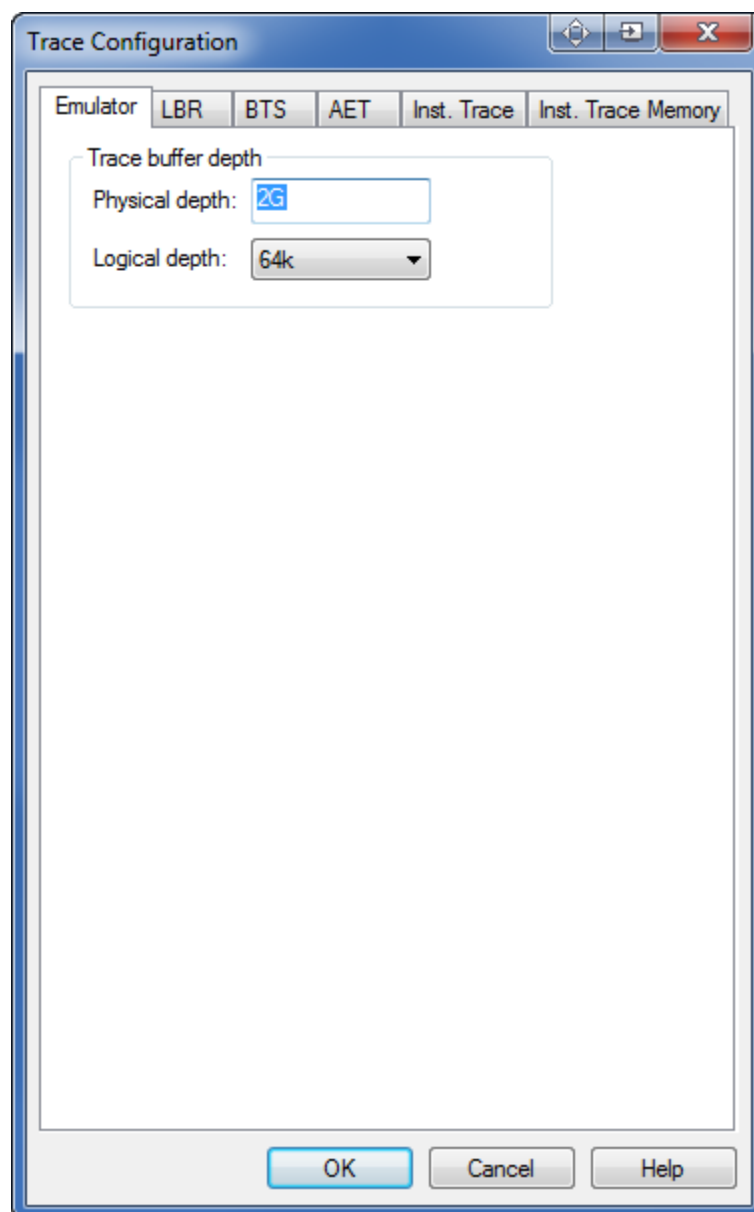
[AET Tab](#)

[Intel PT Tab](#)

[Intel PT Memory Tab](#)

[Trace Hub Tab](#)

Emulator Tab



Emulator Configuration Tab

The emulator tab configures settings specific to emulator. It will only be displayed if you have an emulator with hardware trace support (e.g., LX-1000).

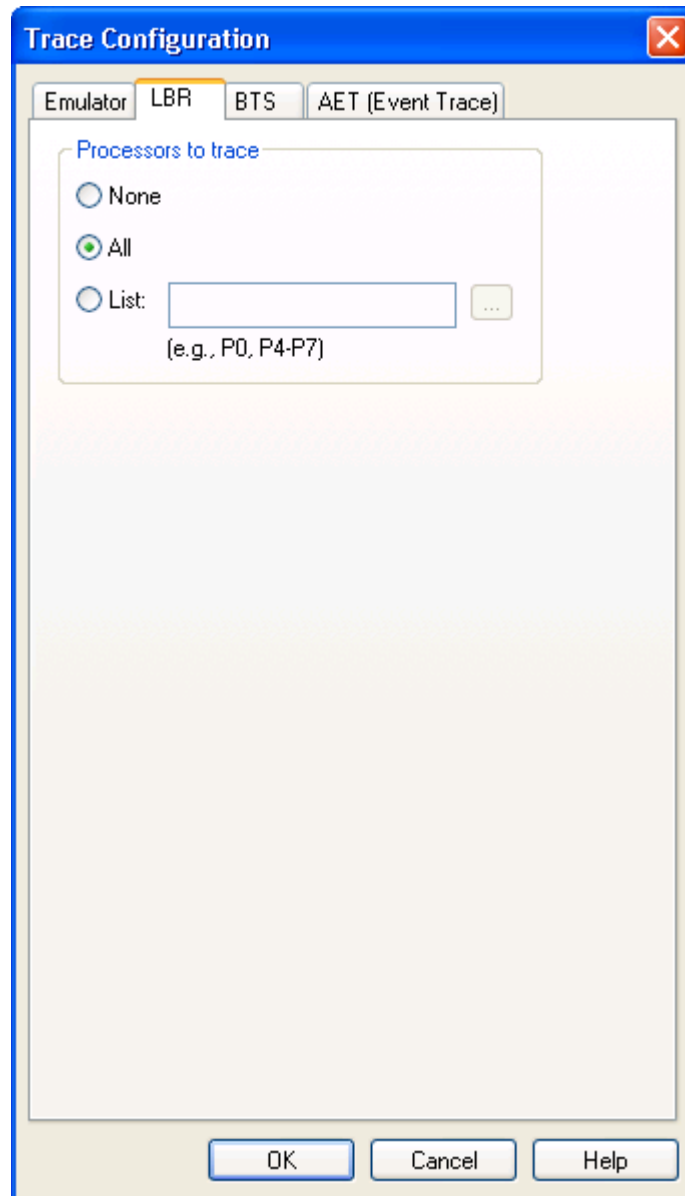
Trace buffer depth section

Physical depth. Displays the amount of trace RAM installed in the emulator. This is a read-only field.

Logical depth. Sets the logical trace depth. This makes it look like the trace buffer is actually much smaller than it really is. One advantage of this is trace cycle numbers can be 5 or 6 digits instead of 10. Another is that searching the entire trace is much faster.

Note: This setting has no effect on the actual recording of trace. If you record a full trace buffer (e.g., 2 Gb), and have a logical trace depth of 64k, there is still 2 Gb of trace data in the emulator. This setting can be changed to different values without having to record new trace data.

LBR Tab



LBR Configuration Tab

Processors to trace section

These controls specify which processors will be traced.

None. Disables LBR trace.

All. Enables LBR tracing on all processors.

List. Enables LBR processor tracing on selected processors. Use commas to list processors (e.g., P2, P4, P6). Use a hyphen to specify a range of processors (e.g., P0-P4). Press the Browse button to the right to graphically specify processors.

BTS tab

Trace Configuration

Emulator LBR **BTS** AET (Event Trace)

Processors to trace

☐ None

☐ All

☒ List: ...
(e.g., P0, P4-P7)

Debug Store location

☐ Use processor settings

☒ Use SourcePoint settings:

Base address:

Length:

Debug Store format

☒ Auto-detect

☐ 32-bit addresses

☐ 64-bit addresses

Trace capture mode

☒ Overwrite

☐ Append

OK Cancel Help

*BTS Configuration Tab***Processors to trace section**

These controls specify which processors will be traced.

None. Disables BTS trace.

All. Enables BTS tracing on all processors.

List. Enables BTS processor tracing on selected processors. Use commas to list processors (e.g., P2, P4, P6). Use a hyphen to specify a range of processors (e.g., P0-P4). Press the Browse button to the right to graphically specify processors.

Debug Store Location section

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 Debug Store.

The Base address field specifies the linear base address of the Debug Store. This field is enabled when Use SourcePoint settings is enabled.

The Length field specifies the length of the Debug 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.

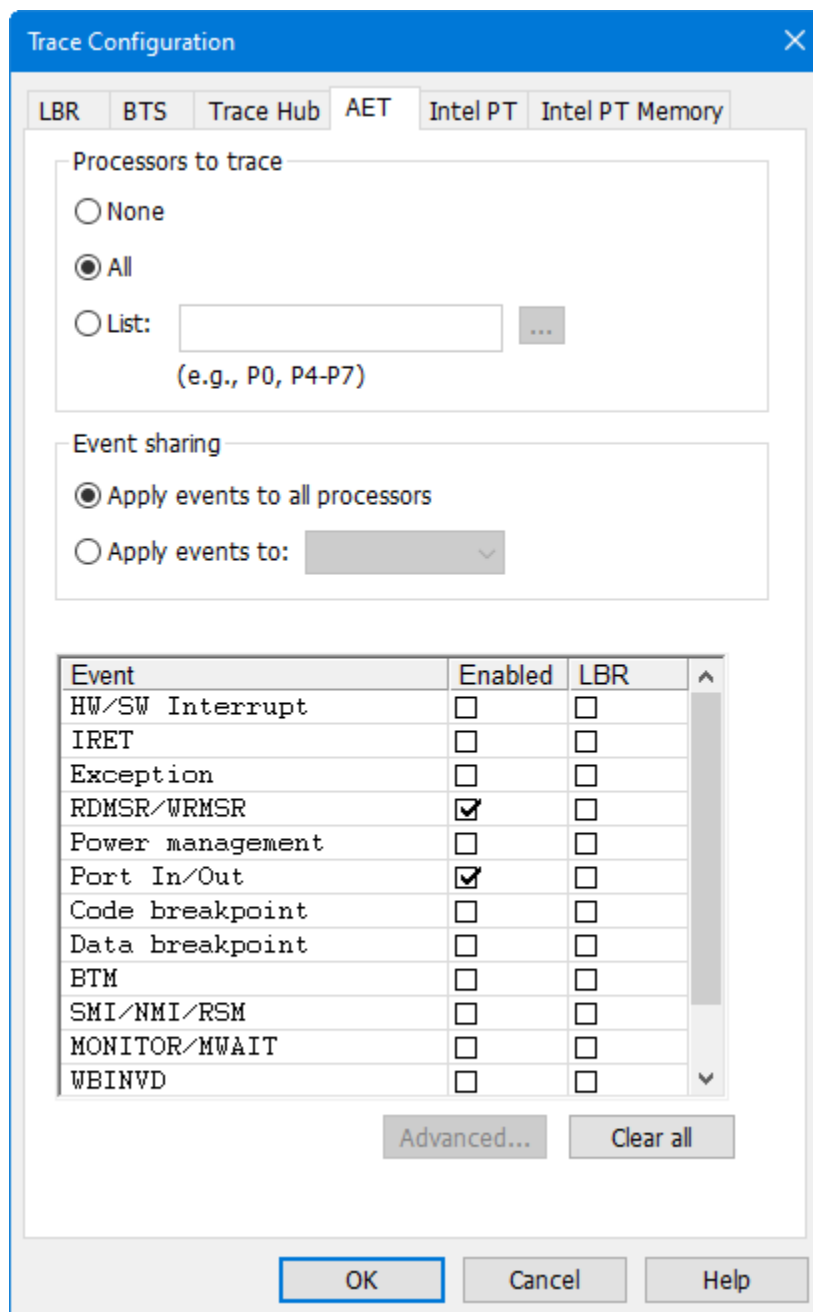
Debug Store Format section

When Auto-detect is selected, SourcePoint will attempt to automatically determine the debug store format. This works in 99% of the cases. If the Trace view is not displaying correct disassembly try forcing this setting to either 32-bit (12 byte BTMs), or 64-bit (24 byte BTMS)

Trace Capture Mode section

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.

AET Trace tab



AET Configuration Tab

Processors to trace section

These controls specify which processors will be traced.

None. Disables Event trace.

All. Enables Event tracing on all processors.

List. Enables Event processor tracing on selected processors. Use commas to list processors (e.g., P2, P4, P6). Use a hyphen to specify a range of processors (e.g., P0-P4). Press the Browse button to the right to graphically specify processors.

Note: If AET is routed through the trace hub, then the trace hub must be enabled in the [Trace Hub Configuration](#) tab.

Event sharing section

These controls specify whether all tracing processors will trace the same events, or whether each processor is tracing it's own events.

When "Apply events to all processors is selected", the events selected in the "Processors to trace" group box will be traced on all processors.

When "Apply events to" is selected, each tracing processor has it's own set of events. The drop down list to the right is used to select a processor. You must select each individual processor, and then select the events you want to trace for that processor

Events to trace section

The list of events varies by processor type. Not all of these events will be available. Each event has an Enabled check box for enabling recording of that event type, and an LBR checkbox. The LBR checkbox controls whether the LBR stack is recorded with each event. This allows for the display of instruction trace leading up to the event.

HW/SW interrupt. Trace hardware and software interrupts, and also trace STI and CLI instructions. The disassembled instruction where the interrupt occurred, and the vector number are displayed.

IRET. Trace IRET instructions. The disassembled instruction is displayed.

Exception. Trace all exceptions. The disassembled instruction that caused the exception, along with the vector number and the value of CR2 are displayed.

RDMSR / WRMSR. Trace all MSR reads and writes. The MSR register number and value are displayed.

Power Management. Trace power state changes.

Port In / Out. Trace all I/O reads and writes. The I/O port number and data value are displayed.

Code breakpoint. Trace code accesses that match hardware (processor) breakpoints in the Breakpoint view.

Note: When this event is enabled, all hardware breakpoints work in conjunction with Event trace, and no longer behave as regular breakpoints.

Data breakpoint. Trace data accesses that match hardware (processor) breakpoints in the Breakpoint view.

Note: When this event is enabled, all hardware breakpoints work in conjunction with Event trace, and no longer behave as regular breakpoints.

BTM. Trace BTM events. This allows for full disassembly of all instructions. Because of the overhead associated with emitting the event data, this can slow execution down quite a bit. BTS trace is a much more efficient way for tracing instructions.

It is also possible to only trace BTM events for a specific range of code. To specify this range, press the Advanced button, and specify the range. See Advanced Configuration below.

Note: If BTS trace or Intel PT is enabled, then they take precedence over BTM event trace, and no BTM events will occur. LBR tracing is not allowed on BTM events.

SMI / NMI / RSM. Trace all SMI, NMI, RSM, INIT and SIPI (Startup IPI) events. The disassembled instruction last executed before the event occurred is shown. For SIPI events, the SIPI vector is also shown.

MONITOR / MWAIT. Trace all MONITOR and MWAIT instructions. The disassembled instruction is shown. For MONITOR the value of EAX is also displayed. For MWAIT the values of EAX and ECX are also displayed.

WBINVD. Trace all WBINVD and CLFLUSH (Cache Line Flush) events. The disassembled instruction last executed before the event occurred is shown. For CLFLUSH events, the address is also shown.

SYSENTER / EXIT. Trace all SYSENTER, SYSEXIT, SYSCALL and SYSRET events. The disassembled instruction is shown. For SYSENTER events, the system call vector is also shown. For SYSEXIT events, the return values of ESP and EIP are also shown.

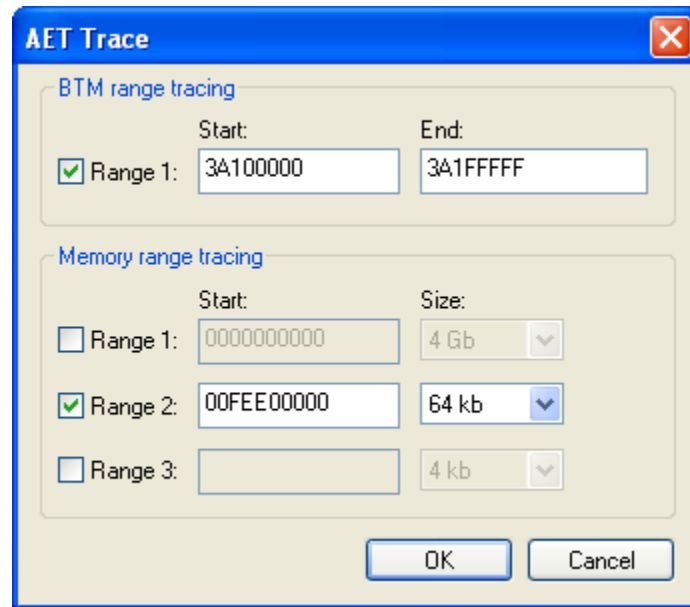
PCICONFIG Read / Write. Trace all PCI Configuration reads and writes. The Bus number, Device number, Function number and Register number, along with the data access value, are displayed.

Memory Trace. Trace accesses to up to three user-defined memory ranges. These ranges are defined in the Advanced configuration dialog (see below). The linear and physical addresses of the access, along with the type (read or write), are displayed.

SGX. Trace the following SGX events: EENTER (enter enclave), ERESUME (resume enclave), EEXIT (exit enclave) and AEX (asynchronous enclave exit). The disassembled instruction is displayed.

Advanced Configuration

The Event Trace Advanced dialog is opened by pressing the Advanced button. It is used to control BTM range and memory range tracing. If neither event is supported on a processor, then the button will be disabled.



Event Trace Advanced Configuration Dialog

BTM range tracing

These controls only have an effect when the BTM checkbox in the Event Trace configuration tab is enabled.

These controls define a specific range of code to trace BTM events. The checkbox enables and disables the range. If the range is disabled, then all BTM events are traced.

The Start and End address values are linear addresses.

Memory range tracing

These controls only have an effect when the Memory Trace checkbox in the Event Trace configuration tab is enabled.

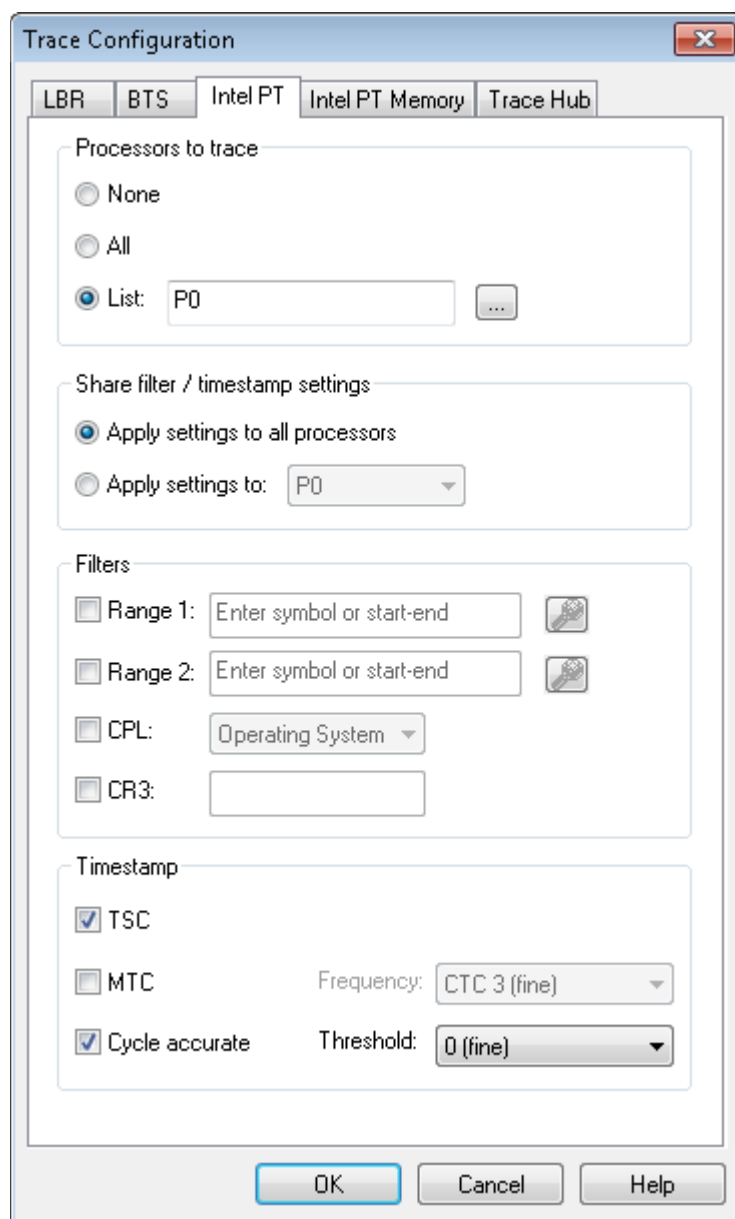
These controls define up to three specific ranges of memory accesses to trace. The checkboxes are used to enable and disable ranges.

The Start address is a physical address. It must be on a 4 kb boundary.

The Size dropdown list defines the size of the memory range to trace. The values available in the list vary based on the alignment of the Start address. For example, if the Start address is on a 16 kb boundary, then 16 kb, 8 kb and 4 kb sizes will be available. If the Start address is on a 4 kb boundary, then the only choice available will be 4 kb.

Intel PT Tab

This tab is used to configure Intel PT trace collection.



Intel PT Configuration Tab

Processors to trace section

These controls specify which processors will be traced.

None. Disables Intel PT trace.

All. Enables Intel PT tracing on all processors.

List. Enables Intel PT processor tracing on selected processors. Use commas to list processors (e.g., P2, P4, P6). Use a hyphen to specify a range of processors (e.g., P0-P4). Press the Browse button to the right to graphically specify processors.

Note: It is processor dependent whether Intel PT trace and LBR/BTS trace can be enabled at the same time. An error message will display at run time if both are enabled on a processor that does not have this capability.

Share filter / timestamp settings section

These controls specify whether all tracing processors share the same filter and timestamp settings, or whether each processor has its own settings.

When "Apply settings to all processors" is selected, the settings selected in the Filters and Timestamp group boxes apply to all processors.

When "Apply settings to" is selected, each tracing processor has its own filter and timestamp settings. The dropdown list to the right is used to select a processor. You must select each individual processor, and then specify the settings you want for that processor

Filters section

Intel PT can optionally be configured to trace instructions only when the CPU is executing code within certain IP ranges. If the IP is outside of these ranges, tracing is disabled.

There are two IP ranges: Range 1 and Range 2. The checkboxes are used to enable these ranges. If neither range is enabled, all instructions are traced. The edit control to the right is used to specify the IP range. It can contain a symbolic range (e.g., a function or module name), or two addresses separated with a hyphen (e.g., 1000-2000). The Symbol finder button to the right can be used to lookup symbol names. It is processor-dependent whether IP ranges are supported.

Intel PT provides the ability to specify whether tracing can occur in OS (CPL=0) or User (CPL>0) modes. The CPL checkbox enables this capability. The dropdown list to the right selects what to trace. If not enabled, then all privilege levels are traced.

Intel PT can enable or disable trace generation depending on the current CR3 value. The CR3 checkbox enables this capability. The edit control to the right specifies the CR3 value to trace.

Timestamp section

Intel PT can be configured to generate timestamp information in the trace stream. Timestamp can be used to measure time within the trace data, or to time align with other Trace views. There are three types of timestamp packets that can be enabled.

TSC. TSC (Timestamp Counter) packets contain TSC values. These are the same values read from the TSC MSR or by using the RDTSC instruction. These packets are required in order to time align with other Trace views. This allows for time alignment with Intel PT from other processors or with SW/FW trace from the Trace Hub.

MTC. MTC (Mini Time Counter) packets contain incremental updates to the CTC (a component of TSC). These values provide slightly more accuracy than TSC packets alone. The frequency of these packets is controlled with the Frequency setting. A value of CTC 6 indicates that a packet is generated whenever bit 6 of the CTC counter changes. These packets are generally not used.

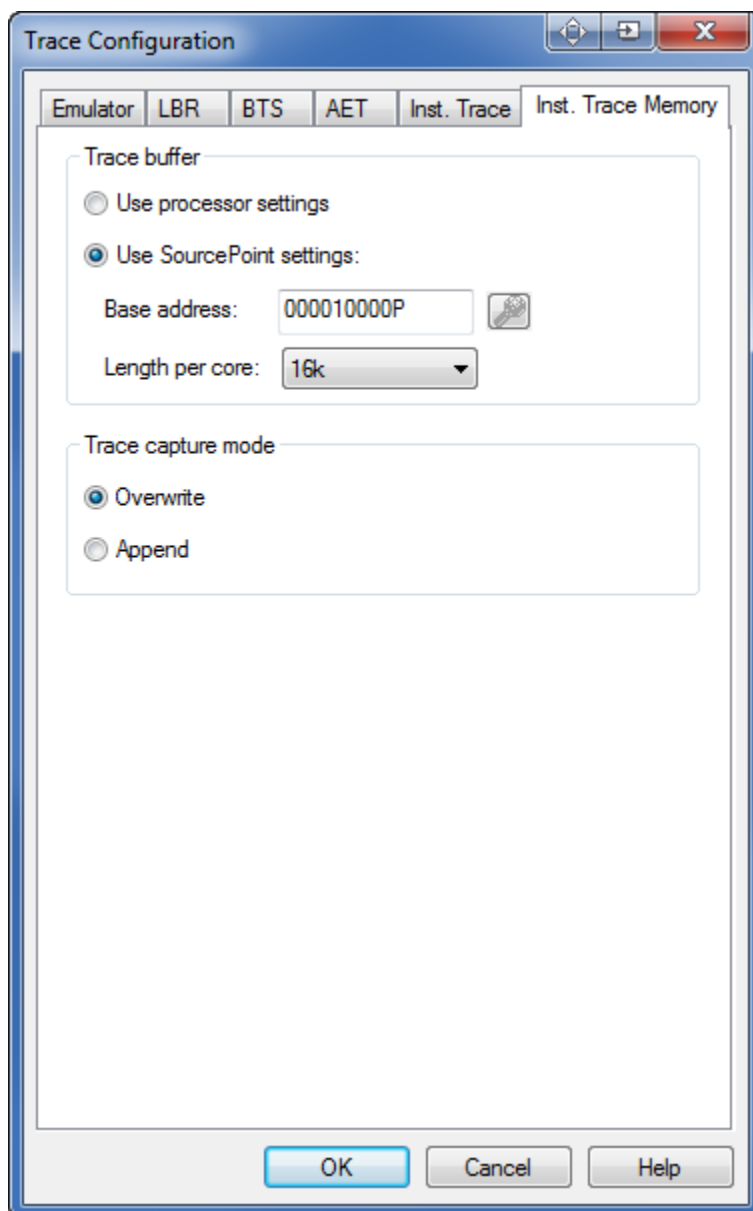
Cycle Accurate. Cycle accurate packets contain information about the number of processor clocks that have occurred. These packets can be used by themselves to measure time within a trace buffer. TSC packets are required to time align with other Trace views. The frequency of these packets is controlled

with the Threshold setting. This value indicates how many processor clocks occur before a packet is emitted.

Refer to the [Timestamp](#) topic in the Advanced section for more information.

Intel PT Memory Tab

The Intel PT Memory tab is used to configure the trace output region in target memory. This area contains a separate trace buffer per processor enabled for trace.



Intel PT Memory Tab

Trace Buffer section

When **Use processor settings** is selected, SourcePoint assumes that the location and size of the trace output region has been set by an operating system, or EFI code.

When **Use SourcePoint settings** is selected, SourcePoint is used to specify the location and size of the trace buffers.

The **Base address** field specifies the physical base address of the trace output region. The base address must be aligned with the length of the trace buffer.

The **Length per core** field specifies the size of the trace buffer. Sizes range from 16 kb to 128 Mb in powers of two. The actual size of the trace output region is the number of processors enabled for trace times the length per core.

Trace Capture Mode section

When **overwrite** is selected, SourcePoint clears the trace buffer prior to each data capture. When **append** is selected, new trace data is appended to existing trace data. The default setting is **overwrite**.

Trace Hub

This tab is used to configure the Trace Hub for system trace collection.

NOTE: The Trace Hub must be enabled in BIOS prior to use. This setting can usually be found in either the PCH section, or in the Debug section.

Software / firmware trace (system trace) is generated by software running on the target. Typically, this consists of ASCII strings, but it can also contain hex data. The Trace Hub supports multiple streams of software trace. In fact, the Trace Hub supports up to 65536 masters, each with up to 256 channels (streams of software trace), for a theoretical maximum of 16M streams of trace.

User-provided metadata is used to describe the display format of System trace. Refer to Trace Hub Metadata for more information.

If the target contains any Trace Hubs, there will be a configuration tab for each.

The image shows the 'Trace Configuration' dialog box with the 'Trace Hub' tab selected. The dialog has a blue title bar and a close button. The tabs at the top are LBR, BTS, Trace Hub (selected), AET, Intel PT, and Intel PT Memory. The 'Masters to trace' section has three radio buttons: 'None', 'All' (selected), and 'List: 112-120'. The 'Trace routing' section has three dropdown menus: 'Trace Hub: System memory', 'Intel PT: System Memory', and 'AET: Trace Hub'. The 'System memory trace buffer' section has two radio buttons: 'Use BIOS settings' and 'Use SourcePoint settings' (selected). Below these are fields for 'Base address: 100000000P' and 'Length: 512k'. The 'Timestamp' section has a checkbox for 'Alignment packets' (unchecked) and a 'Frequency: CTC 16' dropdown. The 'Master / Channel definitions' section has a 'Filename: rs\My Documents\Project files\SKL ports.xml' field. At the bottom are 'OK', 'Cancel', and 'Help' buttons.

Trace Hub Configuration Tab

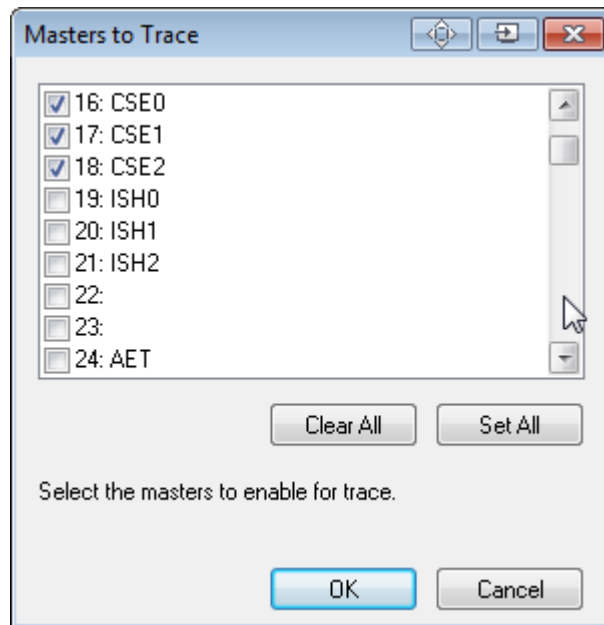
Masters to Trace Section

These controls specify which masters will be traced.

None. Disables Trace Hub trace.

All. Enables tracing of all masters.

List. Enables tracing of selected masters. Use commas to list individual masters (e.g., 2, 4, 6). Use a hyphen to specify a range of masters (e.g., 32-256). Press the Browse button to the right to graphically specify masters.



Masters to Trace dialog

This dialog displays master names rather than just the master numbers. The names come from the XML metadata file user-specified in the Master / channel definitions file.

Trace Source Routing section

These controls specify the routing of different types of trace data to different destinations.

Trace Hub. Trace Hub routing refers to Software / Firmware trace. There are currently three trace destinations: system memory, MTB and DbC. When system memory is selected, trace data is stored in memory according to the settings in the Trace Buffer section. When MTB is selected, trace data is stored in the MSU trace buffer inside the Trace Hub. The advantage of using the MTB is that trace data can be captured very early in the boot process, before system memory has been initialized. The disadvantage of using the MTB is that it's very small (typically 2 kB), where a memory-based trace buffer can be as large as desired. The DbC selection is only available when you have a DCI connection to the target. In this case, trace data is streamed from the target and stored in an internal 512 MB trace buffer inside SourcePoint. This trace mode has the same advantages of MTB in that trace can be captured before system memory is available.

Intel PT. Intel PT can be routed through the Trace Hub intermixed with other types of trace data. Intel PT can also be routed directly to system memory without going through the Trace Hub. Currently, SourcePoint only supports the latter. All configuration for Intel PT is performed in the two Intel PT tabs.

AET. AET (Architectural Event Trace) can currently only be routed through the Trace Hub intermixed with other types of trace data. All configuration for AET is performed in the AET tab.

Trace Buffer section

When Use BIOS settings is selected, SourcePoint assumes that the location and size of the trace output region has been set by the BIOS. This is the safest setting to use since the BIOS will ensure the memory is dedicated to the Trace Hub, and will have defined a MTRR register to make it uncacheable.

When Use SourcePoint settings is selected, SourcePoint is used to specify the location and size of the trace buffer.

The Base address field specifies the physical base address of the trace output region. If Use BIOS settings is selected this field display the region allocated by the BIOS. If Use SourcePoint settings is selected this field displays the user-allocated region. The base address must be aligned on a 4 kb boundary. This region must be in uncacheable memory.

The Length field specifies the size of the trace buffer. Sizes range from 8 kb to 2 Gb in powers of two.

Timestamp section

SW/FW trace can be time aligned with other trace sources including AET and Intel PT. In order to convert SW/FW timestamp values to TSC values for alignment, timestamp alignment packets must be enabled.

Alignment packets: Enables timestamp alignment packets. This is only required when time alignment with other trace sources in other Trace views is desired. Note that this setting does not control whether SW/FW trace actually contains timestamp packets. This is controlled by user code generating the SW/FW trace data.

Frequency: Controls the frequency of the timestamp alignment packets. A value of CTC 16 indicates that whenever bit 16 of the CTC timer changes, a packet is emitted. The correct value to use is dependent on the density of the SW/FW trace. The sparser the trace, the higher this value should be set. Ideally, an alignment packet should be present every few thousand bytes of SW/FW trace.

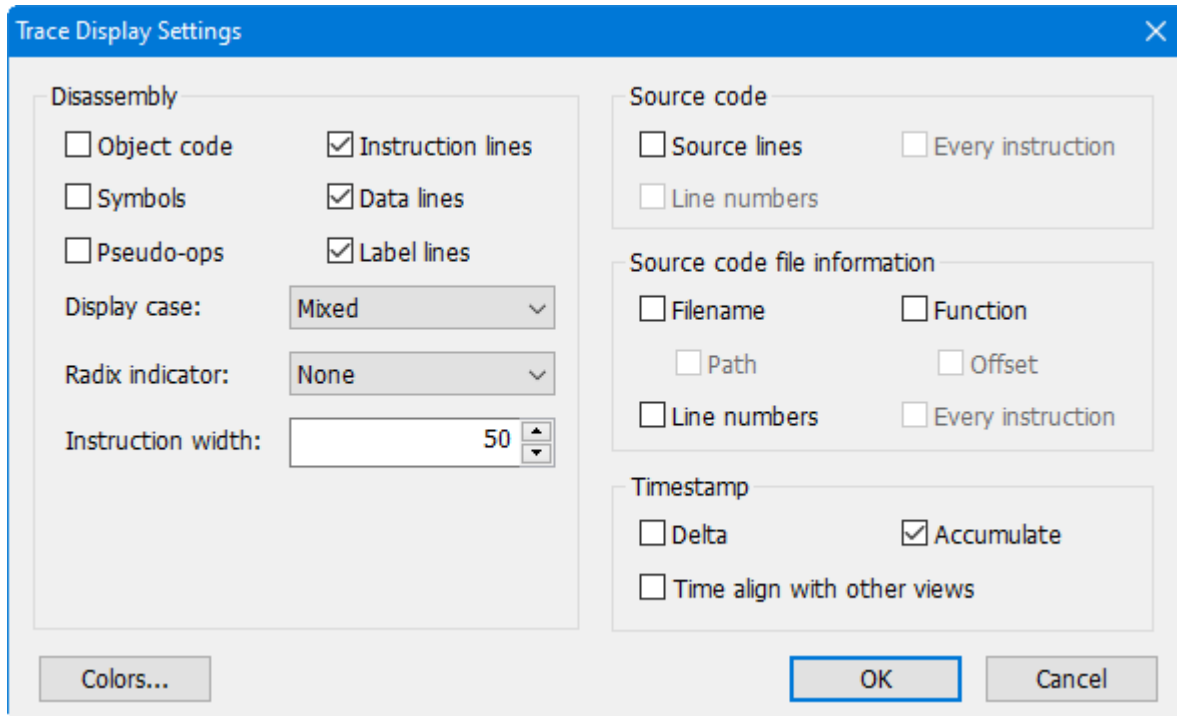
Refer to the [Timestamp](#) topic in the Advanced section for more information.

Master / Channel Definitions section

The Filename control is used to specify the location of the Trace Hub metadata file. This XML file contains the Master / Channel display formatting information. The button to the right can be used to browse to the file. See [Trace Hub Metadata](#).

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.



Disassembly section

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.

Pseudo-ops. Enables the display of pseudo-ops (e.g., far ptr, near ptr, byte ptr, etc.)

Instruction lines. Enables the display of disassembled instruction lines.

Data lines (Event trace only). Enables the display of data values for I/O and MSR events.

Label lines. Enables the display of informational lines (best if enabled).

Display Case. Options are Mixed, Upper, and Lower.

Radix Indicators. Options are Prefix, Suffix, and None.

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.

Timestamp Section

These controls allow you to configure timestamp display (BTS and Event trace only).

Delta. This option displays the time between cycles.

Accumulate. This option displays the accumulated time from a given point (by default, the trigger location). This point can be changed by selecting zero timestamp in the Trace window menu.

Time Align with Other Views. This option enables multiple trace views to track each other. When a tracking Trace window is scrolled, all other tracking windows will automatically scroll to show the same area of trace. Timestamp data is used to perform the alignment. A common example is to have one Trace window display BTS trace and another Trace window display Event trace, and have them both set to track each other.

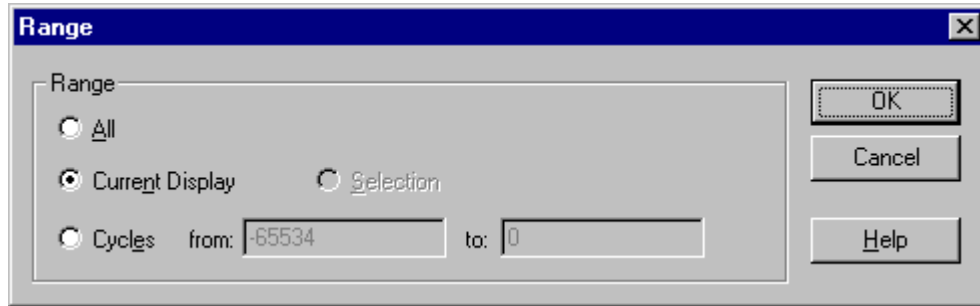
Colors. A shortcut to the Colors tab under Options | Preferences. This allows for changing the colors (including processor background colors) in the Trace view.

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*

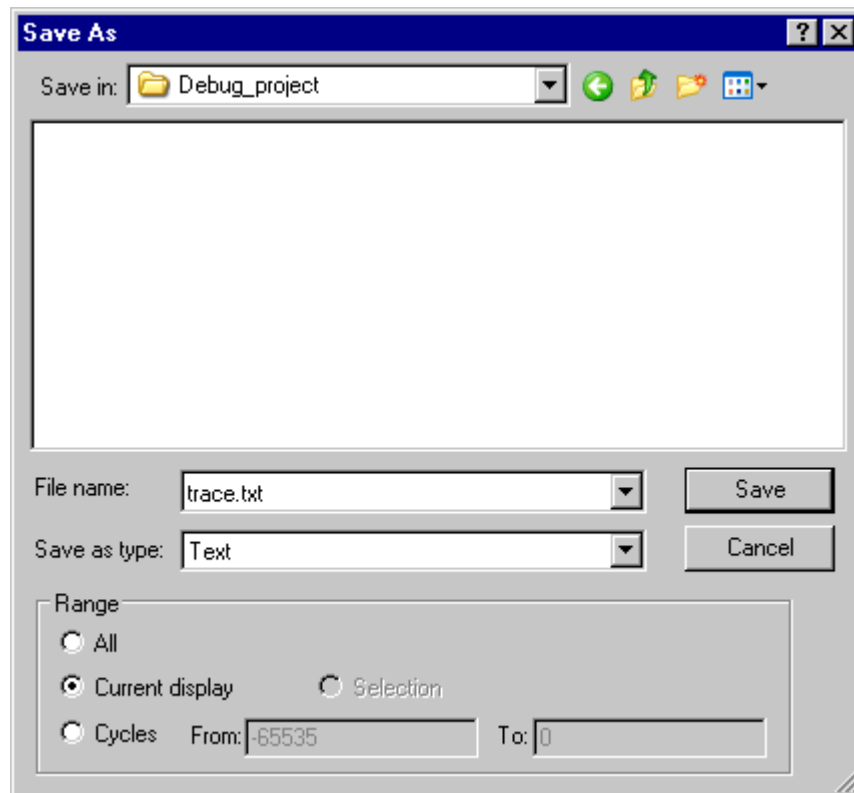
2. Choose the range or cycles to print.
3. Click the **OK** button.

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.

Advanced

Timestamp

Overview

Timestamps are used to measure time in a trace buffer. They are displayed in the Trace view, the Trace Search view, and the Trace Statistics view. Their use is optional in all cases except for the Calls Chart tab and the Function Profiling tab, which require timestamps in order to measure function execution times.

There are two timestamp display modes, Accumulate and Delta. In Accumulate mode, time is measured from the zero time point in the trace buffer (by default the trigger location). In Delta mode, time is measured from the current trace cycle to the previous trace cycle. The timestamp display mode is set in the Trace Display Settings dialog.

Trace Types

Each trace type has different trace buffer and timestamp capabilities. Following is a brief description of each:

LBR Trace

LBR trace is instruction trace captured in the Last Branch Record MSRs. LBR trace has a separate trace buffer for each core. LBR trace does not have timestamp capability.

BTS Trace

BTS trace is similar to LBR trace, but is stored in system memory rather than MSRs. BTS trace has a separate trace buffer for each core. BTS trace timestamp capability is optional. BTS timestamp is only available with an AET trace patch that enables it.

Timestamp values are based on the processor's current TSC value (the same value read from the IA32_TIME_STAMP_COUNTER MSR). These values can be used to time align BTS trace between cores or with AET trace. Contact Intel for more information about the AET availability.

AET

AET trace records events in the LX-1000 emulator trace buffer or in system memory via the Trace Hub. This single trace buffer contains Event trace from all cores. Like BTS trace, AET trace has timestamp values based on the TSC.

Intel PT

Intel PT records instruction trace in system memory. It is the replacement for BTS trace. Intel PT has a trace buffer per core. Intel PT has two types of timestamp, Cycle Accurate and TSC.

Cycle Accurate counts processor clocks so that relative time can be measured within a trace buffer. It does not allow for time alignment with other Trace views. It is enabled with the Cycle Accurate checkbox in the Intel PT Configuration tab.

TSC timestamp is enabled with the TSC checkbox in the Intel PT Configuration tab. It consumes slightly more bandwidth (extra timestamp packets are generated), but it allows for time alignment with other Trace views.

The typical uses cases are:

1. Timestamp is not required. Both checkboxes are unchecked.
2. Timestamp is required. Both checkboxes are checked.

See [Intel PT Configuration](#) for more information.

Trace Hub

The Trace Hub trace buffer contains SW/FW trace from all cores. SW/FW trace though the Trace Hub uses its own timestamp counter that is not related to TSC timestamp. This allows for relative time to be measured within the buffer, but does not allow for time alignment with other Trace sources that use TSC timestamp (e.g., AET and Intel PT).

SW/FW timestamp values can be converted to TSC times by enabling Timestamp Alignment packets. These are extra packets that contain snapshots of both counters. These are enabled with the “Alignment packets” checkbox in the Trace Hub Configuration tab.

See [Trace Hub Configuration](#) for more information.

Time Alignment

Each SourcePoint Trace view displays trace from a single trace source. The choices available are shown at the top of the Trace view context menu. As mentioned above some trace sources have a trace buffer per core, while others have a single trace buffer for the entire system. For trace sources with a buffer per core, the core can be selected by selecting Viewpoint in the Trace view context menu. The current trace source and trace buffer are indicated in the Trace view Title bar.

Once timestamp has been configured correctly, Trace views can be time aligned, so that scrolling in one causes automatic scrolling in another.

Time alignment is enabled with the “Time Align with Other Views” checkbox in the Trace Display Settings dialog. A time aligned Trace view has “(time aligned)” added to its Title bar. When you place the caret in a time aligned Trace view, or scroll in that view, all other time aligned views will update to display trace at the same absolute time as the first view.

Trace Hub Metadata

The Trace Hub metadata is one or more XML files used by SourcePoint to format and display Trace Hub trace data. It includes information about the masters and channels in a system. This includes names, display formats, and system-wide settings.

The metadata is used for display purposes only. It is not required to decompress the Trace Hub trace stream. If metadata is not present, hex values will be displayed for master and channel names, and ASCII will be the default display format.

Following is a description of the different sections contained in the metadata file(s):

Files Section

The Trace Hub metadata can be contained in a single XML file, or multiple XML files. The files section is used to specify additional metadata files to process. File paths can be absolute or relative to the location of the top level file. There is no limit to the levels of file nesting.

Following is an example of nested metadata files. The top level file (specified in the Trace Hub Configuration tab) includes two additional files:

```
<files>
  <file name="SKL ports1.xml"/>
  <file name="C:\Users\harry\Documents\Project files\SKL ports2.xml"/>
</files>
```

Settings section

The Settings element contains system-wide settings. These settings apply to all masters and channels in the system.

```
<settings>
  <setting name="mark terminates message" value="false"/>
  <setting name="newline terminates message" value="false"/>
  <setting name="payload nibbles swapped" value="true"/>
  <setting name="master/channel display base" value="decimal"/>
</settings>
```

Messages are made up of multiple Trace Hub data packets. Data packets can be “marked” to indicate message boundaries. Whether a marked packet contains the first byte(s) of a message, or the last byte(s) of a message, depends on the software generating the messages.

The “mark terminates message” attribute is used to inform SourcePoint how marks are to be interpreted. If the "mark terminates message" setting is set to true, marks terminate a message. If the "mark terminates message" attribute is set to false, marks begin a message.

Newlines can be implied to terminate ASCII messages. If the "newline terminates message" attribute is true then newlines end a message. If messages contained multiple lines, then this setting should be set to false.

The "payload nibbles swapped" attribute needs to be set on certain early Trace Hub implementations.

The "master/channel display base" attribute controls the display base of master and channel numbers. The default is "hex".

Masters Section

The masters element defines the masters and channels in a system. These include master names, channel names, and display formats. Following are two simple master definitions:

```
<masters>
  <master id="0x74" name="P1" format="ASCII">
    <channels>
      <channel id="0x43" name="Norwegian"/>
      <channel id="0x45" name="Blue"/>
      <channel id="0x4a" name="Parrot"/>
    </channels>
  </master>
  <master id="0x75" name="P2" format="ASCII">
    <channels>
      <channel id="0x20" name="Walter"/>
      <channel id="0x21" name="White"/>
      <channel id="0x22" name="Heisenberg" format="hex"/>
    </channels>
  </master>
</masters>
```

The first master corresponds to master number 0x74 (master numbers are dependent on the target hardware). The metadata file associates the name "P1" with master number 0x74. When data from this master is displayed in the Trace view, "P1" will be displayed instead of 0x74.

There are 3 channels defined for master P1 (usually there would be many more). Channel numbers are dependent on the software generating the Trace Hub trace data. The metadata associates the name "Blue" with channel number 0x45. Without the metadata file, the Trace view will display the master and channel as 74:45. With the metadata file, the Trace view will display the master and channel as P1:Blue.

Each individual channel can have its own display format (more on display formats later). If all channels in a master use the same display format, then the format can be specified at the master level. This is the case for master P1 which specifies a display format of "ASCII" for all channels.

If most channels in a master are one display format, and a few channels are a different display format, then the display format can be overridden at the channel level. This is the case for channel 0x22 in master P2. Instead of displaying ASCII data, hex data will be displayed.

Display Formats

The following display format are supported:

ASCII. Data packets are collected and displayed as a single ASCII string. Strings are delimited by data markers, flag packets, null characters or optionally newline characters.

Example: format="ASCII"

Hex. Each Trace Hub data packet is displayed as a single hex value. The Size attribute specifies the display size.

Example: format="hex" size="32"

Tabular Hex. Data packets are collected and displayed as tabular hex data. The Columns and Size fields are used for formatting. Tabular data is delimited by either data markers or flag packets.

Example: format="tabular hex" columns="16"

Hosted Printf. Similar to ASCII string display, but SourcePoint performs the printf. See [Hosted Printf](#) for more information.

Example: format="hosted printf"

Table Printf. Similar to ASCII string display, but SourcePoint performs the printf. See [Table Printf](#) for more information.

Example: format="table printf"

Bitfields. Displays a value as a series of bit fields rather than as a single hex value. See [Bit Fields](#) for more information.

Example: format="bit field" template="myBitData"

Display Format Usage

Display formats can be specified at two different levels: per master, or per channel. This design is meant to minimize the size of the metadata.

Example - Most channels of a master display ASCII data, but a few display hex data. Set the master display format to ASCII, and override the default at the channel level.

Note: This design does not allow a single channel to use a mix of display formats. If this is desired, then a pair of channels should be allocated.

Bit Fields

Bit fields display an 8, 16, 32 or 64-bit value as a series of bit fields rather than as a single hex value. There are two ways to specify a bit field definition. It can be entered in the channel definition, or if there are multiple channels that share the same bit field format, it can be entered as a bit field template, and referenced by name.

Example 1 – Bit field definition in channel definition

The following defines 3 bit fields in a 32-bit value. The start attribute indicates the starting bit position for the field. The size attribute specifies the size in bits of the field.

```
<channel id="0x26" name="blue" format="bit field">
  <bitfields size="32" format="a=%04X, b=%02X, c=%02X">
    <bitfield start="16" size="16"/>
    <bitfield start="0" size="8"/>
    <bitfield start="8" size="8"/>
  </bitfields>
</channel>
```

The bit field display format is defined by a user-specified printf format string (format above). The arguments to printf are the bit fields in the order they appear in the metadata. Assuming a 32-bit hex value of 0x12345678, the output will look like this:

```
a=1234, b=78, c=56
```

Example 2 – Bit field definition from a bit field template

The following example generates the same output as Example 1, but uses a bit field template. The bit fields are defined once (in the bit field templates section), and a name is assigned to the template (in this case myBitData). The channel definition then references the bit field template by name.

```
<channel id="0x26" name="blue" format="bit field" template="myBitData"/>
<bitfieldtemplates>
  <bitfields name="myBitData" size="32" format="a=%04X, b=%02X, c=%02X">
    <bitfield start="16" size="16"/>
    <bitfield start="0" size="8"/>
    <bitfield start="8" size="8"/>
  </bitfields>
</bitfieldtemplates>
```

Enumerations

Enumerations are used to show strings instead of values in bit fields. In the following example there are two enumerations defined: “animals” and “people”. In the bit field definition, the second field specifies that the value is to be displayed using the “animals” enumeration.

```
<bitfields size="32" format="a=%04X, b=%s, c=%02X">
  <bitfield start="16" size="16"/>
  <bitfield start="0" size="8" enumeration="animals"/>
  <bitfield start="8" size="8"/>
</bitfields>

<enumerations>
  <enumeration name="animals">
    <item name="cat" value="0x78"/>
    <item name="dog" value="0x44"/>
  </enumeration>
  <enumeration name="people">
    <item name="bob" value="0"/>
    <item name="carol" value="1"/>
    <item name="ted" value="2"/>
  </enumeration>
</enumerations>
```

```
<item name="alice" value="3"/>
</enumeration>
</enumerations>
```

Assuming a 32-bit hex value of 0x12345678, the display will look like this ("cat" displayed instead of "78"):

```
a=1234, b=cat, c=56
```

Hosted Printf - Normal

The Hosted Printf display format specifies that SourcePoint should perform the printf rather than target software. This is accomplished by emitting the address of the printf format string, and the printf arguments as data packets. This greatly reduces the number of bytes traced.

For example, the following printf call generates 28 bytes of trace data if the printf is performed by target software. Using Hosted Printf results in only 12 bytes of trace data.

```
printf("index = %2d, value = %08X", _nIndex, _nValue)
```

An additional advantage of Hosted Printf is the decreased processing time for target software to generate the message. Rather than executing potentially millions of printf's (which may be thrown away if the trace buffer wraps), the printf's are performed just-in-time by SourcePoint as they are displayed.

Note: Hosted printf requires changes to target software generating the Trace Hub data. Contact ASSET InterTech for more information.

Note: Hosted printf can only be used on systems where the program traced is either still in memory, or loaded into the SourcePoint (SourcePoint needs access to the original printf format string).

Hosted Printf – Table-based

Table-based Hosted Printf is identical to normal Hosted Printf except the format strings come from tables in the metadata. Rather than the address of the format string being emitted with the trace data, a key value is generated instead. The key can be 8, 16, 32 or 64 bits.

Following is an example of a format string table and how to use it:

```
<master id="0x77" name="Table" format="ASCII">
  <channels>
    <channel id="0x20" name="Printf" format="Table printf"
table="table1"/>
  </channels>
</master>

<messagetables>
  <messagetable name="table1">
    <message id="0x1000" format="This is test 1: %08X"/>
    <message id="0x20000000123" format="This is test 2: %016l1X"/>
    <message id="0x3000" format="This is test 3"/>
  </messagetable>
</messagetables>
```

```
</messagetables>
```

Trace View Coloring

Masters and Channels can be displayed in different colors in the Trace view (background colors only). To display all channels in a particular master with the same color, a color attribute is specified at the master level. To display an individual channel with a different color, specify the color at the channel level.

Up to eight different colors can be used. These colors are shared with the eight different colors used in processor trace display. Press Colors in the Trace Display Settings dialog to access these color settings. The colors are labelled Processor 0 – Processor 7. If a color is not specified, then the default color (labelled TraceHub) will be used.

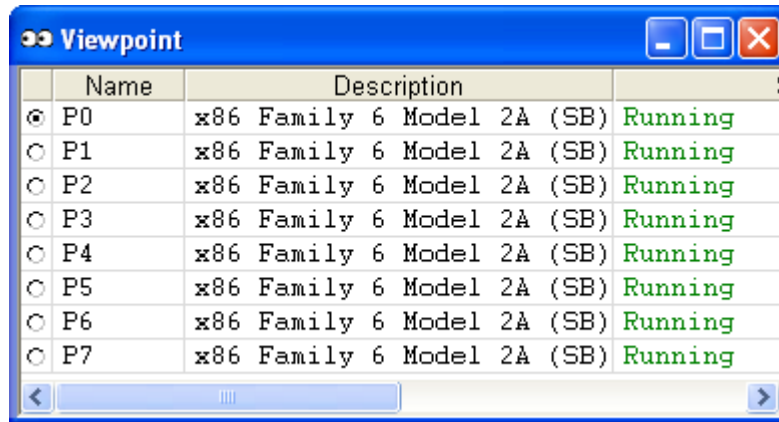
Following is an example of specifying colors. All channels within the master use color 2, except for channel 22, which uses color 3.

```
<master id="0x78" name="SVEN" format="Hosted printf" color="2">
  <channels>
    <channel id="0x20" name="Parrot"/>
    <channel id="0x21" name="Blue"/>
    <channel id="0x22" name="Spam" color="3"/>
  </channels>
</master>
```

Viewpoint Window

Viewpoint Window Introduction

The **Viewpoint** shows the processors in the target system.



	Name	Description	
<input checked="" type="radio"/>	P0	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P1	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P2	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P3	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P4	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P5	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P6	x86 Family 6 Model 2A (SB)	Running
<input type="radio"/>	P7	x86 Family 6 Model 2A (SB)	Running

Viewpoint window

Viewpoint column (no heading). 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.

Name column. The name column displays the processor name. By default, SourcePoint names processors P0, P1, etc. Processor names can be changed in the Target Configuration dialog (Options | Target Configuration | Devices), or by using the vpalias command in the Command window.

Description column. The description column displays a description of the processor.

Status column. The status 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)".

Viewpoint Window Menu

To bring up the **Viewpoint** window context menu, right click on the window.

Set Viewpoint. Set the selected processor as the current viewpoint processor. It performs the same function as clicking on the radio button in the first column.

Viewpoint Tracks Breakpoints (MP targets only). When this menu item is enabled, the viewpoint processor switches automatically to whichever processor hits a breakpoint.

Hide Processors. Opens the Hide Processors dialog to allow particular processors to be hidden from the Viewpoint view.

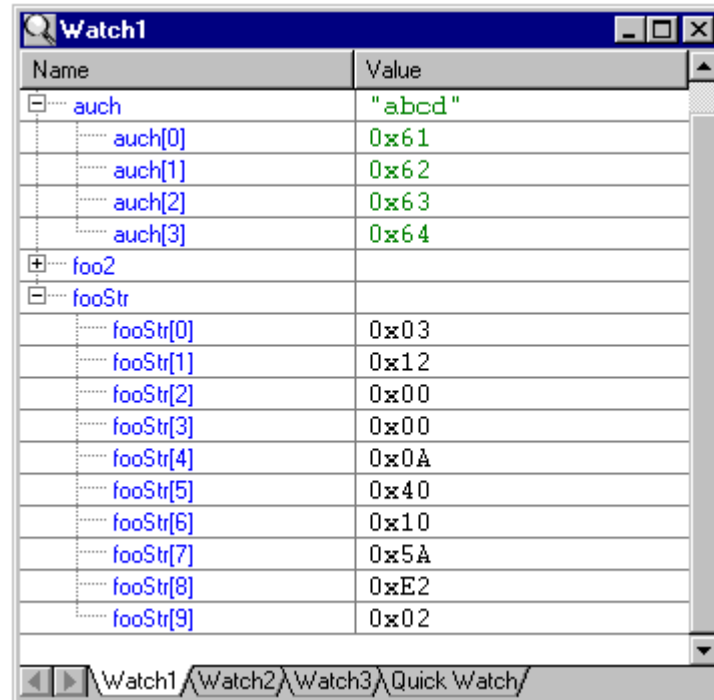
Show Hidden Processors. Forces all processors to be displayed regardless of the settings in the Hide Processors dialog.

Show Sleeping Processors. Displays sleeping processors in the Viewpoint view. If not selected, sleeping processors are automatically hidden from the view.

Watch Window

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 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 node.

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

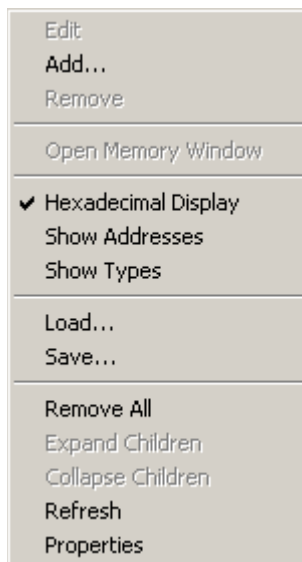
Colors can be changed via the **Colors** tab under **Options|Preferences**.

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.

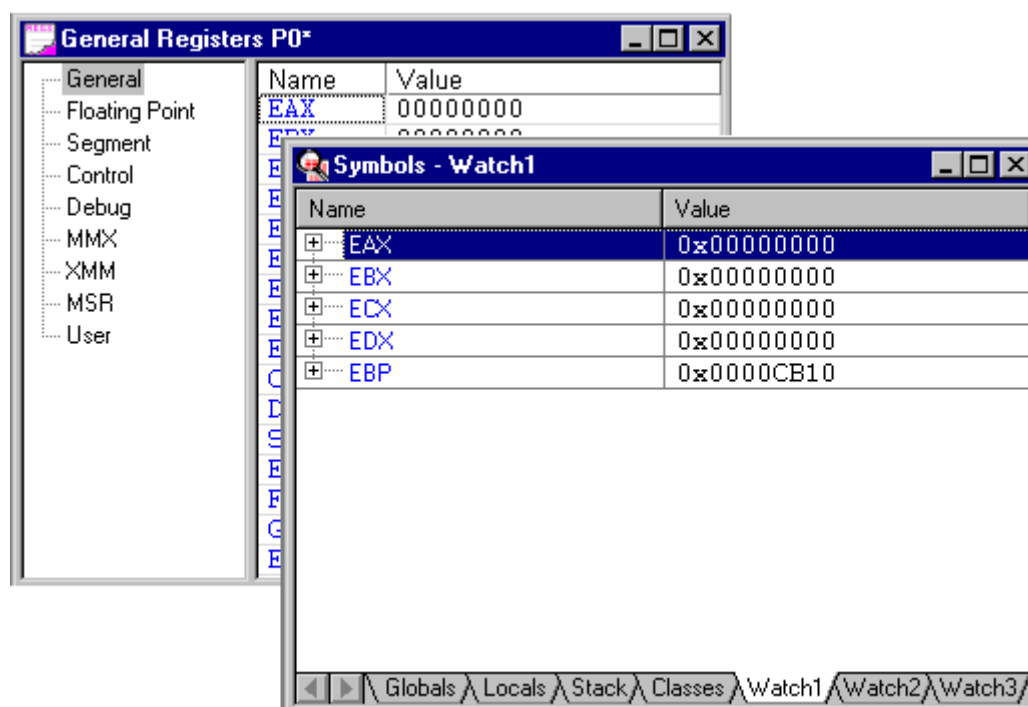
How To - Watch Window

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.

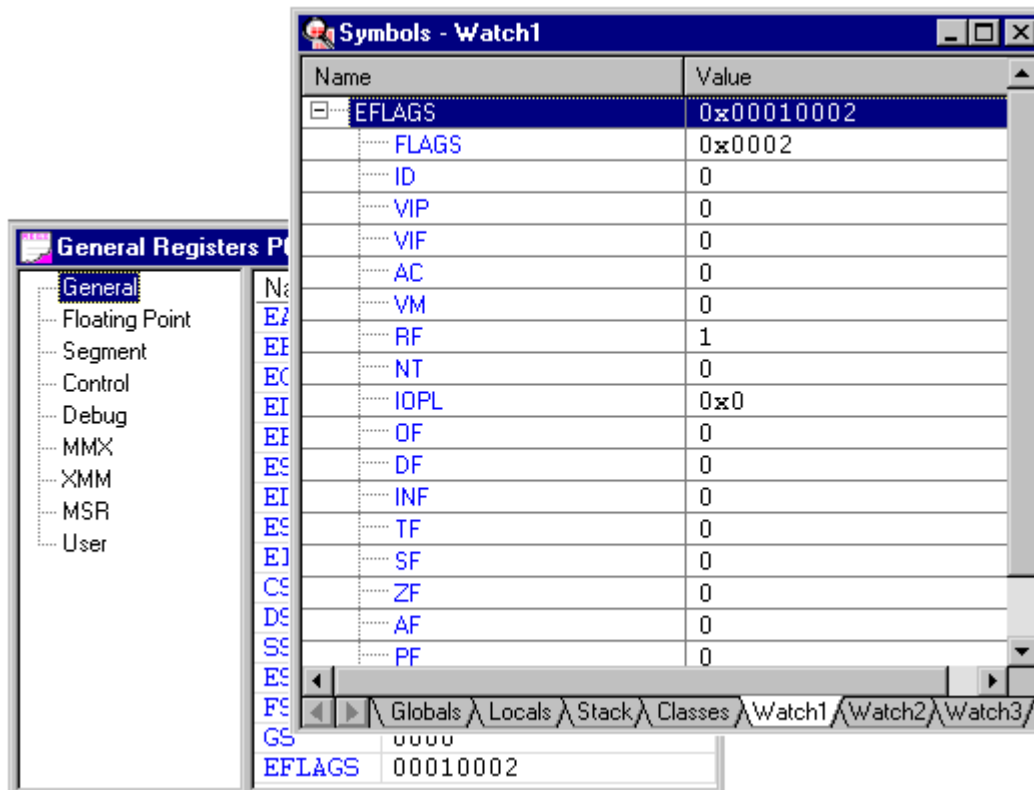
The register and its value move together. The register is fully editable in the **Watch** view. Changes made in the **Watch** view are automatically updated in the **Registers** window.



*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.

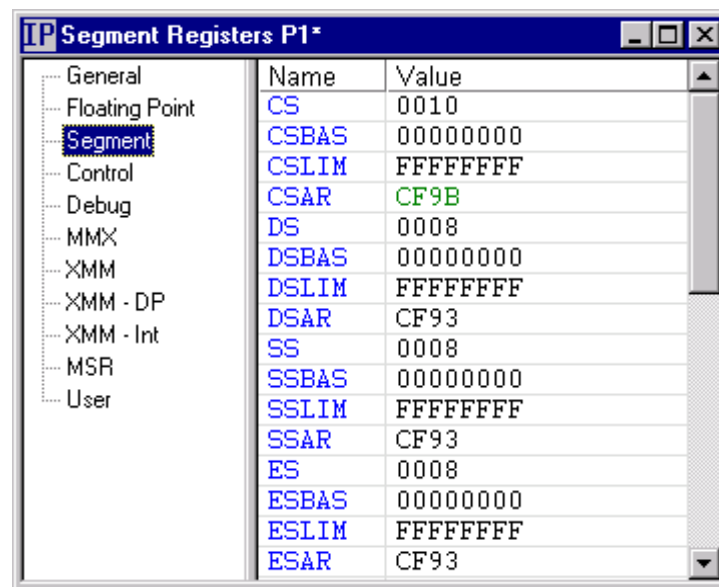
Technical Notes

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.



General	Name	Value
Floating Point	CS	0010
Segment	CSBAS	00000000
Control	CSLIM	FFFFFFFF
Debug	CSAR	CF9B
MMX	DS	0008
XMM	DSBAS	00000000
XMM - DP	DSLIM	FFFFFFFF
XMM - Int	DSAR	CF93
MSR	SS	0008
User	SSBAS	00000000
	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 FFFFFFF0 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 web site 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.

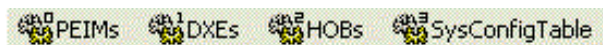
UEFI Macros

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 'cwd' 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.

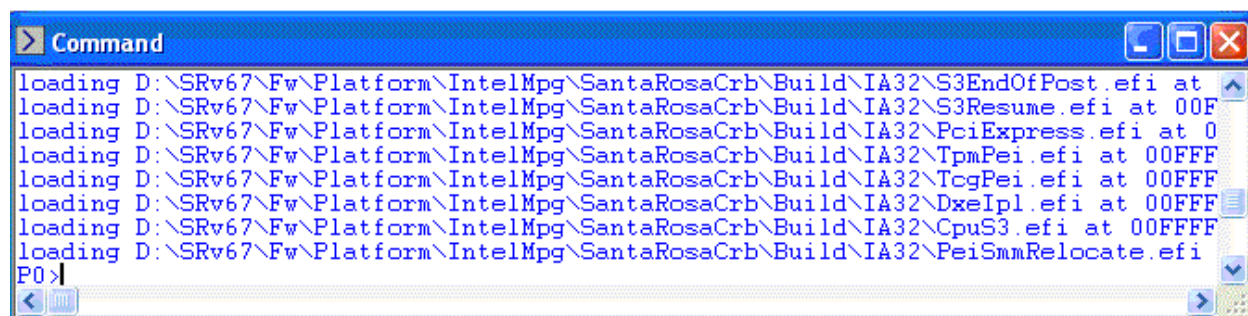
To configure SourcePoint for source-level debugging of PEI code, follow these steps.

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):

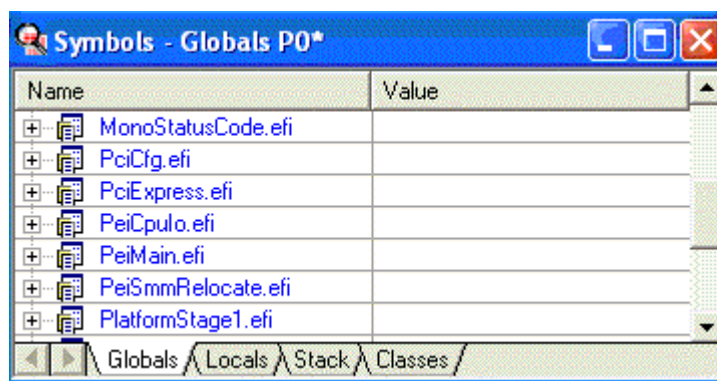
Start	End	Type
00000000P	000FFFFFFP	DRAM
FEF00000P	FFEFFFFFFFFP	SRAM
FFF00000P	FFFFFFFFFP	FLASH

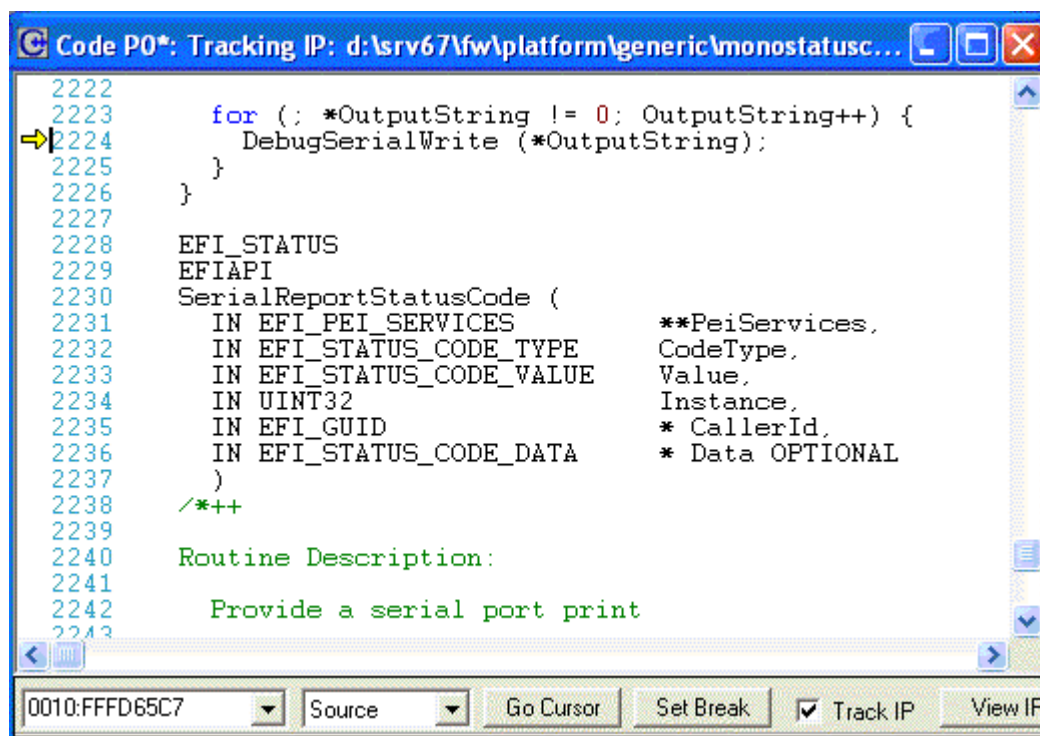
The first entry in the table designates the first 1MB of system memory. The middle entry designates the location of the cache RAM mapped into system memory. The third entry designates the firmware ROM.

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 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.

```

Code P0*: (32-bit) Tracking IP: 0010:00000000 - 0010:FFFFFFFF
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

```

0010:1EE8E54E Mixed Go Cursor Set Break ☒ Track IP View IP Refresh

DXE Code window

HOBs

Open the **Command** window, and then click the HOBs toolbar icon to display the hand-off blocks on the target.

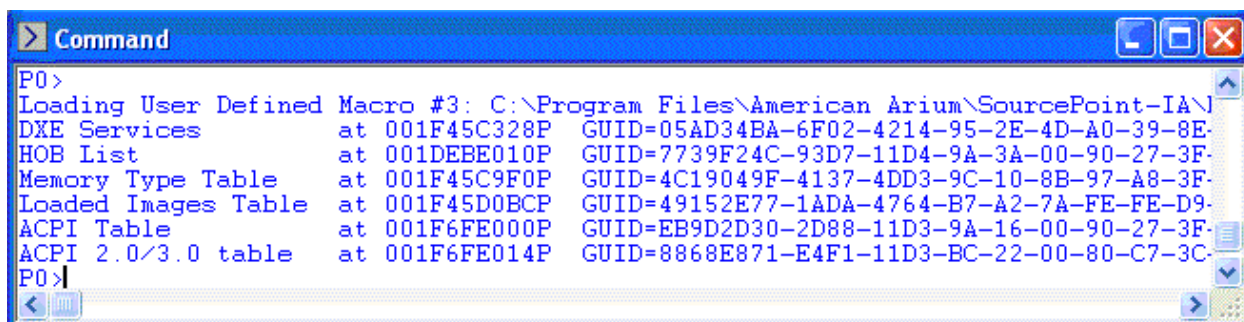
```

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            0x000000000000A0000
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.

*Example of System Configuration Table***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.

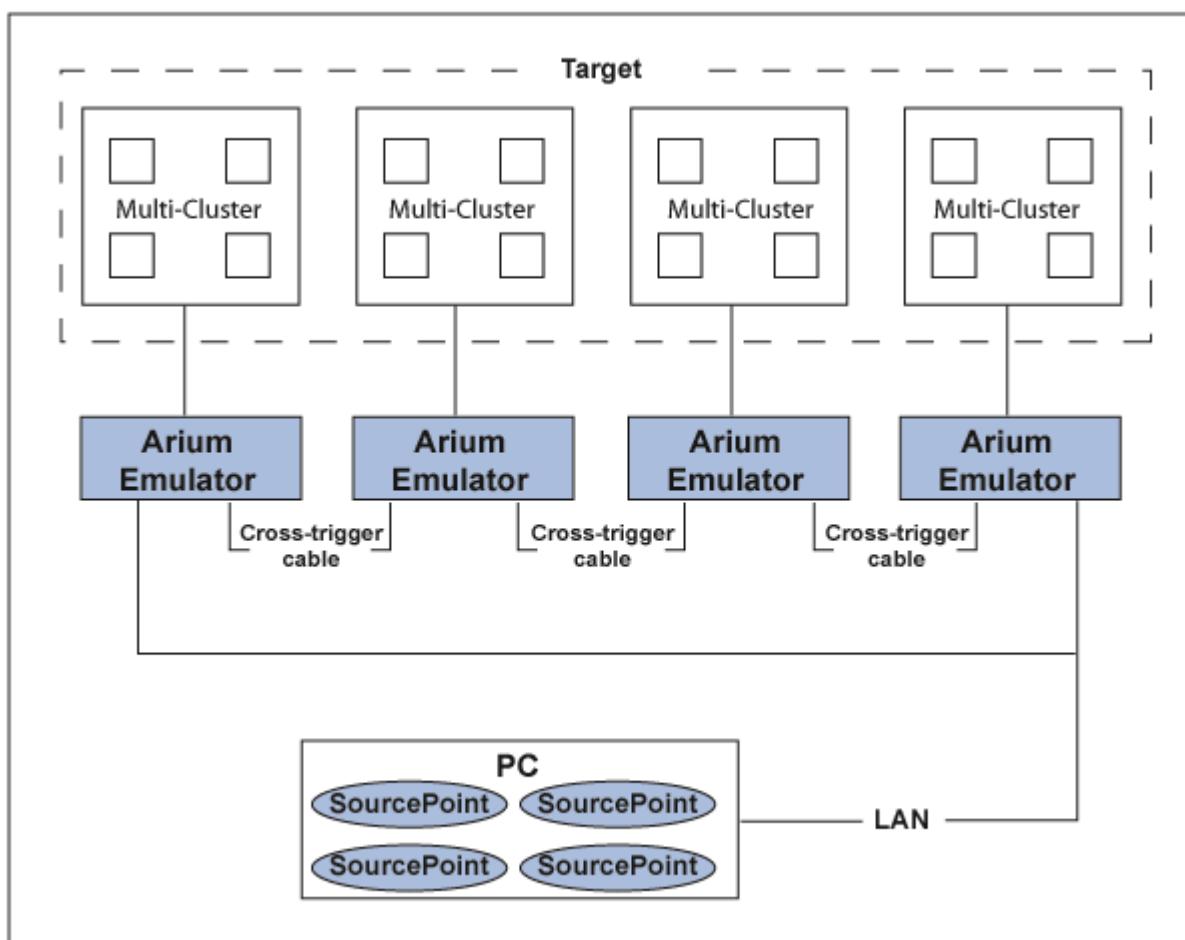
Note: The cable(s) for multi-cluster debug support must be ordered separately. Such cables are not part of the regular emulator package.

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.

Python/CScripts Introduction, Installation and Usage

Introduction

CScripts (Customer Scripts) are a collection of Python scripts provided by Intel to assist customers with platform debug and validation. Asset-InterTech has support in place to run these scripts in its SourcePoint product.

Note: This technical note discusses running CScripts from within the SourcePoint Command view. CScripts can also be run in an external CLI. This has the advantage of supporting tab completion, coloring, and other advanced features. See the [OpenIPC Integration, Installation and Usage](#) for more information.

Requirements

CScripts are designed to be run inside SourcePoint, using its Command view. Therefore, the host system requirements are the same as the requirements and pre-requisites for SourcePoint. Refer to SourcePoint's Installation Guide for these requirements.

As CScripts are written in Python, the host system will need to have Python installed. Current support is in place for 32-bit Python version 2.7. Some scripts are compiled, so CScripts will not function correctly if you have a different version of Python installed on your host.

In case you don't already have a python installation, SourcePoint's installer comes packaged with Python 2.7.5. Allow the installer to maintain the default settings to ensure compatibility.

Installation

CScripts come packaged in a .zip file. This .zip file can be extracted to anywhere on the host system; however, it is recommended that it be extracted into the C:\CS folder. If a different location is chosen, please ensure there are no blank spaces in the folder name. It is a requirement that the directory structure be kept intact when extracting the .zip file, as individual script files reference other files using virtual path names.

Usage in SourcePoint

The SourcePoint Command View is not a separate python command window, rather a python environment integrated inside the application's GUI. Some aspects of SourcePoint python may not behave identically to Intel ITP II's CLI, or to a native python command window, but the CScripts should provide the same behavior.

SourcePoint python retains the concept of a viewpoint processor and supports the Intel ITP II logical node tree and the CScripts devicelist-based logical node tree.

This user's guide describes using SourcePoint on a Microsoft Windows system with USB interface. It does not describe setup of the network interface. The SourcePoint help files have documentation on these topics.

Known current limitations:

SourcePoint does not support pyreadline, and has no tab-completion. Use the `dir()` command with a module name to see names of interest.

Invoking CScripts under SourcePoint

There are currently three ways to load CScripts. Option #1 is easy and recommended for both new and experienced users.

Option #1:

- Locate the root directory of the CScripts and use the mouse drag/drop technique to drag the file `<startup script name>.py` into the SourcePoint Command view. This will switch to python mode, import the CScripts, and display the initial screen.

Option #2:

- In the Command view, enter 'python' to switch to python mode.
- Enter a second prompt to cause the python version to appear, indicating interactive mode python initialization.
- At the prompt, enter (for example) `'arium.runfile("<path>\startivt_oem_arium.py")'` to import the CScripts. The name of the startup script may be slightly different from the above, depending on the version of CScripts being used. If that's the case, just replace the name in the above command with the correct script name.

Option #3:

- In the command view, enter 'python' to switch to python mode.
- Enter a second prompt to cause the python version to appear, indicating interactive mode python initialization.
- At the prompt enter `'arium.runfile()'` and hit return to bring up a windows explorer window, then navigate to the root directory of the CScripts package and double-click on the startup script `start_oem_arium.py` (or similar) to run it.

The various functions of CScripts can now be used as described in the CScripts User Guide from the SourcePoint Command view, noting that tab completion doesn't work, and the display will be embedded in the application UI. The CScripts User Guide can be found in the root directory of the CScripts package.

To exit the session, enter the command `'cscript()'` to exit python mode, then exit to leave SourcePoint.

OpenIPC Integration, Installation and Usage

Introduction

SourcePoint can be integrated with the OpenIPC component of Intel's System Debugger (ISD). This is typically used to run Customer Scripts (CScripts) from a separate Command Line Interface (CLI) window. CScripts are a collection of Python scripts provided by Intel to assist customers with platform debug and validation.

Installation

ISD is not installed with SourcePoint. It must be obtained separately from Intel which requires a signed Non-Disclosure Agreement (NDA).

CScripts are written in Python. Currently CScripts require either Python 2.7 or Python 3.6. Depending on the ISD package version, either Python 2.7, Python 3.6, or both will be installed on the Host PC as part of the installation process. ISD uses 64 Bit versions of Python with OpenIPC, hence, 32 Bit Python is not supported.

Note: Newer ISD installs have removed support for Python 2.7

Usage

There are two usage modes: SourcePoint starts OpenIPC or OpenIPC starts SourcePoint in a tool mode (no visible GUI), which is used as a transport layer to the ASSET emulator hardware (ECM).

SourcePoint Starts OpenIPC

This usage mode is for current SourcePoint customers who are used to running SourcePoint but would also like to be able to open a CLI to run CScripts.

The user starts SourcePoint first and needs to enable OpenIPC (Options | Preferences | IPC). Select the target system type from the drop-down list then and close the options window. The user will then need to open a SourcePoint command window, and type: OpenIPC. SourcePoint will automatically open a CLI and it will become the focus window. At this point, the user can still use the functions of the main SourcePoint GUI.

To run the CScripts, the user can type: `arium.runfile()` as in previous versions of SourcePoint. Once the CScripts are completed, the user can exit from the CScripts, `exit()` and close the CLI window. However, when SourcePoint is closed, it will automatically close the CLI window regardless of exiting the CScripts.

OpenIPC Starts SourcePoint

This usage mode is for customers who normally use the CLI to run CScripts but may have never used SourcePoint.

The user opens the CLI and starts OpenIPC (see Sample Code below). OpenIPC automatically starts SourcePoint in the background. SourcePoint is started in tool mode, where the SourcePoint UI is not visible, and appears as a process rather than as an application.

When the user closes the CLI, OpenIPC automatically closes SourcePoint.

Note: When OpenIPC starts, if it detects SourcePoint is already running, it will connect to that SourcePoint rather than starting a new instance. When the user closes the CLI, OpenIPC will leave SourcePoint running since it didn't start it.

SourcePoint Configuration

Configuration settings are located in the Preferences dialog (Options | Preferences | IPC). The Enable and Target controls in this tab are used when using the "SourcePoint starts IPC" mode. The Update Settings button can be used to enable the "IPC Starts SourcePoint" mode. This tells IPC which version of SourcePoint to run, and which project file to load". See IPC_Tab for more information.

A complete guide on how to install and configure OpenIPC can be seen in Customer Advisory Bulletin #48 on the SourcePoint Community Site (<https://sourcepointhelp.com/>)

Sample Code

The following code can be used to start SourcePoint from the Python CLI, and validate the connection to the target

```
import ipccli
ipc = itp = ipccli.baseaccess( )
ipc.devicelist()
```

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

SourcePoint Licensing

ASSET InterTech uses FLEXIm to license its products. Both hardware (emulators) and software (SourcePoint) are licensed. There are two licensing models, Perpetual and Subscription. The decision of which model to use is made at purchase time.

Perpetual Model

In this mode, you own the hardware and the right to use the software on the purchased version. For the exact definition of what is covered, please refer to the License Agreement. The purchase price includes free SourcePoint updates for a period of one year. To receive additional updates (for bug fixes) and phone/email support, a Star1 service contract must be purchased.

Each emulator comes with a FLEXIm license file. The license file includes the emulator serial number, which high-level features are enabled, and Star1 information. Demo licenses also include an expiration date. SourcePoint can be installed on as many computers as desired, but can only connect to an emulator when it finds a valid license file (one with the correct emulator serial number).

The licenses in these files are "uncounted" (as opposed to a pool of "counted" licenses maintained by a FLEXIm license file server). The Perpetual model does not require a FLEXIm license file server. License files can be stored locally on the machine running SourcePoint, or can be located centrally on a server (not to be confused with a license file server).

Subscription Model

When you purchase a SourcePoint subscription, you have the right to use the software until the subscription expires. Unlike the Perpetual model, SourcePoint will no longer run once a subscription expires.

There are two ways that a subscription can be licensed. The typical method is with a FlexNet (FLEXIm) license file server. The alternate approach is to lock the use of SourcePoint to a particular emulator serial number.

License File Server:

The Subscription model uses two types of license files, one for emulators and one for SourcePoint. The emulator license files are similar to ones used in the Perpetual model. The SourcePoint license file contains the counted (floating) licenses for the SourcePoint software. For example, if you buy 10 emulators and 5 SourcePoint licenses, you will receive 10 emulator license files, and a single license file for SourcePoint. The SourcePoint license file will contain the count of 5, which is the maximum number of SourcePoint instances that the license file server will allow to run concurrently. The SourcePoint license file is used by a FLEXIm license file server to manage usage.

Serial Number Locked License:

Only the emulator license file is used in this case. It contains the serial number of the emulator that SourcePoint is locked to. Additional feature information (that would normally have been obtained from the license file server) is included in this file

In the Perpetual model, emulator license files can optionally enable high-level features. In the Subscription model, most of these features are included in the subscription price. The only items licensed separately in the Subscription model are:

1. ARM and Intel support are licensed separately.
2. ARM flat OS support (e.g., ThreadX) is licensed separately.

Mobile Licensing

There are times when a license file server may not be available. An example of this is carrying a laptop to a customer site where VPN access is not available.

FLEXIm allows the user to "borrow" a floating license from the pool of licenses. This creates a temporary node-locked license that expires after a certain amount of time. When a license is borrowed, the count of licenses available is decremented by one. When the license expires, the count is incremented.

Installing the SourcePoint Vendor Daemon

Please refer to the email you received when the SourcePoint subscription license file was generated. It includes directions for downloading and installing the vendor daemon.

Current License File Information

Select Help | License File to open the FLEXIm License File Information dialog. Refer to [License File dialog](#) for more information.

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>

Target Configuration

Overview

The emulator communicates with the target through the JTAG chain. In order to do this, it needs to know what devices are on the JTAG chain. The collective process of discovering these target devices and configuring them for communication is referred to as "target configuration".

Rather than hard-coding the target configuration process in SourcePoint or the emulator, a target configuration macro file is used. This gives the user maximum flexibility to control the target configuration process. For example, special JTAG commands might be required to unlock a JTAG chain before the emulator can communicate with it. These commands can be inserted in the target configuration macro.

Simple Targets

Simple targets can use the default target configuration files provided with SourcePoint.

Use the New Project Wizard (File | Project | New Project) to create a new project file. Under "Settings basis" select "Use default settings". This creates a project file with the target configuration event macro set to run a debug procedure called Configure. The default version of Configure is in config-utils.mac (in the macros directory). For simple targets it will work unmodified.

Complex Targets

Examples of complex targets include targets that need special commands to unlock debug capabilities, and targets that don't scan properly. On these targets, the default Configure procedure will need to be modified.

Targets that need to be unlocked. The changes required are target-dependent, but usually consist of a series of msgscan commands to send special JTAG commands to the target.

Targets that don't scan properly. In this case, one or more of the scan commands (jtagscan, apscan, and/or devicescan) is omitted. The corresponding SourcePoint configure command is used to send the configuration to the emulator. The configuration typically comes from a special target configuration file created for the particular target, but it can also be manually generated (see below).

Configuration Command Overview

The following is a list of the commands and control variables used in the target configuration process. For more details refer to the separate command topics for each.

Configuration Commands

jtagtest	Test the JTAG chain.
jtagscan	Causes the emulator to scan the JTAG chain for devices.
jtagconfigure	Sends the SourcePoint JTAG configuration to the emulator.
jtagchain	Display or define a JTAG configuration.
verifyjtagconfiguration	Verifies that the SourcePoint and emulator configurations match.

devicescan	Causes the emulator to scan for devices.
deviceconfigure	Sends the SourcePoint device configuration to the emulator.
verifydeviceconfiguration	Verifies that the SourcePoint and emulator configurations match.
devicelist	Displays a list of all configured devices.
autoconfigure	Performs the same function as the default Configure procedure.
disconnect	Disconnect emulator from target (discard current configuration).
reconnect	Reconnect using the target configuration event macro.
uncorescan	Causes the emulator to scan the JTAG chain(s) for uncore devices.
uncoreconfigure	Sends the SourcePoint uncore configuration to the emulator.

Database Commands

jtagdevices	Display the JTAG device database.
jtagdeviceadd	Add a device to the JTAG database.
jtagdeviceclear	Remove a device from the JTAG database.

Control Variables

num_jtag_chains	The number of JTAG chains.
num_jtag_devices	The number of JTAG devices.
num_all_devices	The number of all devices.
num_processors	The number of processors.
emulatorstate	The emulator connection state (0, 1 or 2; see below).
num_uncore_devices	The number of uncores.

Advanced Topics

The JTAG Device Database

SourcePoint maintains a database of known JTAG devices in `targets\jtag-devices.xml`.

The Configuration view is used to display and edit the database. It is accessible by pressing the Configuration button in Options | Target Configuration | Target Devices, and then selecting JTAG Info.

The command language also supports viewing and modifying the database. The `jtagdevices` command displays the database. The `jtagdeviceadd` and `jtagdeviceclear` commands add and remove devices.

If you install a new version of SourcePoint, it may include a newer version of the database file. In this case SourcePoint merges the old and new files, so any user changes to the database are not lost.

Manually Defining the JTAG chain

If the JTAG chain cannot be scanned, and the target you are trying to connect to does not have a target configuration file, then it is possible to manually define the JTAG chain configuration. This can be

accomplished in the JTAG Configuration tab in the Configuration view. All devices must already exist in the JTAG device database (the JTAG info tab).

The command language also supports defining the JTAG chain configuration. The `jtagchain` command displays or sets the devices on the JTAG chain. The `jtagchainclear` command clears the devices on the JTAG chain.

What the Configure Procedure Does

The Configure debug procedure (in `macros\config-utils.mac`) is the default target configuration method. It will work for most targets. Configuration consists of 2 or 3 phases.

The first phase uses the `jtagscan` and `jtagconfigure` commands to scan and configure the JTAG chain. The `jtagscan` command automatically performs a `jtagtest` command prior to the scan. After JTAG configuration is complete the `emulatorstate` control variable transitions from state 0 (disconnected) to state 1 (JTAG configured).

On targets with uncore devices, the next phase uses the `uncorescan` and `uncoreconfigure` commands to scan and configure all the uncore devices on the JTAG chains.

The last phase uses the `devicescan` and `deviceconfigure` commands to scan and configure devices. After successful device configuration, the `emulatorstate` control variable transitions from state 1 (JTAG configured) to state 2 (fully configured). The target is now ready for run control.

Manually Executing Configuration Commands and Creating a Target Configuration File

Connecting to a new kind of target for the first time can be a tricky process. The target may have hardware issues, software issues, security issues, or a whole host of other problems that cannot be anticipated.

By using a special target configuration file (`manual.tc`), and the command line commands listed in previous sections, it is possible for a user to manually discover the target, step by step.

This has the huge benefit of allowing the user to pinpoint exactly where in the connection process that the target is providing difficulties to the standard connection process.

The following describes the general process for manually discovering and configuring a new target for the first time. If there are external requirements by the target (security unlock, physical JTAG chain configuration, etc) at various stages during the process, it is the user's job to know what these requirements are, how to carry them out, and when to cause those processes to occur. It is not possible for SourcePoint to know the implementation-specific requirements of any given target, unless Arium has been previously informed of said requirements.

The emulator should be powered up and connected to the target, which should also be powered up. The user should then create a new project file with the "manual.tc" file that is included in the SourcePoint distribution. Tell SourcePoint the address of the emulator (TCP/IP or USB).

SourcePoint will connect to the emulator, and stop. No further automated configuration actions will take place on the part of SourcePoint. From this point on, all steps will be carried out manually by the user in the Command window. At this point, the emulator is in the disconnected from the target state (`emulatorstate` control variable = 0).

The user now has the option of using the `jtagtest` or the `jtagscan` command. The `jtagtest` command may be used to manually run any one of the six JTAG tests individually. These tests are:

- 1) Test for power currently on.
- 2) Test for reset currently asserted.
- 3) Determine the IR length of the JTAG chain.
- 4) Scan the JTAG ID's on the JTAG chain.
- 5) Check the integrity of the JTAG chain.
- 6) Check whether the target supports Adaptive TCK.

If these tests are run one at a time via the `jtagtest` command, the user can determine, with very fine granularity, what an exact problem may be with the JTAG chain. **Note:** Enable logging with `aalog=20987` to view the test results in the Log window.

The `jtagscan` command runs tests 1 - 5 above automatically, in a predefined order, simulating the automated test that was run in older versions of firmware. This may be done on targets whose JTAG chain the user has confidence in.

Once the `jtagscan` command has been run successfully, SourcePoint will have sufficient information (i.e. the list of JTAG ID's) to proceed to the next step. This is to run the `jtagconfigure` command at the command line.

Once this command has been issued, the firmware in the emulator will be in the JTAG Configured state (emulatorstate control variable = 1), and will be ready to proceed to the next step.

In the case of a target with an uncore, the user should now use the `uncorescan` command. This causes the firmware to discover uncores on the JTAG chain. After this has been successfully completed, the `uncoreconfigure` command should be issued. This prepares the firmware for the final stage of configuration.

The user should issue the `devicescan` command. This causes the emulator to discover what "devices" are in the system (i.e. processors, ETM's, etc). That information is reported to SourcePoint.

After this information has been discovered, the user should issue the `deviceconfigure` command. When this is done, the STS light on the front of the emulator should turn on. The emulator is now fully configured (emulatorstate control variable = 2)

The last stage is to allow SourcePoint to configure itself by issuing the `connect` command. After a short delay, SourcePoint should become connected to the emulator and display its debug windows.

SourcePoint and emulator are now fully configured and should be able to debug the target.

The project file should be saved, and a configuration macro should be created (or an existing file should be modified) to allow SourcePoint to carry out automated connection from now on. This configuration file should include any target specific actions that are required to activate or connect to the target, in the correct locations in the macro file.

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**, and **Command** 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** and **Log** 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.

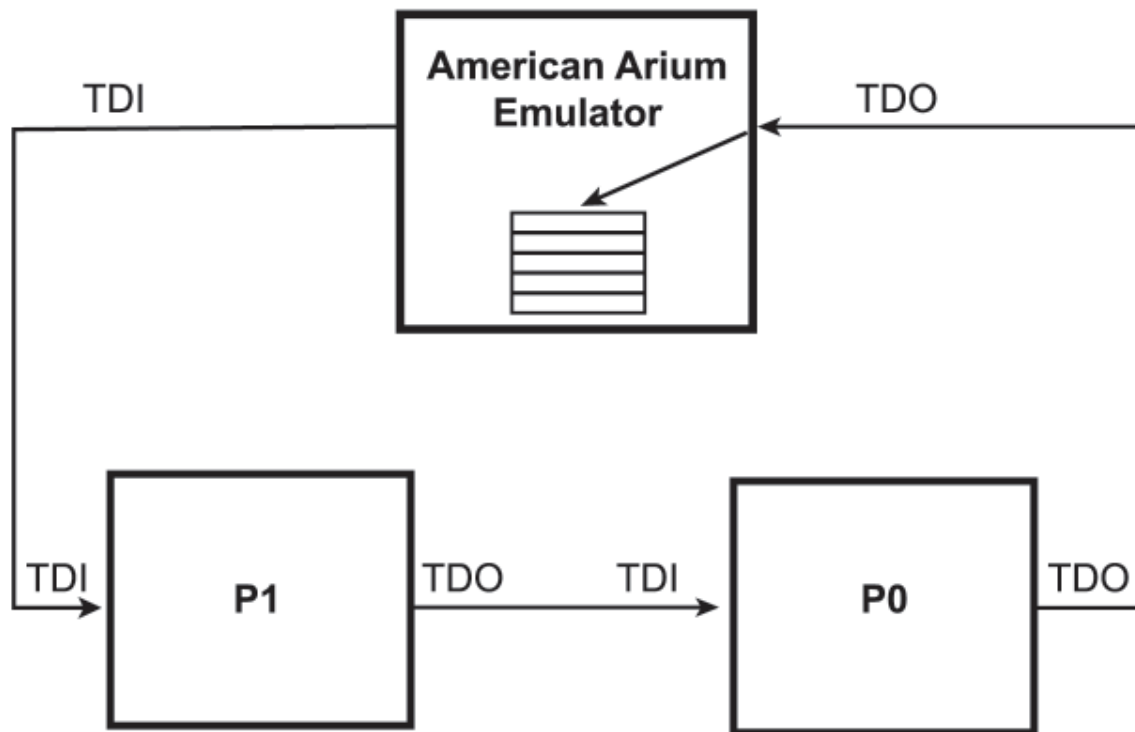


Figure 1

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.

Getting Started with DbC

Overview

The Direct Connect Interface (DCI) is an emerging transport technology for closed-chassis debug access. The primary purpose of DCI is to allow existing functional I/O (e.g., a USB port) to be used for debug of the system and/or silicon, and to gain access to trace/debug features in the silicon. DCI DbC allows target debug with a single USB connection between the host computer and the target. No debug probe is required. This document explains the steps necessary to get DbC up and running.

There are two forms of DCI DbC, one that uses USB 2.0 (DCI DbC2), and one that uses USB 3.0 (DCI DbC3). For run control purposes the speed of either is sufficient. DCI DbC2 has the advantage of allowing debug through resets and power cycles. Throughout the rest of this document DbC will be used to indicate DCI DbC2.

Preparing the Firmware image

A firmware image needs to be modified to configure the silicon (PCH) for the appropriate USB port which will be used for DbC. This is achieved by using the Intel® FIT tool and the firmware image. The default setting of the FIT tool is to not enable any ports for DbC.

There are two ways to modify a firmware image. Automatically by an XML during the build process, or manually by loading the firmware image into the FIT tool after the build process has completed. This Getting Started guide will concentrate on using the manual method for modifying an existing image.

The end user will need access to the target user guide or schematics to look up the connectivity of the physical USB port from the silicon to the required USB connector on the target. For example, a target could have a rear USB connector, J1, which is wired to the PCH USB2.0 Port 5. In this case, the FIT tool is configured to use USB2.0 Port 5

NOTE: It should be noted that the FIT tool version used to enable DbC should be the same version (or newer) as the existing firmware image.

To enable DbC for a specified USB2.0 port:

1. Open the Intel FIT tool
2. Load an existing firmware image (File->Open)
3. The FIT tool will create a log view within the GUI. The details of the version used to create the firmware image will be displayed:

```
FIT version used to build the image: 12.0.0.1051
```

4. Ensure the FIT tool used to enable DbC is a version that is the same (or greater) than the version stated in (3)
5. In the left hand tree view, scroll down to the Debug section, and click
6. In the right-hand tree view, scroll down to the Direct Connect Interface Configuration section

- a. Ensure that Direct Connect Interface (DCI) Enabled is set to 'Yes'
7. In the right-hand tree view, scroll down to the Early USB2 DbC over Type-A Configuration section
 - a. Ensure that USB2 DbC port Enable is set to the USB port required for the physical connector on the target to be used for DbC. For example, 'USB2 Port 1'
8. Create the new image (Build->Build Image or CTRL-B)
 - a. If prompted that Boot Guard is disabled on the platform, then click 'Yes' to accept and proceed
9. A new firmware image called 'outimage.bin' will be created. Rename this image appropriately, and program onto the target.

Enabling DbC in BIOS

The Debug Consent setting in the BIOS must be set to enable DbC debug.

1. Type 'exit' at the EFI prompt to enter BIOS setup (or press F8 if booting to an OS).
2. Select Intel Advanced Menu
3. Select Debug Settings
4. Change the Platform Debug Consent setting to either 'USB2 DbC' or 'DCI OOB+[DbC]'.
5. Save the settings and exit.

Note: These steps may vary between different versions of BIOS.

Connecting the Host to the Target

There are two types of USB cables that can be used:

1. USB3.x Type-A to Type-A
2. USB3.x Type-A to Type-C.

Caution: These are not standard USB cables. They are special cables with no VBus connection. Using a standard cable may damage the host computer and/or the target. Which cable you need depends on whether you enable DbC (in the BIOS image) on a Type-A or Type-C connector.

These cables are available at Intel's Design-In Store (<https://designintools.intel.com>). Following are the part numbers. Intel recommends the 1 Meter cables, but in most instances the 1.8 Meter cables work fine.

Part Number	Description
ITPDCIAMAM1M	C01 - Intel SVT DCI DbC2/3 A-to-A Debug Cable 1 Meter
ITPDCIAMAM2M	C09 - Intel SVT DCI DbC2/3 A-to-A Debug Cable 1.8 Meter
ITPDCIAMCM1MU	C06 - Intel SVT DCI DbC2/3 A-to-C UFP Debug Cable 1 Meter

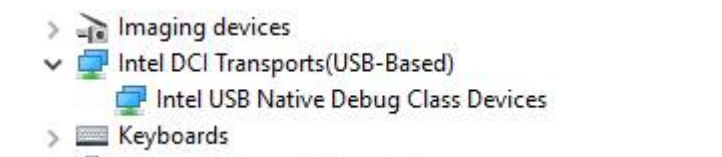
ITPDCIAMCM2MU

C12 - Intel SVT DCI DbC2/3 A-to-C UFP Debug Cable 1.8 Meter

Connect the Type-A end of the cable to any USB2 or USB3 connector on the host computer. Connect the other end to whichever USB port on the target has DbC enabled.

Installing the Intel DCI Driver

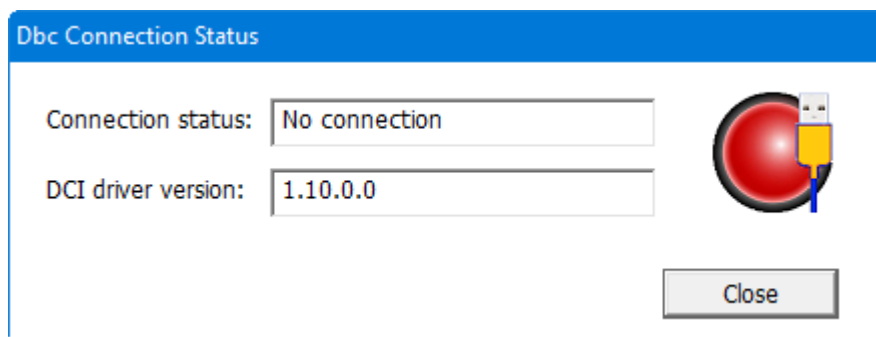
The Intel DCI driver is installed automatically when SourcePoint is installed. When active (the host is connected to a target with DbC enabled), the following driver entry will be present in Device Manager:



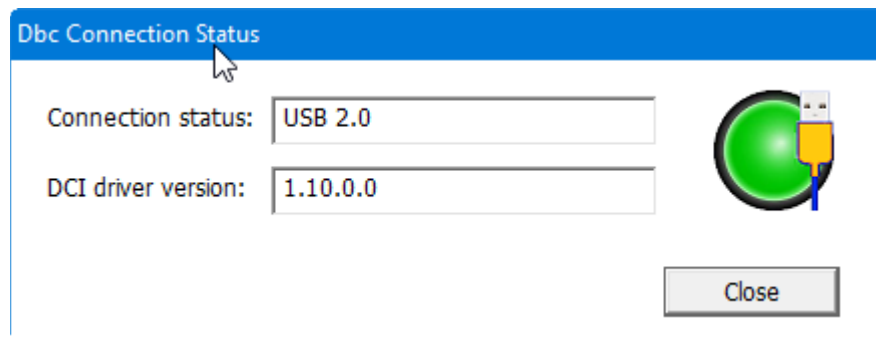
Testing the Connection

Included in the SourcePoint installation is a DbC debug utility that can be used to determine if a valid DbC connection has been made. It can be found in the same folder as sp.exe and is called DbCStatus.exe.

1. Power down the target.
2. Connect the USB cable between the host and target.
3. Start the DbC Indicator program. It should show red for no connection:



4. Power on the target.
5. The DbC Indicator program should now show green for a USB2 connection (you'll also hear the Windows USB device connected sound).

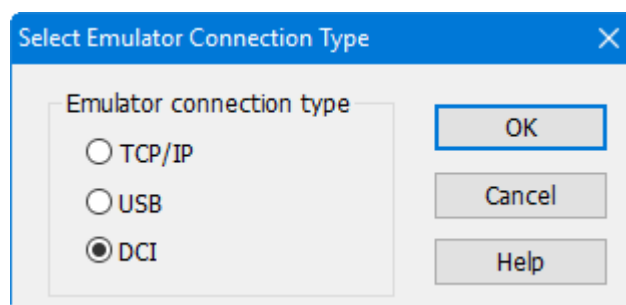


6. Close the DbC indicator program.

Connecting with SourcePoint

SourcePoint behaves the same whether it's connected to a debug probe (e.g., ECM-XDP3e), or directly to the target via DbC. The only difference is the emulator connection type. With a debug probe, you create a TCP/IP or USB connection, with DbC you create a DCI connection type.

1. Install SourcePoint.
2. Start SourcePoint.
3. The New Project Wizard will start.
 - a. Specify the connection type as 'DCI'.
 - b. Point to the target configuration file for the type of target you're using.



SourcePoint Command Language

Introduction

This manual describes the SourcePoint command language. The command language is very similar to the C language, with additional commands added for run control, target access, etc.

Commands can be typed one at a time in the Command window, or multiple commands can be executed from a command file.

The Command window interface is described in the Command Window section of the User's Guide.

Syntax Notation

GeneralGeneral

command	verbatim text (case insensitive)
<i>italics</i>	user-provided parameter
[item]	item is optional
{item}* [item]+	0 or more instances of item 1 or more instances of item
{item1 item 2} [item1 items 2]	either item1 or item2 must be selected either item1 or item2 may be selected
...	indicates that the preceding item can be repeated
punctuation	must be entered exactly as shown except for {}, [] and .

SourcePoint specific

[px]	is the viewpoint override, including punctuation ([]), specifying that the viewpoint is temporarily set for this command to the specified processor. The processor can be specified as px (where x is the processor ID), or an alias you have defined for a given processor ID (using vpalias command). See Viewpoint Processor and Processor Overrides .
[all]	is a special viewpoint override specifying that all processors are affected. The brackets are required punctuation
expr	is an expression as described in expressions

Comments

SourcePoint supports both old-style C comments (`/*...*/`), and new-style C comments (`//`).

Examples

```
go           // this is a comment
wait        /* this is another comment */
```

Constants

Integer Constants

Integer constants are of the form:

```
[0y | 0Y]{0-1}+[y | Y]           // binary constant
[0o | 0O]{0-7}+[q | Q]           // octal constant
[0n | 0N]{0-9}+[t | T]           // decimal constant
[0x | 0X]{0-9 | a-f | A-F}+[h | H] // hex constant
{0-9}+{k | K}                     // Kb
{0-9}+{m | M}                     // Mb
```

In the absence of an explicit base prefix or suffix, the default number base is specified by the [base](#) control variable.

Examples

```
10t           // constant is decimal and has value 10
0n10          // constant is decimal and has value 10
10y           // constant is binary and has value 2

base = hex
10            // constant has value 16 decimal

base = dec
10            // constant has value 10 decimal
```

Floating Point Constants

Floating point constants are of the form:

```
[+ | -]{Digit}+.{Digit}*[Exp]
[+ | -]{Digit}*.{Digit}+[Exp]
[+ | -]{Digit}+[Exp]
```

Where:

```
Digit    [0-9]

Exp       [E | e][- | +]{Digit}+
```

Examples

```
.234
-1.1234
123.45e3
10e-5
```

Character Constants

Character constants follow the C language convention.

Examples

```
'a'           // value = 0x61
'\t'          // tab character
'\'           // backslash character
'\377'        // octal value 377 = 255 decimal
```

String Constants

String constants follow the C language convention. Constants longer than 256 characters are truncated.

Examples

```
"abcd"        // abcd
"ab\\cd"       // ab\cd
"abcd\n"       // abcd + newline character
```

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:

```
ord2 1000 len 10
```

Result:

```
00001000 0080 8D01 42B9 D00A F8B4 03B8 EB04 1000  
00001010 F500 712E F3C1 018F F8A0 12A8 E008 F8B4
```

Expressions

Expressions consist of one or more operands combined with operators. The assignment operator may be used once in an expression

Operands

Valid operands include constants, control variables, debug variables, debug procedures, symbolic references (user-program symbols), register names, and memory accesses.

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.

Emulator Operators in Order of Precedence			
Category	Symbol	Associate	Function
primary	()	left	group expressions
	[]	left	index into string
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
	*	left	multiplication
	/	left	division
binary	%	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	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

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.

Debug Variables

There are two types of variables: control variables and debug variables.

Control variables are predefined variables in SourcePoint. See [Control Variables](#) for a list of these variables and links to their individual help topics.

Debug variables are defined by the user with the [define](#) command. They can be displayed with the [show](#) command, and removed with the [remove](#) command.

Debug variable types include integers, reals, strings, and pointers (for accessing target memory). Pointers to debug variables are not supported.

Debug variables arrays are supported (see [Debug Variable Arrays](#)). Debug variable structures are not supported.

Debug variable names are case sensitive. Names are of the form:

```
{Letter}[Letter | Digit]*
```

Where:

Digit [0-9]

Letter [a-zA-Z_@]

Examples:

```
define ord4 x1 = 100h           // 32-bit unsigned integer variable
define int16 y                  // 16-bit signed integer variable
define nstring foo = "abcd"     // string variable
define ptr addr = 0x1000        // pointer variable into target memory
define nstring names[10t]       // 10 element array of strings
define real8 percent = 0.4      // 64-bit floating point variable
define bool bEnable = false     // Boolean variable
```


Debug Variable Arrays

The [define](#) command is used to create debug variable and debug variable arrays. 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.

Debug Procedures

Debug procedures are the equivalent to functions in the C language. When a debug procedure is defined, it is saved in memory for later execution. Debug procedures can accept arguments and return values.

The [define](#) command is used to define a debug procedure. The [show](#) command lists debug procedures, and the [remove](#) command can be used to remove debug procedures. The [proc](#) command can be used to display a debug procedure definition.

Typically, a command file is loaded that contains one or more debug procedure definitions. The user can type a procedure name at the command line to execute it, or assign the procedure to a user-defined toolbar button, and press the button to execute it.

Syntax

```
define proc [data-type] proc-name ([argument-name][,...])
[define argument-type argument-name] [...]
{
    commands [...]

    [return expr]
}
```

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).
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 [proc](#) 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 an argument name is specified but not an the argument type, 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 (see [define](#)). 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. If the return data type does not match the calling data type, then an explicit data type conversion occurs. If a return datatype is not specified, then the type comes from the value returned.

If a proc executes an emulation command (such as [go](#) or [step](#)), 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.

Example 1

To define and then execute a procedure named avg that accepts three parameters and returns their average:

Note: Types are not specified for a, b and c, so the caller's data type is assumed.

Command input:

```
define proc avg(a,b,c)
{
    return ((a + b + c) / 3)
}
avg(4, 6, 3)
```

Result:

4T

Example 2

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)
}

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:

24T

Example 3

To use the forward option to create a recursive procedure:

Command input:

```

define proc ord4 factorial (n)
define ord4 n
{
    forward proc ord4 factorial                    // recursive proc
                                                    // forward reference
    if (n == 0)
        return 1
    else
        return (n * factorial(n-1))
}

base = 10t
factorial (4)

```

Result:

24T

Example 4

A return data type is not specified, so the type comes from the value returned.

Command input:

```
define proc truefalse(b)
{
    if(b) {
        return "true"
    } else {
        return "false"
    }
}
```

Related Topics

[define](#)

Control Variables

Control variables are predefined debug variables that are always defined in the SourcePoint Command language.

Examples

```
base // display the base control variable
base=10t // set the display base to decimal
define ord4 svbase=base // save the current display base
```

Control variable names are case insensitive. Following is a list of control variables, and their attributes.

Control variable	Type	Read/Write	Options	Default
advanced	bool	rw	true, false	true
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
issleeping	bool	r	true, false	n/a
issmm	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_activeprocessors	ord4	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
num_uncore_devices	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

Command Files

Command files are text files containing multiple commands. Creating command files helps to automate oft repeated operations. Command files are also referred to as macro files, script files or include files. There are several ways to execute a command file:

1. Use the [include](#) command in the Command window.
2. Drag and drop a command file from Windows Explorer to the Command window.
3. Select **File | Macro | Load Macro** from the main menu.
4. Select **File | Macro | Configure Macros** to attach a command file to a user-defined toolbar button, and then press the button.
5. Select **File | Macro | Configure Macros** to attach a command file to an event. Examples of events include: go, stop, project load, power cycle, etc. When the event occurs the macro will automatically execute.
6. Define a breakpoint and specify a command file to execute when the breakpoint hits.

Recently executed macro files are shown in **File | Recent Macros**. Selecting a command file from this list will re-execute the file. Breakpoint and event macros are excluded from this list.

When a command file is executing, the name of the file is shown in the SourcePoint Status bar (at the bottom of the SourcePoint window).

Note: When SourcePoint finishes executing a command from a file, it immediately begins to execute the next command. In the case of the [go](#) command this may not be desired. To delay execution of the command file until a breakpoint hits, you must use the [wait](#) command.

Filenames

Many commands take a filename as an argument. A filename can be specified as a string constant or by an nstring debug variable. Filenames with spaces must be enclosed in quotation marks.

Filenames can be entered with absolute paths or with relative paths. SourcePoint takes a relative path and converts it to an absolute path. There are two methods used:

1. If the command is typed into the Command view, then SourcePoint uses the [defaultpath](#) control variable as the base portion of the filename.
2. If the command is part of a macro file, then SourcePoint uses the [macropath](#) control variable as the base portion of the filename.

Viewpoint Processor and Processor Overrides

The following applies to multi-processor targets only.

Viewpoint Processor

The viewpoint processor is an application-wide setting that indicates the default processor to use when none is specified.

There are several ways to display and/or set the viewpoint processor:

1. The command line prompt in the Command view displays the viewpoint (e.g., P0>).
2. The Status Bar at the bottom of the SourcePoint main window shows the current viewpoint processor.
3. The view control variable can be used to display or set the current viewpoint.
4. The viewpoint view displays and changes the current viewpoint.

There are several windows that display data from a particular processor (Code, Memory, Registers, etc.). These views all have a viewpoint submenu that allows a particular processor to be specified. In addition, these views can be configured to track the viewpoint processor, so that when the viewpoint is changed, the window will automatically switch to displaying data from the new viewpoint.

Processor Overrides

There are numerous commands that affect a single processor (e.g., read/write a processor register, read/write processor memory, go, stop, step, etc.). By default, the viewpoint processor is used.

A processor override can be specified to cause the command to act on a different processor than the current viewpoint. The override is a prefix of the form [px].

Examples

```
stop           // stop the viewpoint processor.
[p1]stop       // stop P1
ord4 0x100     // read 4 bytes of memory at address 0x100 (viewpoint)
[p2]ord4 0x100 // read 4 bytes of memory at address 0x100 (P2)
[p0]pc         // read the PC from processor P0
```

Certain commands can also use the All processor override (e.g., [all]stop stops all processors).

Processor Numbering

Processors are numbered P0, P1, ... Pn depending on the number of processors in the target. P0 is the first processor on the JTAG chain, P1 is the next, etc.

Processor names can be changed to more meaningful names in **Options | Target Configuration | Devices**.

Symbolic References

References to program addresses and variables

Syntax:

```
label
procedure
variable
variable[array-expr]
composite-variable.member
composite-variable->member
compound-variable
variable-ref=expr
*ptr-variable
&variable
```

Where:

<i>label</i>	program label
<i>procedure</i>	procedure name
<i>variable</i>	variable name
<i>composite-variable</i>	structure or union name
<i>member</i>	structure or union member name
<i>compound-variable</i>	a combination of other variable types
<i>ptr-variable</i>	pointer variable name
<i>[array-expr]</i>	specifies a number or expression identifying an element in an array
<i>variable-ref</i>	specifies a variable, an array variable, a composite variable or a compound-variable
<i>expr</i>	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 (\). If a name exists more than once in a program, see [Qualified Symbol Names](#).

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]: 0x00000000
ai[1]: 0x00000001
```

```
ai[2]: 0x00000002
ai[3]: 0x00000003
ai[4]: 0x00000004
ai[5]: 0x00000005
```

Command input:

```
ai[4] = 0
ai[4]
```

Result:

```
0x00000000
```

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
```

Result:

```

IntArray[0]:
a: -1
b: -2
c: -3

IntArray[1]:
a: 1
b: 2
c: 3

```

Command input:

```

IntArray[1].b = -5      // change one member of one element
IntArray[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 members a, b, and c

```

Result:

```

a: 0
b: 5
c: 0

```

Command input:

```

&ints                    // display address of ints

```

Result:

```

00000C78

```

Command input:

```

pInts                    // pInts points to int

```

Result:

00000C78

Command input:

```
*pInts                // display ints through the pointer pInts
```

Result:

```
a: 0  
b: 5  
c: 0
```

Command input:

```
pInts->b = 0           // change a member of ints through pInts
```

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.

Qualified Symbol Names

Resolve ambiguities on a symbol file reference

Syntax

```
[\][::program][:module.]symbol
```

Where

\	force symbol name lookup before keyword lookup
program	specifies the program name containing the symbol
module	specifies the module name containing the symbol
symbol	specifies any symbolic reference expression

Discussion

When SourcePoint looks up a name it uses the current program scope. If the symbol name is not found, it continues looking in containing scopes trying to find the symbol. It's possible there may be more than one instance of a symbol name. This can occur when there is static data with the same name in two different modules. It can also occur when multiple programs are loaded, and more than one program has the same symbol name.

The qualified symbol name syntax allows SourcePoint to references the correct symbol.

If you have a symbol name that conflicts with a SourcePoint keyword, preface the name with the '\` character. This forces SourcePoint to assume the name is a program symbol rather than a keyword

Example 1

To display a structure ints found in the module csample:

Command Input:

```
:csample.ints
```

Result:

```
a: 1
b: 2
c: 3
```

Example 2

To display a structure ints found in the module csample in the program flat:

Command Input:

```
::flat:csample.ints
```

Result:

a: 1
b: 2
c: 3

Related Topics

[Symbolic References](#)

Commands and Control Variables

aadump

Display the current configuration of SourcePoint and the emulator.

Syntax

```
[result =] aadump([filename])
```

Where:

<i>filename</i>	specifies a filename. See Filenames for details.
<i>result</i>	specifies an nstring variable 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.

Discussion

The aadump function displays the current settings in SourcePoint and the emulator. The output can optionally be written to a file.

Example 1

To save output to an nstring variable:

Command input:

```
define nstring foo=aadump()
```

Example 2

To display output in the Command window:

Command input:

```
aadump()
```

Example 3

To save output to a file called "dump.txt":

Command input:

```
aadump("dump.txt")
```

abort

Abort command file processing.

Syntax

```
abort
```

Discussion

The abort command aborts command file processing. If command file execution is nested (nested include commands), all command files are terminated.

Example

To conditionally abort command file processing:

Command input:

```
if (getchar()=='x') abort
```

Related Topics:

[include](#)

abs

Return the absolute value of an expression.

Syntax

```
[result =] abs(expr)
```

Where:

result specifies a debug object 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.

expr specifies a number or an expression that evaluates to an integer or real number.

Discussion

The `abs` function returns the absolute value of an expression.

Note: Values returned by this 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.

Example 1

To take the absolute value of a number and display it on the Command line:

Command input:

```
define int4 myVar = -23
abs(myVar)
```

Result:

```
00000023H
```

Example 2

To take the absolute value of a real number and assign it to a debug variable:

Command input:

```
define real8 var1 = -1.23
define real8 var2 = abs(var1)
var2
```

Result:

```
1.23
```


acos

Return the arc cosine of an expression.

Syntax

```
[result =] acos(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8 evaluated in radians.

Discussion

The acos function 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).

Note: Values returned by this 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.

Example

Command input:

```
acos(-1)
```

Result:

```
3.14159
```

Related Topics:

[asin](#)
[atan](#)
[atan2](#)
[cos](#)
[sin](#)
[sqrt](#)

advanced

Display or change the advanced mode setting.

Syntax

```
advanced [= true | false]
```

Where:

false	advanced mode disabled.
true	advanced mode enabled.

Discussion

The advanced control variable enables and disables advanced mode. When advanced mode is enabled all configuration settings in SourcePoint are available. When advanced mode is disabled only the most commonly used settings are displayed / enabled. This control variable has the same effect as changing the checkbox in Options | Preferences | General.

Example 1

To display the current advanced mode state:

Command input:

```
printf("Advanced mode is %s\n", advanced ? "on" : "off")
```

Result:

```
Advanced mode is on
```

Example 2

To change current advanced mode state:

Command input:

```
advanced = false
```


asin

Return the arc sine of an expression.

Syntax

```
[result =] asin(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8 evaluated in radians.

Discussion

The asin function 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).

Note: Values returned by this 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.

Example

Command input:

```
asin(1)
```

Result:

```
1.5708
```

Related Topics:

[acos](#)
[atan](#)
[atan2](#)
[cos](#)
[sin](#)
[tan](#)

asm

Display memory as disassembled instructions or assemble instructions in-line.

Syntax

Disassembler:

```
[[px]] asm addr-spec
```

Assembler:

```
[[px]] asm addr-spec = "statement" [, "statement"]
```

```
[[px]] asm addr =
```

Where:

<i>[px]</i>	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.
<i>addr-spec</i>	{ <i>addr</i> <i>addr</i> to <i>addr</i> <i>addr</i> length <i>expr</i> }
<i>statement</i>	{ instruction directive }
<i>addr</i>	specifies an address for display or assembly.
length	specifies the number of instructions to be displayed.
<i>expr</i>	specifies a number or an expression that indicates the number of instructions to be displayed.
<i>instruction</i>	is the Intel assembly language instruction.
<i>directive</i>	is an 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 a fixed format. 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 have not yet been defined; however, 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 look something like this:

```
addr codebytes mnemonic [[operand,] operand]
```

Where:

addr specifies the address of the start of the instruction.

codebytes is the byte encoding of the instruction.
 mnemonic is the instruction mnemonic.
 operand is an instruction operand. The number of operands is instruction specific. There may be zero or more.

Example 1

To display a single instruction at offset 0ah in the current code segment:

Command input:

```
asm 0ah
```

Result:

```
00000000AH 0000 ADD [BX+SI] ,AL
```

Example 2

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
```

Example 3

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
```

Example 4

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

The assembler assembles instructions in-line using the current processor modes and settings and places the instruction code in memory. The current focus processor settings are used by default, but 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 Intel assemblers.

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)
tdword 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. See **constants**.

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.

Example

To generate an infinite loop at the current execution point:

Command input:

```
asm $ = "jmp $"
```

Numeric expressions can be formed wherever a number or address is required. The following expression operators are recognized.

/	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

When assembling, the instructions can be entered in one of two 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; ending a line with a comma indicates line continuation. The instructions are parsed and placed in memory. Any errors encountered are reported at the end. This form is well suited to inclusion in macros or procedures. This is also the form to use to fill a region of memory with an instruction sequence.

If a range of memory is given as the addr-spec using either the addr to addr or addr length expr, then that range of memory is filled with the code bytes generated by the statements given. This is very useful for rapidly filling memory with single or repeated instruction sequences.

Example 1

To patch 16 NOP instructions starting at address 5000:

Command input:

```
asm 5000 len 16t = "nop"
```

Example 2

To fill a region of memory starting at physical address 4000 with 20 repetitions of a repeating sequence of instructions:

Command input:

```
asm 4000 len 20t = "add bx,ax", "add cx,ax"
```

Interactive Assembly

The second form of in-line assembly 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 a macro, debug procedure, or include file, the ENDASM command is required.

Note: Interactive assembly is not available within procedures.

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 is filled with the code bytes generated by the statements given. This is useful for rapidly filling memory with single or repeated instruction sequences.

Example

To patch code starting at address 4000:

Command input:

```
asm 4000 =  
@00004000>mov bx,ax  
@00004004>mov bx,ax  
@00004008>endasm
```

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 size indicated by the current processor state. This allows you to input 16-bit code while the processor is in 32-bit mode.*
- use32 temporarily overrides the code size indicated by the current processor state. This allows you to input 32-bit code while the processor is in 16-bit mode.*

* The command language [use](#) control variable may have already overridden the current processor setting. If so, then the assembler use16 and use32 directives temporarily override the mode set via the command language override and not the current processor default.

Address Directive

```
org addr-expr
```

Where:

addr-expr is the numeric and/or symbolic expression that evaluates to an address.

Data Directive

```
data-op data-value
```

Where:

data-value { data | count dup ([data,] data) }

data { 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

Example 1

To enter a string as data at linear address 1000:

Command input:

```
asm 1000L =
P0@00001000L>byte "This is a test string."
P0@00001016L>
```

Example 2

To clear 10 bytes to zero starting at linear address 5432:

```
asm 5432L =
P0@00005432L>db 10 dup ( 0 )
P0@0000543CL>
```

Example 3

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>
```

Example 4

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>
```

Related Topics

[Memory Access](#)
[use](#)
[verify](#)

asmmode

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.

Note: 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
```

Related Topics

[asm](#)
[Expressions](#)
[use](#)

atan

Return the arc tangent of an expression.

Syntax

```
[result =] atan(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8 evaluated in radians.

Discussion

The atan command returns the arc tangent of an expression. The return value is in the range $-\pi/2$ to $\pi/2$.

Note: Values returned by this 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.

Example

Command input:

```
atan(1)
```

Result:

```
0.785398
```

Related Topics:

[acos](#)
[asin](#)
[atan2](#)
[cos](#)
[sin](#)
[tan](#)

atan2

Return the second arc tangent of *expr2* divided by *expr1*.

Syntax

```
[result =] atan2(expr1, expr2)
```

Where:

result specifies a debug object of type real8 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.

expr1,
expr2 specifies a number or an expression of type real8 evaluated in radians.

Discussion

The atan2 command returns the second arc tangent of *expr2* divided by *expr1*. The return value is in the range -pi to pi.

Note: Values returned by this 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.

Example

Command input:

```
atan2(1,2)
```

Result:

```
0.463648
```

Related Topics:

[acos](#)
[asin](#)
[atan](#)
[cos](#)
[sin](#)
[tan](#)

autoconfigure

Automatically scan and configure target devices.

Syntax

```
[result =] autoconfigure([scan])
```

Where:

result specifies a boolean variable 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.

scan indicates whether the emulator should scan the JTAG chain.

Discussion

The autoconfigure command is used to automatically configure the target. It verifies the integrity of the JTAG chain, scans for JTAG devices and configures them. A return value of true indicates the command was successful.

Example

Command Input:

```
autoconfigure(true) // configure target (force a JTAG chain scan)
```

Related topics

[deviceconfigure](#)

[devicescan](#)

[jtagconfigure](#)

[jtagscan](#)

base

Display or change 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 {2, 8, 10 or 16}. If any other value is entered, an error message is displayed. The default is 16 for 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 default number base in the command interpreter. 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.

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 will not change the base to decimal if the current base is hexadecimal. Use base = 10t instead.

Example 1

To display the current number base:

Command input:

```
base
```

Result:

```
336
```

```
0010H
```

Example 2

To set the current number base to decimal:

Command input:

```
base = 10t
base
```

Result:

```
10T
```

Example 3

To set the current number base to hexadecimal:

Command input:

```
base = hex
base
```

Result:

```
0010H
```

Example 4

To save and then restore the current display base:

Command input:

```
define ord2 svBase = base
base = oct
base
```

Result:

```
0000100
```

Command input:

```
base = svBase
base
```

Result:

```
0010H
```

Related Topics:

[Expressions](#)

bell (beep)

Cause an alert to sound.

Syntax

```
{bell | beep}
```

Discussion

The bell command can be used in scripts to signal that an event has occurred. Beep is a synonym for bell.

Example

```
while (ord4 0x10000 != 1000)
{
    go
    wait
}
bell
```

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:

<i>[px]</i>	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
<i>component</i>	is a valid register name or debug variable.
<i>bit-offset</i>	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.
<i>bit-size</i>	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.
<i>expr</i>	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

Exit from a control block.

Syntax

```
break
```

Discussion

Use the break command to cause termination of the nearest enclosing [while](#), [do while](#), [for](#), or [switch](#) command.

Example

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
```

Related Topics:

[do while](#)
[for](#)
[switch](#)
[while](#)

breakall

Display or change whether all target processors start and stop together in a multiprocessor system.

Syntax

```
breakall [= bool-cond]
```

Where:

bool-cond specifies a number of 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.

Example 1

Command input:

```
breakall           // display the current setting
```

Result:

```
true
```

Example 2

Command input:

```
breakall=false
go           // only viewpoint processor is run
```

Example 3

Command input:

```
breakall=false
[p0]go       // P0 processor override used to run only P0 processor
```

SourcePoint 7.12

Related Topics:

[go](#)

cachememory

Display or change how 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 128 bytes at a time. This minimizes the time it takes for refreshing Code and Memory windows. The data read is 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 work properly.

There are times, however, primarily when executing command files that perform numerous memory accesses, where 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.

Example 1

Command input:

```
cachememory           // display the current setting
```

Result:

```
false
```

Example 2

Command input:

```
cachememory = true    // enable block memory reads and caching
```

SourcePoint 7.12

```
ord4 100          // cachememory is true, only one target
                  // memory read at 100-17f will occur
ord4 10
ord4 108
```

Example 3

Command input:

```
cachememory = false // disable block memory reads and caching
ord4 100           // cachememory is false, three separate
                  // target memory reads will occur
ord4 10
ord4 108
```


cause

Display the reason for the last target stop.

Syntax

```
[[px]] cause
```

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.

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 or ETM breakpoints.

The information is also displayed automatically in the SourcePoint Status Bar immediately after the stop (although it will be overwritten by later status information). Being able to determine the cause of a target stop (which breakpoint hit), enables the use of breakpoint macros (e.g., a macro file can be executed whenever a specific breakpoint occurs).

Possible return values include:

```
"Unknown reason"
"User stop"
"Step completed"
"Target reset"
"Processor breakpoint @ 00010004"
"Software breakpoint @ 00010008"
```

Example 1

To display the reason for the last target stop:

Command input:

```
cause
```

Result:

```
Software breakpoint @ 00010008
```

Example 2

To assign the cause string to a debug variable:

Command input:

```
define nstring strReason=cause  
strReason
```

Result:

```
Software breakpoint @ 00001020
```

Character Functions

Built-in functions for character classification and transformation.

Syntax

```
[result =] function(char-expr)
```

Where:

<i>result</i>	specifies the debug object to which the function return value is assigned. If result 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.
<i>function</i>	specifies the name of the character function (see the following table).
<i>char-expr</i>	specifies a quoted character or an expression specifying a character.

Discussion

There are two classes of character functions: 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 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.

Character Functions

Function	Discussion
isalpha	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	Returns true when char-expr is an uppercase alphabetic character. The hexadecimal values for these characters are 41 through 5a (A . . . Z).
islower	Returns true when char-expr is a lowercase alphabetic character. The hexadecimal values for these characters are 61 through 7a (a . . . z).
isdigit	Returns true when char-expr is a numeric digit. The hexadecimal values for these characters are 30 through 39 (0 . . . 9).
isxdigit	Returns 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	Returns 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	Returns 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	Returns 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	Returns true when char-expr is a printable character. The hexadecimal values for these characters are 20 through 7e.
iscntrl	Returns true when char-expr is a delete character (hexadecimal 7f) or any control character (hexadecimal 0 through 1f).
isascii	Returns true when char-expr is a coded value (hexadecimal 0 through 7f).
toupper	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	Returns the lowercase value of char-expr. If char-expr does not contain an uppercase letter, the result is the original char-expr, unchanged. The char-expr itself is not changed.
toint	Returns the "weight" of a hexadecimal digit: 0 - 9 for the characters "0" through "9", respectively, and 10 - 15 for the letters "a" through "f" (or "A" through "F"), respectively.
toascii	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

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:

SourcePoint 7.12

```
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
```

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
```

Result:

```
'a'
```

Related Topics:

[Expressions](#)

clock

Return the elapsed time (in ms) since SourcePoint started.

Syntax

```
[result =] clock()
```

Where:

result specifies an ord4 variable 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.

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
```

Related Topics:

[ctime](#)
[time](#)

continue

Transfer control from within a control block to the end of the block.

Syntax

```
continue
```

Discussion

Use the continue command to cause a jump to the end of the immediately enclosing iteration statement ([while](#), [do while](#) or [for](#)).

Example

This example shows a continue command within a for loop. The variable x contains the sum of numbers between 0 and 12 whose modulus equals 2.

Command input:

```
define int2 a
define int2 x = 0
for (a = 0; a <= 12; a += 1)
{
    if ((a % 3) != 2)
        continue
    x = x + 1
}
x
```

Result:

```
0014H
```

Related Topics:

[break](#)
[do while](#)
[for](#)
[while](#)

COS

Return the cosine of a radian expression.

Syntax

```
[result =] cos(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8 evaluated in radians.

Discussion

The cos command returns the cosine of *expr*.

Note: Values returned by this 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.

Example

Command input:

```
cos(0)
```

Result:

```
1.000
```

Related Topics:

[acos](#)
[asin](#)
[atan](#)
[atan2](#)
[sin](#)
[tan](#)

cpubreak, cpuremove, cpudisable, cpuenable

Set, clear, display, enable and disable processor breakpoints.

Syntax

```
cpubreak
cpubreak = [sts,] type [, name] [, processor-spec]

cpuremove [all]
cpuremove = {type | proc} [ ,... ]

cpuenable = {type | name | proc} [ ,... ]

cpudisable [all]
cpudisable = {type | proc} [ ,... ]
```

Where:

<i>sts</i>	{ e[nabled] d[isabled] }
<i>proc</i>	p[rocessor] = { P0 P1 P2 ... }
<i>type</i>	{smm entry smm exit power cycle machine check}
<i>name</i>	n[ame] = <i>breakpoint name</i>

Discussion

The cpubreak command sets and displays processor breaks. Cpubreak with no arguments displays a list of the current processor breakpoints.

The cpuremove command removes any or all of the processor breaks. Arguments to this command qualify which processor breakpoints are to be removed. For instance, cpuremove = p=P0 removes all processor breakpoints associated with processor 0. Cpuremove with no arguments removes all processor breakpoints.

The cpuenable command selectively enables processor breakpoints. Arguments to this command qualify which processor breakpoints are to be affected. For instance, cpuenable = p=P1 enables only processor breakpoints associated with processor 1.

The cpudisable command selectively disables processor breakpoints. Arguments to this command qualify which processor breakpoints are to be affected. For instance, cpudisable = smm entry, disables only processor breakpoints with the type set to smm entry. If no arguments are specified, all processor breakpoints are disabled.

Processor breakpoints can also be set, displayed, etc. from the [Breakpoints Window](#).

Examples

To display all processor breaks:

```
cpubreak
```

To break when a processor enters smm:

```
cpubreak = smm entry
```

To break when processor 1 exits smm:

```
cpubreak = smm exit, p=P1
```

To remove all processor breaks:

```
cpuremove
```

To remove the smm entry catch break:

```
cpuremove = smm entry
```

To remove all breaks associated with processor 1:

```
cpuremove = p=P1
```

To disable all processor breaks:

```
cpudisable
```

Related Topics:

[Breakpoints View](#)
[dbgbreak commands](#)
[emubreak commands](#)
[softbreak commands](#)

cpuid_eax

Execute the CPUID assembly instruction and return the value in EAX.

Syntax:

```
[result =] [[px]] cpuid_eax [(eax[,ecx])]
```

Where:

<i>[px]</i>	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.
<i>eax</i>	is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
<i>ecx</i>	is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
<i>result</i>	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:

```
00020652H
```

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:

```
00000001H
```

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

Related Topics

[cpuid_ebx](#)

[cpuid_ecx](#)

[cpuid_edx](#)

cpuid_ebx

Execute the CPUID assembly instruction and return the value in EBX.

Syntax:

```
[result =] [[px]] cpuid_ebx [(eax[,ecx])]
```

Where:

<i>[px]</i>	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.
<i>eax</i>	is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
<i>ecx</i>	is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
<i>result</i>	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:

```
04100800H
```

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:

```
00000002H
```

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
```

Related Topics

[cpuid_eax](#)

[cpuid_ecx](#)

[cpuid_edx](#)

cpuid_ecx

Execute the CPUID assembly instruction and return the value in ECX.

Syntax:

```
[result =] [[px]] cpuid_ecx [(eax[,ecx])]
```

Where:

<i>[px]</i>	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.
<i>eax</i>	is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
<i>ecx</i>	is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
<i>result</i>	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:

```
0298E3FFH
```

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:

```
00000100H
```

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:

00000005H

Related Topics

[cpuid_eax](#)

[cpuid_ebx](#)

[cpuid_edx](#)

cpuid_edx

Execute the CPUID assembly instruction and return the value in EDX.

Syntax:

```
[result =] [[px]] cpuid_edx [(eax[,ecx])]
```

Where:

<i>[px]</i>	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.
<i>eax</i>	is the value to be stored in EAX before the CPUID instruction is executed. If no value is specified, 1 is used by default.
<i>ecx</i>	is the value to be stored in ECX before the CPUID instruction is executed. If no value is specified, 0 is used by default.
<i>result</i>	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:

```
BFEBFBFFH
```

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:

```
00000004H
```

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:

00000004H

Related Topics

[cpuid_eax](#)

[cpuid_ebx](#)

[cpuid_ecx](#)

createprocess

```
createprocess(command, waitforcompletion)
```

Where:

command is a command line string to create the process.

waitforcompletion is a boolean indicating whether SourcePoint should wait for the process to complete.

Discussion

The createprocess function call can be used to start another program. The command string includes the program name, along with any arguments to pass to the program.

Examples

Command input:

```
createprocess("notepad c:\\temp\\foo.txt", false)    // start notepad, do not  
wait for it to be closed
```

Command input:

```
createProcess("c:\\temp\\unlock.bat", true)         // run a batch file, wait  
for it to complete
```

Related Topics:

[shell](#)

cscfg and local_cscfg

Display or change the contents of configuration registers in a chipset device through an ITP debugger interface.

Device	Access	Status	Addressing
TBG_CORE	JTAG	Error Status 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.
TNB	JTAG	Error bit has known problem in certain register address ranges. DE recommends ignoring it. ITP does not test it.	4k configuration space, no memory mapped addressing. No transaction-type field. If system transaction flow 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.
XMB	JTAG	Error bit 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, no memory mapped addressing. No transaction-type field. Does not support the JCONFG instruction register. ITP maps both cscfg and local_cscfg to the JCONFL instruction register.
PXH	JTAG	Error bit not implemented, always 0.	4k configuration space, no memory mapped addressing. Has Transaction-type field but it is always 0
LHMCH (Cayuse)	JTAG	Error bit not implemented, always 0.	4k configuration space, plus memory mapped addressing. Has Transaction-type field, it can be 0 or 1.

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 * 0n8) + SNC_SPAD_FCN)
#define  SIOH_SPAD_DEV_FCN ((SIOH_PCI_DEV_NUM * 0n8) + 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 0n2
#define  DWORD_WRITE 0n3
#define  DWORD_READ 0
```

Result:

```
// get the read-only DEVICE and VENDOR ID 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:

```
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 0x3ffff on Lindenhurst device, using memory-mapped addressing:

Command input:

```
cscfg(LHMCHPF0, 1, 0x3ffff, 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
```

Related Topics

[devicelist](#)

csr

Read/write control bus registers for Uncore devices.

Syntax

```
[result =] csr([DID,]reg) [= value]
```

Where:

<i>reg</i>	is an expression that evaluates to a valid control bus register number.
<i>value</i>	is an expression that evaluates to a 32 bit value to load in reg.
<i>result</i>	specifies a debug variable 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.
<i>DID</i>	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)
```

Related Topics

[devicelist](#)

ctime

Convert the output of the [time](#) command into a null-terminated ASCII string.

Syntax

```
[result =] ctime(expr)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>expr</i>	specifies a number or an expression.

Discussion

The ctime function 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."

Example

Command input:

```
define ord4 now = time()
ctime(now)                // answer will be current day and time
```

Result:

```
Wed Jun 07 16:26:07 2008
```

Related Topics:

[_strdate](#)
[_strtime](#)
[time](#)

cwd

Set or display the current working directory.

Syntax

```
cwd [pathname]
```

Discussion

The current working directory is the default path that SourcePoint uses to locate files. This applies to all commands that accept a filename as an input. The `cwd` command without an argument displays the current working directory.

Note: The `cwd` command is deprecated. Use the `defaultpath` control variable instead.

Example 1

To display the current working directory:

Command input:

```
cwd
```

Result:

```
c:\Program Files\Arium\SourcePoint
```

Example 2

To set the current working directory:

Command input:

```
cwd c:\temp  
cwd
```

Result:

```
c:\temp
```

Example 3

To change to a new `cwd` relative to the existing `cwd`:

Command input:

```
cwd samples  
cwd
```

Result:

c:\Program Files\Arium\SourcePoint\Samples

Related Topics:

[defaultpath](#)

[homepath](#)

[macropath](#)

[projectpath](#)

dbgbreak, dbgremove, dbgdisable, dbgenable

Set, clear, display, enable, and disable debug register breakpoints.

Syntax

```
dbgbreak
dbgbreak = [sts,]{type},location[,name][,proc][,translate]

dbgremove [all]
dbgremove = {type | name | location | size | proc} [,...]

dbgenable = {type | name | location | size | proc} [,...]

dbgdisable [all]
dbgdisable = {type | name | location | size | proc} [,...]
```

Where:

```
sts      { e[nabled] | d[isabled] }
type     { execute | data access | data write | I/O access } [in smm]
name     n[ame] = breakpoint name
proc     p[rocessor] = { P0 | P1 | P2 | ... | All }
location l[ocation] = address
size     s[ize] = { b[yte] | h[alf-word] | w[ord] }
translate x = { once | every go }
```

Discussion

The dbgbreak command sets and displays debug register (hardware) breakpoints. Dbgbreak with no arguments displays a list of the current debug register breaks.

The dbgremove command removes any or all of the debug register breaks. Arguments to this command qualify which debug register breaks are to be removed. For instance, dbgremove=data write, s=byte, removes all debug register breaks with the type set to data write and size set to byte. Dbgremove with no arguments removes all debug register breaks.

The dbgenable command selectively enables debug register breaks. Arguments to this command qualify which debug register breaks are to be affected. For instance, dbgenable=execute enables only debug register breaks with the type set to execute.

The dbgdisable command selectively disables debug register breaks. Arguments to this command qualify which debug register breaks are to be affected. For instance, dbgdisable=execute disables only debug register breaks with the type set to execute. If no arguments are specified, all debug register breaks are disabled.

Debug register breaks can also be set, displayed, etc. from the Breakpoints window.

Examples

To set a debug register break on a word access at location 1234:

```
dbgbreak = data access, location=1234, size=word
```

To set a debug register break on a word read at address 1000p:

```
dbgbreak = data read, location=1000p, size=word
```

To remove all debug register breaks:

```
dbgremove
```

To remove all debug register breaks with type set to data write and size set to bytes:

```
dbgremove = data write, size=byte
```

To disable all debug register breaks:

```
dbgdisable
```

To disable all debug register breaks with type set to execute:

```
dbgdisable = execute
```

To enable all debug register breaks with location set to 1234:

```
dbgenable = 1=1234
```

Related Topics:

[Breakpoints View](#)
[cpubreak commands](#)
[softbreak commands](#)

defaultpath

Set or display the current working directory.

Syntax

```
[result =] defaultpath [= newpath]
```

Where:

<i>result</i>	specifies an nstring variable to which the function return value is assigned. If <i>result</i> is not specified, the return value is displayed on the next line of the screen.
<i>newpath</i>	is an nstring variable or string constant specifying a new working directory.

Discussion

The defaultpath control variable displays the default path that SourcePoint uses to locate files. This applies to all commands that accept a filename as an input.

Example 1

To display the current working directory:

Command input:

```
defaultpath
```

Result:

```
c:\Program Files\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
```

Result:

```
c:\Program Files\Arium\SourcePoint
```

Related Topics:

[cwd](#)

[homepath](#)

[macropath](#)

[projectpath](#)

#define

Create a debug alias.

Syntax

```
#define alias-name commands
```

Where:

alias-name is an identifier that serves as an alias for the given command string.

commands is a 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. SourcePoint reserved words cannot be used as aliases.

The debug alias can be defined inside or outside a control construct, a compound statement, or a debug procedure and is always global.

The [show](#) command displays a list of currently defined aliases. The [remove](#) and [#undef](#) commands can be used to remove alias definitions.

Example

To define an alias for an often-used load command:

Command input:

```
#define ld load c:\src\targdev  
show alias
```

Result:

```
ld      alias      "load c:\src\targdev"
```

Related Topics:

[#undef](#)
[include](#)
[remove](#)
[show](#)

define

Create a debug object.

Syntax

```
define debug-proc
define [global] data-type name [=expr]
define [global] data-type name [array-size]
```

Where:

<i>debug-proc</i>	specifies a debug procedure definition.
<i>name</i>	specifies a unique, user-defined name for the object being defined.
global	indicates that a debug variable is globally recognized. Data types are global unless defined inside a debug procedure.
<i>data-type</i>	specifies an emulator data type.
<i>expr</i>	is an expression that assigns an initial value to the object.
<i>array-size</i>	is an expression indicating the size of a debug variable array.

Discussion

Use the define command to define a debug variable or a debug procedure. 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.

An initial value may be assigned to a debug variable. If no initial value is specified, the variable is assigned a default value, depending on its data type:

Type	Default Value
ordn, intn	0
nstring	""
bool	false
ptr	invalid

Arrays of debug variables can also be created. For more information see [Debug Variables](#). For more information on defining a debug procedure, see [Debug Procedures](#).

Example 1

To define a debug variable:

Command input:

```
define int2 max = 400
max
```

Result:

0400H

Example 2

The following example shows how to define a procedure named "power." This proc returns the result of a value and its exponent.

Command input:

```
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(2,4)           // execute the proc
```

Result:

16T

Related Topics:

[#define](#)

[#undef](#)

[proc](#)

[remove](#)

[show](#)

[Data Types](#)

[Debug Procedures](#)

[Debug Variables](#)

[Expressions](#)

definemacro

Assign macros (include files) to user-definable buttons on the Macro toolbar.

Syntax

```
definemacro(id, filename[, echo, text])
```

Where:

<i>id</i>	is an integer (0-19) indicating the toolbar button to assign. If a button already has an assigned macro, then the previous definition is overwritten.
<i>filename</i>	specifies a filename. See Filenames for details.
<i>echo</i>	is 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.
<i>text</i>	is 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

The defineMacro function is used to assign macros (include files) to user-definable buttons on the Macro toolbar. (The Macro toolbar is enabled by selecting View|Toolbars|Macro). Macro 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 20 user-definable buttons. By default, four are displayed. You can right-click on the macro 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.

Example 1

To assign the macro "c:\test\qa.mac" to the first button in the toolbar:

Command input:

```
defineMacro(0, "c:\\test\\qa.mac", true, "Run QA tests")
```

Note: The contents of this macro are echoed to the Command window when the button is clicked. The button is labeled Run QA tests.

Example 2

To assign the macro "loop.txt" (found in the current working directory) to the ninth button in the macro toolbar (the index is 0 based):

Command input:

```
defineMacro(8, "loop.txt")
```

Note: The contents of this macro are not echoed to the Command window. The button does not have a text label (icon only).

Example 3

To clear the macro definition of the first toolbar button:

Command input:

```
defineMacro(0, "")
```

Related Topics:

[include](#)

deviceconfigure

Synchronize the device configurations between SourcePoint and the emulator.

Syntax

```
[result =] deviceconfigure([force])
```

Where:

- | | |
|---------------|--|
| <i>result</i> | specifies a boolean variable 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. |
| <i>force</i> | indicates whether the Device configuration table should be forced into the emulator when the existing emulator configuration differs. |

Discussion

The deviceconfigure function synchronizes the device configurations between SourcePoint and the emulator. If the "force" flag is true, SourcePoint's device configuration replaces any existing configuration in the emulator. If the "force" flag is false the configurations are verified for consistency. In the event of a mismatch, the configurations are presented to the user to select which configuration is to be used. This operation is valid when the emulatorstate control variable is set to 1 or 2. If it succeeds, the emulatorstate control variable transitions to state 2.

Related Topics

[autoconfigure](#)
[devicelist](#)
[devicescan](#)
[emulatorstate](#)
[Target Configuration](#)
[verifydeviceconfiguration](#)

devicelist

Display names and other device information of currently attached or configured devices.

Syntax

```
devicelist
[result =] devicelist[expr]
[result =] 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).

result specifies a debug variable 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.

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.

Fieldname	Description	Data Type
debugport	debug port	Ordinal
cp	chain position	Ordinal
did	device id	Ordinal
tapport	Tap port id	Ordinal
scanchain	scan chain	Ordinal
threadid	thread/core ID of this device	Ordinal
device	device name	String
alias	alias for this device	String
idcode	idcode for JTAG device only	Ordinal
stepping	device stepping	String
devicetype	device type	String
isenabled	thread/device's enabled status	Boolean
bustype	bus type used by the device	String

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:

DID	DP	TP	SC	Alias	Type	Step	Idcode	BusType	T/C	Enabled
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
```

Related Topics

[first_jtag_device](#)
[idcode](#)
[num_jtag_devices](#)

devicescan

Direct the emulator to perform device discovery on the target.

Syntax

```
[result =] devicescan()
```

Where:

result specifies a boolean variable 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.

Discussion

The devicescan function directs the emulator to perform device discovery on the target. While the implementation varies on different target types, the results of this operation are used to populate the SourcePoint Device configuration table, which is the source for the deviceconfigure function.

This command is acceptable only when the emulatorstate control variable is set to 1, following a successful jtagconfigure or [uncoreconfigure](#) operation.

The devicelist command can be used to display a list of discovered devices.

Example

Command Input:

```
devicescan()
```

Result:

```
TRUE          // command succeeded
```

Related Topics

[autoconfigure](#)
[deviceconfigure](#)
[devicelist](#)
[emulatorstate](#)
[Target Configuration](#)

disconnect

Disconnect the emulator from the target.

Syntax

```
disconnect
```

Discussion

The disconnect command disconnects the emulator from the target. The emulatorstate control variable transitions to state 0 (disconnected). This command has the same effect as pressing the Disconnect button in the Processor toolbar.

A common use of the disconnect command is when changing to a different target. This forces the emulator to discard its current target configuration and either scan the target for a new one or have SourcePoint download a new configuration.

Example

Command Input:

```
disconnect      // disconnect from target
reconnect       // reconnect to target
```

Related Topics

[emulatorstate](#)

[reconnect](#)

[Target Configuration](#)

displayflag

Determine 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).

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. The default value for displayflag is false. If you enter the displayflag control variable by itself, 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
```

do while

Group and conditionally execute emulator commands.

Syntax

```
do {commands} while(bool-cond)
```

Where:

<i>commands</i>	specifies one or more emulator commands. The braces are required when you enter multiple commands.
<i>bool-cond</i>	specifies that the loop ends when bool-cond is false. The bool-cond option specifies a number or an expression that must evaluate to true (non-zero) or false (zero).

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'
```

Related Topics:

[break](#)
[continue](#)
[for](#)
[while](#)

dos

Execute 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. Key in "exit" (without the quotation marks) 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. Shell is a synonym for DOS.

Example 1

To copy a file to a different directory:

Command input:

```
dos copy c:\tmp\test.list c:\save
```

Example 2

To open a DOS window:

Command input:

```
dos
```

Related Topics:

[shell](#)

dport

Display or change the contents of a 32-bit I/O port.

Syntax

```
[result =] [[px]] dport(io-addr) [= expr]
```

Where:

- result* specifies a debug variable 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.
- [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.
- 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.

Example 1

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
```

Example 2

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
```


drscan

Scan the data registers of devices on the JTAG chain.

Syntax

```
drscan(device, bitCount, readArray [, writeArray])
drscan(device, writeArray)
```

Where:

<i>device</i>	is an int4 that specifies the position of the device to access. Device positions are displayed by the devicelist command.
<i>bitCount</i>	is an expression that evaluates to the number of bits to scan to or from the data register of the designated device as selected by the current instruction register value.
<i>readArray</i>	is a debug array you have defined that is large enough to hold the scanned data register. This array must be equal to or larger than the number of bits specified to scan from the data register. Array elements that are not used are unchanged.
<i>writeArray</i>	is a debug array you have defined that holds the value you want to scan into the data register. If this array is smaller than the number of bits to be scanned into the data register, an error occurs.

Description

Use drscan to read or write the data registers of devices on the JTAG chain. The device specification determines which device in the chain is scanned. All other devices are in bypass.

The instruction current in the instruction register of the specified device determines the data register that is scanned. Use the irscan command to write an instruction to the instruction register.

The bitCount value determines the number of bits that will be scanned. A debug array that will read these scanned bits must contain at least as many bits as are to be scanned. A debug array that will write the scanned bits also must have at least as many bits as the number of bits to be scanned in. The drscan command will only use array types for parameters, you cannot use a signed or unsigned value (e.g., ord8) even though it may contain enough bits.

The drscan command can be used to either read a data register, write a data register, or both read and write a data register. When writing only from a device data register you must use the extra comma (,) placeholder to specify the write-array parameter position. The following are acceptable forms for the drscan command:

```
drscan(0, 1, ReadArray)
drscan(0, 1, , WriteArray)
drscan(0, 1, ReadArray, WriteArray)
```

Example 1

To write instruction and data to device 0 and ignore return data:

Command input:

```
irscan(0, 1)
define ord1 a_olWriteToDeviceArray[0n20] // big enough to hold 159 bits
drscan(0, 0n159, , a_olWriteToDeviceArray)
```

Example 2

To write instruction and data to device 5 and save return data:

Command Input:

```
irscan(5, 0x10)
define ord1 ReadFromDeviceArray[0n17] // enough to hold 0x82 bits
define ord1 WriteToDeviceArray[0n17] // filled with 0's to write
drscan(5, 0x82, ReadFromDeviceArray, WriteToDeviceArray)
```

Example 3

To write instruction and data to device 0 and save return (note that write data is not specified and will default to all 0's):

Command Input:

```
irscan(5, 2)
define ord4 ReadFromDeviceArray[1] // enough to hold 32 bits
drscan(5, 0x20, ReadFromDeviceArray)
ReadFromDeviceArray[0]
```

Result:

182C1013

Related Topics

[irscan](#)
[msgscan](#)
[tapdateset](#)
[tapdatashift](#)

edit

Open a file for edit.

Syntax

```
edit [proc] [filename]
```

Where:

proc indicates the file should be processed after edit.

filename specifies a filename. See [Filenames](#) for details.

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").

Example

Command input:

```
editor="c:\vslick\win\vs.exe" // change editor to slick  
cwd cmd\test                 // change working directory  
edit proc fileio.cmd         // edit fileio.cmd and re-parse
```

Related Topics:

[editor](#)

editor

Specify 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.

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

To specify an editor:

Command input:

```
editor="c:\vslick\win\vs.exe" // change editor to slick  
cwd cmd\test                // change working directory  
edit fileio.cmd              // edit fileio.cmd
```

Related Topics:

[edit](#)

emubreak, emuremove, emudisable, emuable

Set, clear, display, enable and disable emulator breakpoints.

Syntax

```
emubreak
emubreak = [sts,] {type}

emuremove [all]
emuremove = {type} [ ,... ]

emuable = {type} [ ,... ]

emudisable [all]
emudisable = {type} [ ,... ]
```

Where:

```
sts                { e[nabled] | d[isabled] }
type               { reset | init | bkpt in }
```

Discussion

The `emubreak` command sets and displays emulator breakpoints (emulator breaks). `Emubreak` with no arguments displays a list of the current emulator breaks.

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
```

Related Topics

[Breakpoints Windows Introduction](#)

emulatorstate

Return an integer representing the emulator connection state.

Syntax

```
[result =] emulatorstate
```

Discussion

The emulatorstate control variable returns an integer representing the emulator connection state.

- 0 Not configured. The disconnect command changes emulatorstate to this value.
- 1 JTAG chain(s) configured and ready for device configuration. The jtagconfigure command changes emulatorstate to this value.
- 2 Devices configured and ready for debug session. The deviceconfigure command changes emulatorstate from 1 to 2. The reconnect command changes emulator state from 0 or 1 to 2.

Example

Command Input:

```
printf("Emulator state is %d", emulatorstate)
```

Result:

```
00000002H // fully connected
```

Related Topics

[deviceconfigure](#)

[disconnect](#)

[jtagconfigure](#)

[reconnect](#)

[Target Configuration](#)

encrypt

Encrypt 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 is an nstring variable or string constant specifying the file to encrypt.

output_file is 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
nodisplay	The error message will not be displayed in the Command window.
warning	The error message will be displayed as a warning but will not affect the execution of SourcePoint scripts.
severe	The error message will be displayed as a severe error but will not affect the execution of SourcePoint scripts.
error	The error message will be displayed as a normal error and will cause the current script control block to be stopped.
fatal	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)
```


evalprogramsymbol

Return the value of the symbol.

Syntax

```
[result =] evalprogramsymbol(symbol)
```

Where:

result specifies an nstring variable 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.

symbol is a constant string or nstring specifying the symbol to look up.

Discussion

The evalprogramsymbol function searches each program loaded in the current context for the specified symbol. If the symbol is found, its value is returned. Otherwise, an error message is displayed.

The value returned for a data symbol (variable) is its current state in the target. The value of a code symbol is its address.

Examples

The following examples demonstrate the evalprogramsymbol function. Here it is assumed that a program is loaded that contains the data symbol mydata with a value of '7' and a procedure symbol mycode at address C0008000

Example 1

Command input:

```
EvalProgramSymbol("mydata")
```

Result:

```
7
```

Example 2

Command input:

```
define nstring s = "mydata"
EvalProgramSymbol(s)
```

Result:

```
7
```

Example 3

Command input:

```
EvalProgramSymbol("mycode")
```

Result:

```
0xC0008000
```

Example 4

Command input:

```
EvalProgramSymbol("test")
```

Result:

```
Error "test" is not a program symbol
```

Related Topics:

[getprogramsymboladdress](#)
[isprogramsymbol](#)

execution point (\$)

Display or change the current execution point (CS:EIP).

Syntax

```
$ [=addr]
```

Where:

addr is 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
```

Example 2

To change the current execution point to the main procedure in the p_main module:

Command input:

```
$=:p_main.main
```

Example 3

To hand-patch a test case at the address 0x0008:000f:000005ff:

Command input:

SourcePoint 7.12

```
$ = 0x0008:000f:000005ff  
asm $ = "mov ax,word ptr [0]", "mov ebx,eax"
```

exit

Exit SourcePoint.

Syntax

```
exit
```

Discussion

Use the exit command to close all open files and terminate the debug session.

exp

Return the exponential function of an expression.

Syntax

```
[result =] exp(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8.

Discussion

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 logarithm. The exp function returns infinite when the correct return value would overflow.

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.

Example

Command input:

```
exp(1)
```

Result:

```
2.71828
```

Related Topics:

[pow](#)

fc

Compare two text files.

Syntax

```
[result =] fc("file1", "file2"[, start column[, end column]])
```

Where:

<i>result</i>	specifies a debug variable 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.
<i>file1, file2</i>	specifies a filename. See Filenames for details.
<i>start column</i>	is the first column of each line to compare, start column = 0 equals the first column of the line.
<i>end column</i>	is the last column of the line to compare.

Discussion

The fc function is used to compare two text files. The function 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.

Note: "Start column" and "end column" are optional input parameters. If not specified, the entire line is compared.

Example**Command input:**

```
define bool result = fc("good.txt", "new.txt")
if (result == true)
    puts("files match\n")
```

fclose

Close a file.

Syntax

```
fclose(file_handle)
```

Where:

file_handle is a file handle returned from a previous [fopen](#) command

Discussion

The fclose function closes a file previously opened by an fopen command.

If an fopen function 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"
```

Related Topics:

[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fputs](#)
[fread](#)
[fseek](#)
[ftell](#)

[fwrite](#)

feof

Test for end of file (EOF).

Syntax

```
[result =] feof(file_handle)
```

Where:

result specifies a debug variable of type int4 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.

file_handle is returned from a previous [fopen](#) command.

Discussion

The feof function 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 returns a value of 00000010H; otherwise, a null value is returned.

Example

To read a binary file and write its contents 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)
    ord1 pMem length nItemsRead = buf
    pMem += nItemsRead
}
fclose(file1)
```

Related Topics:

[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fputs](#)

[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)

fgetc

Read a character from a file.

Syntax

```
[result =] fgetc(file_handle)
```

Where:

<i>result</i>	specifies a debug variable of type int4 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.
<i>file_handle</i>	is the file handle returned from a previous fopen command

Discussion

The fgetc function reads a character from a file previously opened by [fopen](#). 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
```

Related Topics:

[fclose](#)
[feof](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fputs](#)
[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)

fgets

Read a string from a file.

Syntax

```
[result =] fgets(string, file_handle)
```

Where:

result specifies an nstring variable 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.

string is an nstring variable to receive the string read.

file_handle is a file handle returned from a previous [fopen](#) command.

Discussion

The fgets function reads a string from a file previously opened by fopen. The string is stored in the first argument specified and is also the return value of the function. Multiple fgets commands will get consecutive, new line-delimited strings. Strings longer than 1024 characters are truncated. Fgets returns an empty string on end of file. The [feof](#) function can also 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"
```

Related Topics:

[fclose](#)

[feof](#)

[fgetc](#)

[fopen](#)

[fprintf](#)

[fputc](#)

[fputs](#)

[fread](#)

[fseek](#)
[ftell](#)
[fwrite](#)

first_jtag_device

Return the device ID of the first JTAG device.

Syntax

```
[result = ] first_jtag_device
```

Where:

result specifies a debug variable 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.

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 nID
  if (num_jtag_devices > 0)
  {
    for (nID=first_jtag_device; nID <= last_jtag_device; nID++)
    {
      printf("%4x %8s (%-11s) port %d, scanchain %d, idcode %x\n",
        devicelist[nID].did,
        devicelist[nID].alias,
        devicelist[nID].devicetype,
        devicelist[nID].debugport,
        devicelist[nID].scanchain,
        devicelist[nID].idcode)
    }
  }
}
```

Related Topics

[last_jtag_device](#)
[num_jtag_devices](#)

flist

Log command line input and responses to a file.

Syntax

```
flist([filename [,append]])
```

Where:

<i>filename</i>	specifies a filename. See Filenames for details.
<i>append</i>	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)
```

Related Topics:

[list, nolist](#)

[log, nolog](#)

flush

Invalidate the processor's internal caches.

Syntax

```
[[px]] flush [nowriteback]
```

Where:

<code>[px]</code>	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.
<code>nowriteback</code>	clears writeback, which is the default condition for the flush command.

Discussion

Use the flush command to invalidate the processor's internal 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
```

Related Topics

[invd](#)
[wbinvd](#)

fopen

Open a file for input or output.

Syntax

```
file_handle = fopen(filename, type)
```

Where:

<i>file_handle</i>	specifies a debug variable of type ord4 to receive the file handle.
<i>filename</i>	specifies a filename. See Filenames for details.
<i>type</i>	"r" opens an existing file for input.
	"w" creates a new file, or overwrites an existing one for output.
	"a" creates a new file, or appends to an existing one for output.

Discussion

The fopen function 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 the file could not be opened, then 0 is returned. 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.

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
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fprintf](#)
[fputc](#)
[fputs](#)
[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)

for

Group and execute commands in a loop.

Syntax

```
for(command1; bool-cond; command2) {commands}
```

Where:

<i>command1</i>	is usually an assignment statement or a function call, but can be any valid emulator command. If you omit the <i>command1</i> option, the semicolon (;) must remain as a place holder.
<i>bool-cond</i>	specifies the test condition. The <i>bool-cond</i> option must evaluate to true (non-zero) or false (zero). If you omit the <i>bool-cond</i> option, the test condition defaults to true, and the semicolon (;) must remain as a placeholder.
<i>command2</i>	is usually a re-assignment, an increment, or a function call, but can be any emulator command. If you omit the <i>command2</i> option, the semicolon (;) must remain as a placeholder.
<i>commands</i>	is one or more emulator commands that are executed when the test condition <i>bool-cond</i> 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 execute a block of commands 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. To break out of a loop press ctrl+break.

Note: You cannot use the [include](#) command within a for control construct.

Example

Command input:

```
define int i
for (i = 0; i < 3; i = i + 1)
    printf("%\n", i)
```

Result:

```
00000000H
00000001H
00000002H
```


Related Topics:

[break](#)
[continue](#)
[do while](#)
[while](#)

forward

Declare a forward reference to a debug procedure.

Syntax

```
forward proc [return-type] name
```

Where:

<i>proc</i>	specifies a procedure is being declared.
<i>return-type</i>	specifies the procedure data type returned.
<i>name</i>	specifies the procedure name.

Discussion

Debug procedures must be defined before they can be referenced. Sometimes this isn't practical. The forward command can be used to declare a debug procedure type, so that it can be referenced (without a syntax error), before it is actually defined.

Note: forward is only allowed within a debug procedure definition

Example 1

To reference a debug procedure named max before it is defined.

Command Input:

```
define proc myProc()
{
    forward proc ord4 max // max returns an ord4
    printf("max = %x\n", max())
}

define proc ord4 max()
{
    return 0x10
}

myProc
```

Result:

```
max = 0x10
```

Related Topics

[Data Types](#)

[Debug Procedures](#)
[define](#)

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 printf routine. (See [printf](#) 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)
```

Related Topics:

[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fputc](#)
[fputs](#)
[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)
[printf](#)

fputc

Write a character to a file.

Syntax

```
fputc(char, file_handle)
```

Where:

char is the character to write.

file_handle is the file handle returned from a previous fopen command

Discussion

The fputc function 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

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputs](#)
[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)

fputs

Write a string to a file.

Syntax

```
fputs(string, file_handle)
```

Where:

string is a string constant or nstring variable to write.

file_handle is the file handle returned from a previous fopen command

Discussion

The fputs function 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"
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fread](#)
[fseek](#)
[ftell](#)
[fwrite](#)

fread

Read binary data from a file into an array.

Syntax

```
[result =] fread(buffer, file_handle)
```

Where:

result specifies a debug variable 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.

buffer is an array variable to receive the data read.

file_handle is the file handle returned from a previous fopen command.

Discussion

The fread function reads binary data from a file previously opened by an [fopen](#) command. The data is stored in the array. The length of each item of data read is specified by the size of the array. The returned value is the number of items or data read.

Note: The [feof](#) command should be used to detect end of file.

Example

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)
    ord1 pMem length nItemsRead = Buf
    pMem += nItemsRead
}
fclose (file1)
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)

[fgetc](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fputs](#)
[fseek](#)
[ftell](#)
[fwrite](#)

fseek

Position at a new location in a file.

Syntax

```
[result =] fseek(file_handle, offset, wherefrom)
```

Where:

<i>result</i>	specifies a debug variable of type int4 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.
<i>file_handle</i>	is a file handle returned from a previous fopen command.
<i>offset</i>	is a signed integer specifying a number of bytes.
<i>wherefrom</i>	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

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
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)

[fputs](#)
[fread](#)
[ftell](#)
[fwrite](#)

ftell

Return the current offset within a file.

Syntax

```
[result =] ftell(file_handle)
```

Where:

result specifies a debug variable of type int4 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.

file_handle is the 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

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
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[putc](#)
[puts](#)
[fread](#)
[fseek](#)
[fwrite](#)

fwrite

Write binary data from an array into a file.

Syntax

```
[result =] fwrite(buffer, file_handle)
```

Where:

result specifies a debug variable 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.

buffer is an array variable containing the data to write.

file_handle is a file handle returned from a previous fopen command.

Discussion

The fwrite function writes binary data to a file previously opened by an [fopen](#) command. The data is stored in the array. The returned value is true if successful.

Example

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)
```

Related Topics:

[fclose](#)
[feof](#)
[fgetc](#)
[fgets](#)
[fopen](#)
[fprintf](#)
[fputc](#)
[fputs](#)
[fread](#)

[fseek](#)
[ftell](#)

getc

Read a character from the Command window.

Syntax

```
[result =] getc()
```

Where:

result specifies a debug variable 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.

Discussion

The `getc` function reads a character from the Command window. The character is the return value of the function. `getchar` is an alias for `getc`.

Example

Command input:

```
define ord1 ch
ch = getc()      // wait for key
ch
```

Result:

```
6BH 'k'
```

Related Topics:

[gets](#)
[putchar](#)
[puts](#)

getchar

The getchar command is an alias for the [getc](#) command.

getnearestprogramsymbol

Return the nearest program symbol from a given address.

Syntax

```
[result =] getnearestprogramsymbol(addr, [proc])
```

Where:

<i>addr</i>	is the address to search.
<i>proc</i>	is the processor to use (default = current viewpoint).
<i>result</i>	specifies an nstring variable 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.

Discussion

The `getnearestprogramsymbol` 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:

```
450
```


main

Related Topics:

[evalprogramsymbol](#)

[getprogramsymboladdress](#)

[isprogramsymbol](#)

getprogramsymboladdress

Return the address of the symbol referenced by symbol name.

Syntax

```
[result =] GetProgramSymbolAddress(symbol_name)
```

Where:

result specifies a Pointer variable 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.

symbol_name is 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.

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.

Example 1

Command input:

```
if (IsProgramSymbol("mydata"))
getProgramSymbolAddress("mydata")
```

Result:

```
0xC0001000
```

Example 2

Command input:

```
define nstring s = "mydata"
getProgramSymbolAddress(s)
```

Result:

```
0xC0001000
```

Example 3

Command input:

```
getProgramSymbolAddress("mycode")
```

Result:

```
0xC0008000
```

Example 4

Command input:

```
getProgramSymbolAddress("test")
```

Result:

```
Error "test" is not a program symbol
```

Related Topics:

[evalprogramsymbol](#)

[getnearestprogramsymbol](#)

[isprogramsymbol](#)

gets

Read a string from the user via the Command window.

Syntax

```
[result =] gets(string)
```

Where:

result specifies a debug variable 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.

string is 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.

Example

Command input:

```
define nstring strInput  
gets(strInput)           // user types a line of text
```

Result:

```
this is a line of text
```

Related Topics:

[fgetc](#)
[putchar](#)
[puts](#)

globalsourcepath

Display or edit the global source path map.

Syntax

```
globalsourcepath [= path mappings]
```

Discussion

The globalsourcepath control variable contains a list of source file path mappings. These mappings translate source file paths embedded in program files to actual source file paths on the host computer. The list is a comma-delimited string of the form:

```
programFilePath1 = hostFilePath1; programFilePath2 = hostFilePath2; etc.
```

The path map can also be specified in the Program Load dialog (when the program is loaded), or from Options | Preferences | Program.

Note: This control variable only has an effect when the "Share source file path map among all programs" option is enabled in Options | Preferences | Program.

Example 1

To set the source file path mappings:

Command input:

```
globalsourcepath = "C:\\AA\\WDB\\6.9.2\\wdb32=C:\\AA\\WDB\\6.9.1"
```

Example 2

To display the current source file path mappings:

Command input:

```
globalsourcepath
```

Result:

```
C:\\AA\\WDB\\6.9.2\\wdb32=C:\\AA\\WDB\\6.9.1
```

Related Topics:

[File Menu - Program Menu Item](#)

[Options Menu - Preferences Menu Item](#)

go

Start program execution and optionally set a breakpoint.

Syntax

```
[[px]] go [forever | tilswb | til addr-event]
```

Where:

<i>[px]</i>	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.
<i>addr-event</i>	addr [length] [type]
<i>length</i>	{byte word dword}
<i>type</i>	{acc exe wr io rd smmacc smmexe smmwr smmio}
<i>acc</i>	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.
<i>exe</i>	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.
<i>forever</i>	temporarily disables all breakpoints and begins emulation.
<i>io</i>	specifies that the event to be recognized is an I/O access (read or write) operation at the corresponding port address.
<i>smmacc</i>	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.
<i>smmexe</i>	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.
<i>smmio</i>	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.
<i>smmwr</i>	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.
<i>til</i>	specifies the following event is to be recognized.
<i>tilswb</i>	specifies that emulation continues until a software break is executed. Other breakpoint types are temporarily disabled.
<i>vls-addr</i>	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.
<i>wr</i>	specifies that the event to be recognized is a memory write operation. The default segment selector is the current DS.
<i>byte,word, dword</i>	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 setting address events.

Hardware breaks are implemented using the on-chip debug registers of the processor. Emulation stops before the instruction at addr-event is executed. However, if addr-event 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 the go command is entered without any specifications, any breakpoints specified in the Breakpoints window are in effect.

Example 1

Go til inst @ 1000p is fetched:

Command input:

```
go til 1000p
```

Example 2

Go til a byte read @ 12000p occurs:

Command input:

```
go til 12000p byte rd
```

Example 3

Temporarily disable all breakpoints and go:

Command input:

```
go forever
```

Example 4

Temporarily disable all breakpoints except for softbreaks and go:

Command input:

```
go tilswb
```

Example 5

Start processor P1:

Command input:

```
[p1] go
```

Related Topics

[step](#)

[stop](#)

halt

Cause the processor to terminate program execution.

Syntax

```
halt
```

Discussion

The halt command stops target program execution. The halt command and the [stop](#) command perform the same function.

Example

Command input:

```
go  
halt
```

Related Topics:

[go](#)
[reset](#)
[step](#)
[stop](#)

help

Display the online 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.

Example 1

To open the Help index:

Command input:

```
help
```

Example 2

To open the Help index with the dbgbreak topic selected:

Command input:

```
help dbgbreak
```

Example 3

To open the Help window with the memory window topic selected:

Command input:

```
help memory window
```

homepath

Return the full path of the directory containing the current SourcePoint .ini file.

Syntax

```
homepath
```

Discussion

The homepath control variable contains a string that is the full path to the directory where the SourcePoint .ini file 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 .ini file exists at c:\Program Files\Arium\SourcePoint\sp.ini

Command input:

```
define nstring mymac = homepath + "mac\\big.mac";  
mymac
```

Result:

```
c:\Program Files\Arium\SourcePoint\mac\big.mac
```

Related Topics:

[defaultpath](#)
[macropath](#)
[projectpath](#)

idcode

Display the boundary scan idcode for a device.

Syntax

```
[result = ][[px]] idcode [(device-number)]
```

Where:

<i>[px]</i>	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.
<i>device-number</i>	is the zero-based position of the device in the scan chain.
<i>result</i>	specifies an ord4 debug variable 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.

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

To display current viewpoint processor idcode:

Command input:

```
idcode
```

Result:

```
082E1013H
```

Example 2

To 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

Group and conditionally execute emulator commands.

Syntax

```
if (bool-cond) {commands1} [ else {commands2} ]
```

Where:

<i>bool-cond</i>	specifies a number or an expression which must evaluate as either true (non-zero) or false (zero).
<i>commands1</i>	specifies one or more emulator commands (commands1) that are executed when bool-cond evaluates to true. The braces ({}) are required when you enter multiple commands.
<i>commands2</i>	specifies one or more emulator commands that are executed if bool-cond evaluates to false. The braces ({}) are required when you enter multiple commands.

Discussion

Use the if control construct to 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 using the else option, any commands in the commands2 specification are executed when the specified condition evaluates to false (zero).

If constructs can be nested. 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.

Example 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 = 1
define int1 bb = 2
define int1 zz
if (aa > bb)
{
  zz = aa
} else {
  zz = bb
```

```
}  
zz
```

Result:

02H

Example 2

The following example shows how to use the if control construct with the else clause.

Command input:

```
if (bState)  
{  
printf("bState is true\n")  
} else {  
printf("bState is false\n")  
}
```


include

Execute emulator commands from a text file.

Syntax

```
include [nolist] filename
```

Where:

<i>nolist</i>	suppresses the echoing of commands to the Command window. Nolog has the same effect as nolist.
<i>filename</i>	specifies a filename. See Filenames for details.

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: The emulator displays a syntax error when the **include** command processes an undefined debug variable. Define all debug variables before referencing.

Example 1

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:

SourcePoint 7.12

```
include myfile.mac
```

Related Topics:

[Command Window Introduction](#)

invd

Invalidate the processor's internal caches.

Syntax

```
[[px]] invd
```

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.

Discussion

Use the invd command to invalidate the processor's internal caches. Data held in internal caches is not written back to main memory. In a multiprocessor system, only the caches for the current viewpoint are invalidated. The invd command can only be used when the target is stopped.

Examples

To invalidate P2's internal caches:

Command input:

```
[p2] invd
```

Related Topics

[flush](#)
[wbinvd](#)

irscan

Scan the instruction registers of devices on the JTAG chain.

Syntax

```
irscan(device, instruction)
```

Where:

<i>device</i>	is an int4 that specifies the position of the device to access. Device positions are displayed by the devicelist command.
<i>instruction</i>	is the instruction to be scanned into the instruction register of the device. The instruction can be of any type except strings or arrays.

Description:

Use the irscan command with the drscan command to read from or write to the data register of a device on the target system boundary scan chain. The irscan command writes the designated instruction value into the instruction register of the specified device.

Example 1

To scan an instruction to device 4 on the JTAG chain:

Command Input:

```
irscan(4,5)
```

Example 2

To emulate idcode for device 0 on the JTAG chain:

Command input:

```
idcode(0)
```

Result:

```
182C1013
```

Command input:

```
define ord4 ReadArray[1]
irscan(0,2)
drscan(0, 0x20, ReadArray)
ReadArray[0]
```

Result:

```
470
```

182C1013

Related Topics

[drscan](#)

[msgscan](#)

[tapdateset](#)

[tapdatashift](#)

isdebugsymbol

Determine if a string is the name of a debug variable.

Syntax

```
[result =] isdebugsymbol(symbol)
```

Where:

result is a boolean variable to which the return value is assigned. It is TRUE if the symbol exists, or FALSE if it does not exist.

symbol is a string constant or nstring variable specifying the debug variable name to look up.

Discussion

The isdebugsymbol function checks to see if a debug variable of that name has been defined.

Example 1

Command input:

```
define ord4 x = 5  
isdebugsymbol("x")
```

Result:

TRUE

Example 2

Command input:

```
define nstring s = "x"  
isdebugsymbol(s)
```

Result:

TRUE

Related Topics:

[isprogramsymbol](#)

isem64t

Display whether the specified processor supports Extended Memory 64 Technology.

Syntax

```
[result =] [[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.
- result*** specifies a debug variable 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.

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

Determine if a string is a symbol within a currently loaded program.

Syntax

```
[result =] isprogramsymbol(symbol)
```

Where:

result is a boolean variable to which the return value is assigned. It is TRUE if the symbol exists, or FALSE if it does not exist.

symbol is a string constant or nstring variable specifying the symbol to look up.

Discussion

The isprogramsymbol function 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 function. Here it is assumed that a program is loaded which contains the symbols foo and fun.

Example 1

Command input:

```
isprogramsymbol("foo")
```

Result:

```
TRUE
```

Example 2

Command input:

```
define nstring s = "fun"  
isprogramsymbol(s)
```

Result:

```
TRUE
```

Example 3

Command input:

```
isprogramsymbol("test")
```


Result:

FALSE

Related Topics:

[evalprogramsymbol](#)

[getprogramsymboladdress](#)

[isprogramsymbol](#)

isrunning

Display whether the specified processor is running.

Syntax

```
[result =] [[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.
result	specifies a debug variable 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.

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 display the state of processor P3:

Command input:

```
[P3]isrunning
```

Result:

```
TRUE
```

Example 3

To save the current viewpoint processor state in a user defined variable:

Command input:

```
define ord1 _isrunning  
_isrunning = isrunning
```

Example

To use isrunning in an expression:

Command input:

```
go  
printf("processor is %s\n", isrunning ? "running" : "stopped")
```

Result:

```
processor is running
```

issleeping

Display whether the specified processor is sleeping.

Syntax

```
[result =] [[px]] issleeping
```

Where:

<i>[px]</i>	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.
<i>result</i>	specifies a debug variable 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.

Discussion

Use the issleeping control variable to determine if a specific target processor is sleeping. Entering the command at the command line or in an expression returns 0 for stopped or running or 1 for sleeping.

Example 1

To display the state of the viewpoint processor:

Command input:

```
issleeping
```

Result:

```
FALSE
```

Example 2

To display the state of processor P3:

Command input:

```
[P3]issleeping
```

Result:

```
TRUE
```

Example 3

To use issleeping in an expression:

Command input:

```
printf("processor is %s\n", issleeping ? "sleeping" : "not sleeping")
```

Result:

```
processor is sleeping
```

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 specifies a debug variable 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.

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

Display and define the target JTAG configuration.

Syntax

```
jtagchain([jtag_id][,jtag_id]+)
```

Where:

jtag_id is an expression resolving to a 32-bit JTAG ID

Discussion

The jtagchain function is used to both display and define the target JTAG configuration.

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 sent to the emulator with the [jtagconfigure](#) command as part of target configuration. See [Target Configuration](#).

Example 1

To display the target JTAG chain:

Command input:

```
jtagchain()
```

Result:

```
JTAG Chain:
id=0x0F0F0F0F, IR length=4, max jtag rate=16 Mhz, processor=0x0704-ARM720T
```

Example 2

To define the target JTAG chain with a single device:

Command input:

```
jtagchain(0x0F0F0F0F)
```

Related Topics:

[jtagconfigure](#)

[itagdeviceadd](#)
[itagdeviceclear](#)
[itagdevices](#)
[itagscan](#)
[Target Configuration](#)

jtagconfigure

Synchronize the JTAG configurations between SourcePoint and the emulator.

Syntax

```
[result =] jtagconfigure([force])
```

Where:

<i>result</i>	specifies a boolean variable 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.
<i>force</i>	indicates whether the JTAG configuration table should be forced into the emulator when the existing emulator configuration differs. Default = true.

Discussion

The jtagconfigure function synchronizes the JTAG configurations between SourcePoint and the emulator. If the "force" flag is true, SourcePoint's JTAG configuration replaces any existing configuration in the emulator. If the "force" flag is false the configurations are verified for consistency. In the event of a mismatch, the configurations are presented to the user to select which configuration is to be used. If it succeeds, the emulatorState control variable transitions to state 1.

Example

Command Input:

```
jtagConfigure()                // send JTAG configuration to emulator
```

Result:

```
TRUE                          // command succeeded
```

Related Topics

[autoconfigure](#)
[emulatorstate](#)
[num_jtag_chains](#)
[num_jtag_devices](#)
[Target Configuration](#)

jtagdeviceadd

Add a JTAG ID to the JTAG database.

Syntax

```
jtagdeviceadd(jtag_id, ir_length, processor_id [, max_rate])
```

Where:

<i>jtag_id</i>	is an expression resolving to a 32 bit JTAG ID.
<i>ir_length</i>	is an expression resolving to an integer between 1 and 128.
<i>processor_id</i>	is an expression resolving to a processor id value.
<i>max_rate</i>	is an expression resolving to an integer between 0-40 (MHz).

Discussion

The jtagdeviceadd function 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 JTAG tab under the [Options|Emulator Configuration](#) main toolbar). This argument is optional. If a value is not specified, then 16 MHz is assumed.

Example

In this example, a JTAG ID of 0x09271013 is added. Its IR length is 10 (decimal). Its processor type is 0x60D.

Command input:

```
jtagdeviceadd(0x9271013,10t,0x60D)
```

Related Topics:

[jtagchain](#)
[jtagdeviceclear](#)
[jtagdevices](#)
[Options Menu - Configure Emulator](#)

jtagdeviceclear

Remove a JTAG ID from the JTAG database.

Syntax

```
jtagdeviceclear(jtag_id)
```

Where:

jtag_id is an expression resolving to a 32 bit JTAG ID.

Discussion

The jtagdeviceclear function is useful for dealing with processors that have either duplicate or uninitialized JTAG IDs. This action is persistent. Cycling power on the emulator or restarting SourcePoint does not restore the deleted ID.

Example

Command input:

```
jtagdeviceclear(0)
```

Related Topics:

[jtagdeviceadd](#)

[jtagdevices](#)

jtagdevices

Display the JTAG device database.

Syntax

```
jtagdevices
```

Discussion

The jtagdevices command displays device information from the JTAG device database. There is one line of display per device. The database is maintained in targets\jtag-devices.xml. For each device, the JTAG ID, IR length, max JTAG rate, processor ID and type are shown.

Example

Command input:

```
jtagdevices
```

Result:

```
[all JTAG device definitions]
```

Related Topics:

[jtagdeviceadd](#)

[jtagdeviceclear](#)

[Target Configuration](#)

jtagscan

Direct the emulator to perform device discovery on the JTAG chain.

Syntax

```
[result =] jtagscan([chain])
```

Where:

<i>result</i>	specifies a boolean variable 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.
<i>chain</i>	JTAG chain number {0 1 -1 = all (default)}

Discussion

The jtagscan command causes the emulator to scan the target JTAG chain to determine the devices (e.g. processors) on the chain. If the chain argument is omitted, all chains are scanned. A return value of true indicates the command was successful.

The jtagchain command displays the results of the scan.

Example

Command Input:

```
jtagscan( )           // scan all JTAG chains for devices
```

Result:

```
TRUE                 // scan succeeded
```

Related Topics

[autoconfigure](#)
[jtagchain](#)
[jtagconfigure](#)
[jtagtest](#)
[num_jtag_chains](#)
[num_jtag_devices](#)
[Target Configuration](#)
[verifyjtagconfiguration](#)

jtagtest

Test the target JTAG chain.

Syntax[

```
[result =] jtagtest([chain [, iterations [, test]])]
```

Where:

<i>result</i>	specifies a boolean variable 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.
<i>chain</i>	JTAG chain number {0 1 -1 = all (default)}
<i>iterations</i>	number of iterations (default =1)
<i>test</i>	{0-5 -1}

Where the values of test are:

0	is the target powered
1	is the target currently held in reset
2	return total IR length of the JTAG chain
3	scan JTAG ID codes
4	test JTAG integrity
5	test for adaptive TCK
-1	run all tests except adaptive TCKI

Discussion

The jtagtest command tests the target JTAG chain. Normally it is run with no arguments which does a complete JTAG test. A return value of TRUE indicates the test passed.

Advanced: Tests that return values (e.g., IR length and scan ID codes) require the user to look in the Log window for results. The value of aalog should be 0x20987.

Examples

Command Input:

```
jtagtest( )           // run all JTAG tests
```

Result:

```
TRUE                 // tests passed
```

Command Input:

```
jtagtest(0, 1, 1)    // check if the target is being held in reset
```

Result:


```
TRUE                // test succeeded (target is not held reset)
```

Related Topics

[jtagconfigure](#)

[jtagscan](#)

[verifyjtagconfiguration](#)

[Target Configuration](#)

keys

Simulate keyboard input from within a command file.

Syntax

```
keys("keystring" [, "keystring"]+)
```

Where:

keystring is 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 function 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 keystring, e.g., keys("123") is the same as keys("1", "2", "3").

Examples

Command input:

```
keys("alt", "v", "c")           // opens a Code window
```

Command input:

```
keys("ctrl", "f")               // opens the Find dialog box
keys("123", "enter")            // searches for the string 123
```

last

Return the last address of a symbol.

Syntax

```
[result =] last(symbol)
[result =] last(:module.procedure)
```

Where:

<i>result</i>	specifies a pointer variable 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.
<i>symbol</i>	is a symbolic reference to a program item (label, variable, array, structure, constant, procedure, module, or program).
<i>module</i>	is a symbolic reference to a module.
<i>procedure</i>	is a symbolic reference to a procedure.

Discussion

The last function returns the last address occupied by a program item. 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 the last function does not work unless those variables are in scope.
- Register variables do not have addresses so the last function will not work with them.
- The return value of last when the argument is a non-external procedure in a module that has not 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 does 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.

Example 1

To find 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:

SourcePoint 7.12

```
last(fooStruct)
```

Result:

000080D3

Example 2

To find the first and last address of the procedure *fooFunk*:

Command input:

```
fooFunk
```

Result:

00000240

Command input:

```
last(fooFunk)
```

Result:

00000273

Related Topics

[sizeof](#)

last_jtag_device

Return the device ID of the last JTAG device.

Syntax

```
[result = ] last_jtag_device
```

Where:

result specifies a debug variable 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.

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 nID
  if (num_jtag_devices > 0)
  {
    for (nID=first_jtag_device; nID <= last_jtag_device; nID++)
    {
      printf("%4x %8s (%-11s) port %d, scanchain %d, idcode %x\n",
        devicelist[nID].did,
        devicelist[nID].alias,
        devicelist[nID].devicetype,
        devicelist[nID].debugport,
        devicelist[nID].scanchain,
        devicelist[nID].idcode)
    }
  }
}
```

Related Topics

[first_jtag_device](#)
[num_jtag_devices](#)

left

Extract a number of characters from the beginning of a string.

Syntax

```
[result =] left(string-expr, n)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string-expr</i>	specifies an nstring variable or string constant.
<i>n</i>	specifies the number of characters to extract.

Discussion

The left function 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:

```
define nstring month = "January"  
define nstring temp = left(month, 3)  
temp
```

Result:

"Jan"

Related Topics:

[mid](#)
[right](#)

libcall

Define a function interface to a DLL library.

Syntax

```
libcall(lib, func, [name], ret-type[, {[byref] arg-type}[, ...]][, ...])
```

Where:

<i>lib</i>	is a string expression indicating the library containing the function to import into SourcePoint.
<i>func</i>	is a string expression specifying the function within the library.
<i>name</i>	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.
<i>ret-type</i>	is the return type of function: {void string <i>data-type</i> }
<i>arg-type</i>	is the type of an argument passed to the function {string <i>data-type</i> <i>data-type</i> []}
<i>data-type</i>	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.
<i>data-type</i> []	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  
TestCall11(byref outvalue)
```

Related Topics:

[remove](#)
[show](#)

license

Provide information on the license available with your SourcePoint software.

Syntax

```
license
```

Example

Command input:

```
license
```

Result:

```
FLEXlm License File Information
  Certified:  yes
  File path:  C:\program files\arium\2463.lic
  Emulator serial number:  1311
  Star1:  yes
  Date:  12-18-2010
  Features:  NDA Feature 4
```

linear

Translate an address to a linear address.

Syntax

```
[[px]] linear(addr)
```

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.

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

To translate a real mode virtual address:

Command input:

```
linear(1234:5678)
```

Result:

```
000179b8L
```

Example 2

To translate a protected mode virtual address:

Command input:

```
linear(18h:14h:0)
```

Result:

```
00C03000L
```

Related Topics

[Expressions](#)
[physical](#)

list, nolist

Record command line activity to a file.

Syntax

```
list [[append | overwrite] filename]  
nolist
```

Where:

append appends results to the end of an existing file.
overwrite overwrites an existing file.
filename specifies a filename. See [Filenames](#) for details.

Discussion

Use the list command to log command line activity to a file. This includes commands and any resulting output. Data can be appended to an existing file, an existing file can be overwritten, or a new list file can be created. If the list command is entered without options, the current list file is used.

The nolist command is used to stop logging and close the log file.

Log and nolog are synonyms for list and nolist.

Example 1

To log the results of a memory operation to a file:

Command input:

```
list c:\temp\mem.log  
ord4 0 length 1000  
nolist
```

Example 2

To append the results of a memory operation to an existing file:

Command input:

```
list append "c:\temp\data results.log"  
ord2 1000h length 20h  
nolist
```

Related Topics:

[log, nolog commands](#)
[Log Window Introduction](#)
[flist](#)

load

Load a user program into target memory.

Syntax

```
[[px]] load filename [init] [nocode] [nosym] [AT address | OFFSET expr]
```

Where:

- px* specifies an optional viewpoint override. If the viewpoint override is omitted, the current viewpoint is used.
- filename* specifies a filename. See [Filenames](#) for details.
- init* specifies that registers are to be initialized from values in the loaded 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.
- nosym* specifies that symbols are not loaded into SourcePoint.

Discussion

The following table shows the supported file types. 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.

Memory writes are verified depending on the state of the [verify](#) control variable.

	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

Note: If both nosymbols and nocode options are specified, the file gets loaded as if nocode were specified (symbols only).

Example 1

To load text.elf and initialize processor registers:

Command input:

```
load c:\test\test.elf init
```

Example 2

To load text.elf onto processor 1:

Command input:

```
[p1] load c:\test\test.elf
```

Related Topics:

[reload](#)

[unload](#)

[verify](#)

loadbreakpoints

Load breakpoint information from a file.

Syntax

```
loadbreakpoints(filename)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

Discussion

The loadbreakpoints function loads a list of breakpoints from a file. This function is the equivalent of selecting Load from the Breakpoints window context menu. Any existing breakpoints are overwritten.

Use the [savebreakpoints](#) function to generate a breakpoint file. Breakpoints can also be loaded from an existing project file.

Example

Command input:

```
loadbreakpoints("c:\\temp\\myBreakpoints.brk")
```

Related Topics:

[savebreakpoints](#)

loadlayout

Load a previously saved SourcePoint window layout.

Syntax

```
loadlayout(filename)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

Discussion

The loadlayout function loads a SourcePoint window layout. A window layout is a set of open SourcePoint windows along with their locations, sizes, docking style, etc. The default file extension is .LYT. A set of layout files can be developed, each with a specific debugging purpose in mind, which can be quickly accessed. Although multiple project files can be used to accomplish this same functionality, loading a layout is less disruptive because it only affects windows from the View menu that are open.

Keying in the command closes all existing windows, then opens the windows specified in the layout file in the same size, position, and docking style in which they were saved.

Note: If the loadlayout command is executed from a macro file, it must be the last command in the file.

Example

Command input:

```
loadlayout("mylayout.lyt")
```

Related Topics:

[savelayout](#)

loadproject

Load a SourcePoint project file.

Syntax

```
loadproject([filename])
```

Where:

filename specifies a filename. See [Filenames](#) for details.

Discussion

The loadproject function loads the specified project file. A project file contains all SourcePoint settings including the position and size of each window. If a project file is not specified, then the name of the currently loaded project file is displayed.

Example 1

To load a project file:

Command input:

```
loadProject("c:\\test\\test.prj")
```

Example 2

To display the name of the currently loaded project file:

Command input:

loadproject()

Result:

```
"c:\test\test.prj"
```

Related Topics:

[reloadproject](#)
[unloadproject](#)

loadtarget

Load a target configuration.

Syntax

```
loadtarget(filename)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

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, and Device window files to load. Target configurations are provided by Arium. User-defined target configurations can be created by selecting Options | Save Target Configuration.

Example

Command input:

```
loadtarget("mytargetconfig")
```

loadwatches

Load a set of variables to watch.

Syntax

```
loadwatches(filename, tab)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

tab is a constant or expression specifying the tab number (1-4)

Discussion

The loadwatches command loads the specified watches into a Watch window tab. Watch files can be created by adding variables to the Watch window and either selecting Save in the view, or by using the savewatches command.

Example

To load a set of watches into the Watch 2 tab:

Command input:

```
loadwatches("mywatches", 2)
```

Related Topics:

[savewatches](#)

log, nolog

See [list, nolist](#).

log10

Return the base 10 logarithm of an expression.

Syntax

```
[result =] log10(expr)
```

Where:

result 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.

Note: Values returned by this function are in real8 (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.

Example

Command input:

```
log10(0x20)
```

Result:

```
1.50515
```

Related Topics:

[exp](#)
[loge](#)
[pow](#)
[sqrt](#)

loge

Return the natural logarithm of an expression.

Syntax

```
[result =] loge(expr)
```

Where:

result 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.

Note: Values returned by this function are in real8 (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.

Example

Command input:

```
loge(0x20)
```

Result:

```
3.46574
```

Related Topics:

[log10](#)

logmessage

Display a user-defined message in the Log window.

Syntax

```
logmessage(type, string-expr1, string-expr2)
```

Where:

<i>type</i>	specifies the type of message (see below).
<i>string-expr1</i>	specifies the text to display in the component field; can be an nstring variable or string constant.
<i>string-expr2</i>	specifies the text to display in the message field; can be an nstring variable or string constant.

Discussion

The logmessage function adds a user-defined message in the Log window.

The type argument specifies the log message type:

LOG_ERROR	Display an error message
LOG_WARNING	Display a warning message
LOG_INFO	Display an informational message

Example

To add an error log message in the Log window:

Command input:

```
logmessage(LOG_ERROR, "user macro", "This is a test")
```

macropath

Display the path of 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 is 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"
```

Related Topics:

[defaultpath](#)
[macropath](#)
[projectpath](#)

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:

<i>[px]</i>	is the viewpoint override, including punctuation (<i>[]</i>), specifying that the viewpoint is temporarily set to processor <i>x</i> of the boundary scan chain. The processor can be specified as <i>px</i> (where <i>x</i> is the processor ID), or an alias you have defined for a given processor ID. ALL cannot be used as a viewpoint override.
<i>data-type</i>	specifies the data type used to access memory (e.g., <i>ord1</i> , <i>ord2</i> , <i>ord4</i> , etc.). For more information, see Data Types .
<i>expr</i>	specifies a number or an expression. You can enter more than one expression by using a comma as a separator.
<i>addr-spec</i>	{ <i>addr</i> <i>addr-range</i> }
<i>addr</i>	specifies an address. For more information, see Memory Access: Addresses, found later in this topic.
<i>addr-range</i>	is an address range. There are two ways to specify a range: <i>addr1</i> to <i>addr2</i> or <i>addr</i> length <i>expr</i> .
<i>destination-range</i>	is a range of memory to write.
<i>source-range</i>	is a range of memory to read.
<i>addr1</i> to <i>addr2</i>	specifies a range of memory beginning with address <i>addr1</i> and including address <i>addr2</i> . <i>Addr2</i> must be greater than <i>addr1</i> .
<i>addr</i> length <i>expr</i>	specifies a range of memory beginning with address <i>addr1</i> . The range includes a number of items (specified by <i>expr</i>).
<i>expr</i>	specifies a number or an expression. You can enter more than one expression by using a comma as a separator.
<i>debug-var-array</i>	is an array of debug variables to write to memory (e.g., <i>ord4 data[10]</i>).
<i>display-base</i>	specifies a temporary override of the current display base (<i>bin</i> <i>oct</i> <i>dec</i> <i>hex</i>).

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
00000110 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
```

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
```

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:

```
ord1 200hp = ord1 100hp  
ord1 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 ord1 buf[1000]  
define ptr pMem = 0  
  
file1 = fopen("test.dat", "r")  
while (feof(file1) == 0)  
{  
    nItemsRead = fread(buf, file1)  
    ord1 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 ord1 buf[1000]  
define ptr pMem = 0  
ord1 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.

Related Topics:

[Data Types](#)

messagebox

Display a user-defined message box.

Syntax

```
[result =] messagebox(string-expr [, icon, buttons])
```

Where:

result is an ord4 return value containing the key pressed by the user.

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 function 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 (messagebox checks whether the Yes button was pressed):

Command input:

```
if (messagebox("Yes or No?", MB_ICONQUESTION, MB_YESNO) == ID_YES)
{
    // execute some additional code
}
```

mid

Extract a number of characters from the middle of a string.

Syntax

```
[result =] mid(string-expr, n, m)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string-expr</i>	specifies an nstring variable or string constant.
<i>n</i>	specifies the 0-based index of the first character to extract.
<i>m</i>	specifies the number of characters to extract.

Discussion

The mid function 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:

```
define nstring month = "January"  
define nstring temp = mid(month, 3, 3)  
temp
```

Result:

```
"uar"
```

Related Topics:

[left](#)
[right](#)

msgclose

Complete the construction of a JTAG message.

Syntax

```
[result =] msgclose(msg-handle)
```

Where:

<i>result</i>	is 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.
<i>msg-handle</i>	is 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
msgopen(h)
msgir(h, 7, 2)
msgdr(h, 20, 2)
msgclose(h)
```

Related Topics:

[msgdata](#)
[msgdelete](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgdata

Retrieve 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:

<i>result</i>	is a boolean variable that contains the return value of this command. TRUE indicates the command was successful. FALSE indicates an error occurred (e.g., the specified JTAG message was not found).
<i>msg-handle</i>	is 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.
<i>return-array</i>	is 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 is 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
define ord2 device = 0
msgopen(h)
msgir(h, 8, 2)      // 8=ir length, 2=idcode
msgdr(h, 0n32, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
```

```
define ord1 data[count]  
msgdata(h, data)  
data  
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdelete](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgdelete

Delete a JTAG message.

Syntax

```
[result =] msgdelete(msg-handle)
```

Where:

<i>result</i>	is 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
<i>msg-handle</i>	is 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
msgopen(h)
msgir(h, 4, 2)
msgclose(h)
msgscan(h)
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdata](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgdr

Add a DR scan into an existing JTAG message.

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:

<i>result</i>	is 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.
<i>msg-handle</i>	is 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.
<i>dr-length</i>	is an ord4 that contains the number of bits to be scanned to a data register (DR).
<i>readwrite</i>	is an ord1 that specifies the type of DR scan. See valid readwrite values below.
<i>write-array</i>	is 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.
<i>write-value</i>	is an ord1, ord2, or ord4 value to scan to the data register. If no write-value is specified, then zeros are scanned.
<i>scan-chain</i>	is 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.
<i>stop-state</i>	is 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:

```
// Read JTAG ID from processor
define handle h
define ord2 device = 0
msgopen(h)
msgir(h, 8, 2)      // 8=ir length, 2=idcode
msgdr(h, 0n32, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
define ord1 data[count]
msgdata(h, data)
data
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgdump

Display all scan operations defined in a JTAG message.

Syntax

```
[result =] msgdump([msg-handle])
```

Where:

result is an nstring debug variable that contains the string representations of each scan operation.

msg-handle is 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 msgdump command to display a textual form of the contents of the specified message including any scan results. If msg-handle is not specified, then all existing messages are displayed.

Example

Command input:

```
define handle h
msgopen(h)
msgir(h, 4, 2)
msgclose(h)
msgdump(h)
```

Result:

```
State:   Open
Scanned: No
Scan Operations:
Register Type: IR
Length:   4 bits
Read/Write: WriteOnly
Write Data:  MSB 0010 LSB
           02
Scan Chain: 0
```

Related Topics

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgdr](#)

[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgir

Add an IR scan into an existing 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:

<i>result</i>	is 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.
<i>msg-handle</i>	is 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.
<i>ir-length</i>	is an ord4 that contains the number of bits to be scanned to an instruction register (IR).
<i>write-array</i>	is 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.
<i>write-value</i>	is an ord1, ord2, or ord4 value to scan to the instruction register. If no write-value is specified, then zeros are scanned.
<i>readwrite</i>	is an ord1 that specifies the type of IR scan. See valid readwrite values below.
<i>scan-chain</i>	is 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.
<i>stop-state</i>	is 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:

```
// Read JTAG ID from processor
define handle h
define ord2 device = 0
msgopen(h)
msgir(h, 8, 2)      // 8=ir length, 2=idcode
msgdr(h, 0n32, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
define ord1 data[count]
msgdata(h, data)
data
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgdr](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgopen

Create a new JTAG message.

Syntax

```
[result =] msgopen(msg-handle)
```

Where:

<i>result</i>	is a boolean variable that contains the return value of this command. TRUE indicates the command was successful. FALSE indicates an error occurred.
<i>msg-handle</i>	is 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 a 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 has already been opened (even if it has been closed), an error is reported.

Example

Command input:

```
define handle h
msgopen(h)
msgir(h, 4, 2)
msgclose(h)
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)
[msgscan](#)

msgreturndatasize

Retrieve 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:

<i>result</i>	is a boolean variable that contains the return value of this command. TRUE indicates the command was successful. FALSE indicates an error occurred, such as the JTAG message was not found.
<i>msg-handle</i>	is the name of a previously defined debug variable of type handle.
<i>data-size-var</i>	is an ord4 debug variable that contains the return data size (in bytes). This is a reference parameter and is modified by the msgreturndatasize command.
<i>device-array</i>	is the previously defined ord2 array of device ids (so that multiple devices can be scanned simultaneously).
<i>device-id</i>	is 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
define ord2 device = 0
msgopen(h)
msgir(h, 8, 2)      // 8=ir length, 2=idcode
msgdr(h, 0n32, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
define ord1 data[count]
msgdata(h, data)
data
msgdelete(h)
```

Related Topics:

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgscan](#)

msgscan

Send a JTAG message to the emulator and scan it to the target devices.

Syntax

```
[result =] msgscan(msg-handle, device-array [, return-array])
[result =] msgscan(msg-handle, device-id [, return-array])
```

Where:

<i>result</i>	is 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.
<i>msg-handle</i>	is 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.
<i>device-array</i>	is the previously defined ord2 array of device IDs (so that multiple devices can be scanned simultaneously).
<i>device-id</i>	is the boundaryscanlist device position (ord2) for a single target system device.
<i>return-array</i>	is 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
define ord2 device = 0
msgopen(h)
msgir(h, 8, 2)          // 8=ir length, 2=idcode
msgdr(h, 0n32, 2)
msgclose(h)
msgscan(h, device)
define ord4 count = 0
msgreturndatasize(h, count, device)
define ord1 data[count]
msgdata(h, data)
data
msgdelete(h)
```


Related Topics:

[msgclose](#)
[msgdata](#)
[msgdelete](#)
[msgdr](#)
[msgir](#)
[msgopen](#)
[msgreturndatasize](#)

msr

Display or change the contents of a specified MSR (Model Specific Register).

Syntax

```
[[px]] msr(n) [= 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.

n specifies an MSR number. The use of parentheses is optional.

expr specifies a 64-bit number. Using this option changes the contents of the selected MSR

Example 1

To display the contents of MSR 5:

Command input:

```
msr(5)
```

Result:

```
0000000000000001H
```

Example 2

To display the contents of MSR 5 for P1:

Command Input:

```
[P1] msr(5)
```

Result:

```
0000000000000040
```

Example 3

To change the contents of MSR 1D9H:

Command input:

```
msr(1D9H) = 41H
```

Related Topics

[register access](#)

[Registers Window Introduction](#)

num_activeprocessors

Display the number of active processors (non-sleeping) on the target.

Syntax

```
[result =] num_activeprocessors
```

Discussion

The num_activeprocessors control variable returns an integer representing the number of active processors on the target. This value will be zero when the target has not yet been configured. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_activeprocessors  
nCount
```

Result:

```
00000003H           // 3 processors
```

Related Topics

[num_activeprocessors](#)

[num_devices](#)

[num_jtag_devices](#)

num_all_devices

Display the number of items in the device configuration.

Syntax

```
[result =] num_all_devices
```

Discussion

The num_all_devices control variable returns an integer representing the number of items in the Device Configuration. After apconfigure(), this value will be the same as num_aps. After deviceconfigure(), it will return the sum of JTAG devices, CoreSight Devices, and CoreSight APs. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_all_devices  
nCount
```

Result:

```
00000012H           // 18 devices
```

Related Topics

[devicelist](#)
[num_jtag_chains](#)
[num_jtag_devices](#)
[Target Configuration](#)

num_devices

Display the number of JTAG devices in the target system.

Syntax

```
[result =] 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.

Note: 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
```

Related Topics

[num_jtag_devices](#)
[num_processors](#)

num_jtag_chains

Display the number of configured JTAG chains on the target.

Syntax

```
[result =] num_jtag_chains
```

Discussion

The num_jtag_chains control variable returns an integer representing the number of configured JTAG chains on the target. This value will be zero when the target has not yet been configured. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_jtag_chains  
nCount
```

Result:

```
00000001H           // 1 JTAG chain
```

Related Topics

[num_all_devices](#)

[num_jtag_devices](#)

[Target Configuration](#)

num_jtag_devices

Display the number of configured JTAG TAP devices on the target.

Syntax

```
[result =] num_jtag_devices
```

Discussion

The num_jtag_devices control variable returns an integer representing the number of configured JTAG TAP devices on the target. This value will be zero when the target has not yet been configured. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_jtag_devices  
nCount
```

Result:

```
00000003H           // 3 JTAG devices
```

Related Topics

[devicelist](#)
[num_all_devices](#)
[num_jtag_chains](#)
[Target Configuration](#)

num_processors

Display the number of processors on the target.

Syntax

```
[result =] num_processors
```

Discussion

The num_processors control variable returns an integer representing the number of processors on the target. This value will be zero when the target has not yet been configured. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_processors  
nCount
```

Result:

```
00000003H           // 3 processors
```

Related Topics

[num_jtag_devices](#)
[num_activeprocessors](#)

num_uncore_devices

Display the number of uncore devices on the target.

Syntax

```
[result =] num_uncore_devices
```

Discussion

The num_uncore_devices control variable returns an integer representing the number of uncore devices on the target. Intel processors typically have one uncore per package.

This value will be zero when the target has not yet been configured. This variable is read-only.

Example

Command input:

```
define ord4 nCount = num_uncore_devices  
nCount
```

Result:

```
00000003H           // 3 uncore devices
```

Related Topics

[num_jtag_devices](#)
[Target Configuration](#)

openipc

Open a CLI window to execute Intel CScripts.

Syntax

```
openipc
```

Discussion

The openipc command opens a Python Command Line Interface (CLI) window to execute Intel Customer Scripts (CScripts). Refer to the [OpenIPC Technical](#) note for more information.

Example

Command input:

```
openIPC
```

Result:

```
<CLI window opens>
```

Related Topics

[OpenIPC Integration, Installation and Usage](#)
[OpenIPC Preferences tab](#)

pause

Suspend macro execution until a key is pressed.

Syntax

```
[result =] pause
```

Where:

result specifies a debug variable 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.

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 macro 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:
```

Related Topics:

[getc](#)
[sleep](#)
[wait](#)

physical

Convert an address to a physical address.

Syntax

```
[[px]] physical(addr)
```

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.

addr specifies an address to be translated into a physical address. The parentheses are optional.

Discussion

Use the physical command to convert the specified address 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.
- 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 1

To translate a virtual address:

Command input:

```
physical 1000:1234
```

Result:

```
11234P
```

Example 2

Command input:

```
define ptr addr = 1234
physical(addr+4)
```

SourcePoint 7.12

Result:

11238P

Related Topics

[linear](#)

port

Display or change the contents of an 8-bit I/O port.

Syntax

```
[result =] [[px]] port(io-addr) [= expr]
```

Where:

<i>result</i>	specifies a debug variable of type ord 1 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.
<i>[px]</i>	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.
<i>io-addr</i>	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.
<i>expr</i>	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.

Example 1

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 ". "
```

Example 2

To assign one port value to another port:

Command input:

```
port 90h = port 88h
```

Example 3

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

Related Topics

[dport](#)
[wport](#)

pow

Raises a value by a power.

Syntax

```
[result =] pow(expr, power)
```

Where:

result 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.

power specifies a number or an expression of type real8.

Discussion

The pow function raises the value specified by *expr* to the power specified by *power* (same as y^x on a calculator).

Note: Values returned by this command are in real8 (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.

Example 1

Command input:

```
pow(3,3)
```

Result:

```
27
```

Example 2

Command input:

```
pow(3.1, 4.2)
```

Result:

```
115.803
```

print cycles

Print trace data to a file.

Syntax

```
print cycles [startCycle to endCycle | startCycle length count]
```

Where :

<i>startCycle</i>	is a trace cycle state number.
<i>endCycle</i>	is a trace cycle state number.
<i>count</i>	is an integer expression.

Discussion

The print cycles command "prints" or saves all of a portion of the trace buffer to a file named "trc.txt" in the CWD path. 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 data.

The range is an optional cycle range. If the range is omitted, print cycles defaults to all cycles.

This same functionality is available in the Trace window by selecting Save As in the File menu.

The formatting of the data matches the Trace window. If more than one Trace window is open, the last active window is used.

Examples

Command inputs:

```
print cycles                // prints the entire trace buffer
print cycles 0 length 100T  // print 100 cycles beginning at state 0
print cycles -100T to 0     // print trace from state -100 to state 0
```

Note: In the Trace window, the cycle number is in decimal. But when entered into the Command window, SourcePoint interprets it as a hex number unless there is a "T" on the end.

printf

Write formatted output to the Command window.

Syntax

```
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.

OperatorDescription

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 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

Related Topics:

[fprintf](#)
[putchar](#)
[puts](#)
[sprintf](#)

proc

Display a debug procedure.

Syntax

```
proc proc-name
```

Discussion

The proc command displays a debug procedure (proc-name) that has been previously defined with the [define](#) command.

Example

The following example shows how to define a procedure named "power." This proc returns the result of a value and its exponent.

Command input:

```
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
}
proc power
```

Result:

```
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
}
```


processorcontrol

Specify which processors in the target system are to be controlled by the emulator.

Syntax

```
processorcontrol [= expr]
```

Where:

expr is a mask value indicating which processors are to be controlled by the emulator.

Discussion

The processorcontrol 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.

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.

Example 1

To display the current mask value:

Command input:

```
processorcontrol
```

Result:

```
3          // P0 and P1 are under control of the emulator
```

Example 2

To enable run control of only the second processor in a two processor target.

Command input:

```
processorcontrol = 2
```


processorfamily

Display a string identifying the family to which the processor belongs.

Syntax

```
[result =] processorfamily
```

Where:

result specifies an nstring variable 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.

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 1

Command input:

```
processorfamily
```

Result:

```
P6
```

Example 2

Command input:

```
define nstring family = processorfamily  
family
```

Result:

```
"P6"
```

Related Topics:

[procesortype](#)

processormode

Display a string identifying the operating mode of the current processor.

Syntax

```
[result =] processormode
```

Where:

result specifies an nstring variable 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.

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.

Example

Command input:

```
processormode
```

Result:

```
64 Bit
```

Related Topics

[processorcontrol](#)
[processorfamily](#)
[procesortype](#)

processors

Display the number of processors present in the target system.

Syntax

```
[result =] processors
```

Where:

result specifies an ord4 debug variable 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.

Discussion

The processors control variable displays the number of processors in the current base setting.

Example 1

Note: Assume three processors are present.

Command input:

```
processors
```

Result:

```
0003H
```

Example 2

Command input:

```
define ord2 nCount = processors  
nCount
```

Result:

```
0003H
```

procesortype

Display a string identifying the processor.

Syntax

```
[result =] procesortype
```

Where:

result specifies an nstring variable 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.

Discussion

Use the procesortype command 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 1

Command input:

```
procesortype
```

Result:

```
x86 Family 6 Model 2A(SB)
```

Example 2

Command input:

```
define nstring type = procesortype  
type
```

Result:

```
"x86 Family 6 Model 2A(SB) "
```

Related Topics:

[processorfamily](#)

projectpath

Display the project file path.

Syntax

```
projectpath
```

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"
```

Related Topics:

[defaultpath](#)

[homepath](#)

[macropath](#)

putchar

Display a character in the Command window.

Syntax

```
putchar(char-expr)
```

Where:

char-expr is a quoted character or an expression that evaluates to a character

Discussion

The putchar command displays a character in the Command window.

Example

Command input:

```
define char cvar = 'a'  
putchar(cvar); putchar(cvar+1); putchar('\n')
```

Result:

ab

Related Topics:

[puts](#)
[printf](#)

puts

Display a string in the Command window.

Syntax

```
puts(string-expr)
```

Where:

string-expr specifies an nstring variable, quoted string constant, or an expression that evaluates to a string.

Discussion

The puts command displays a string in the Command window. The '\n' character is required to flush output.

Example

Command input:

```
define nstring date = "6/2/53\n"  
puts(date)
```

Result:

```
6/2/53
```

Command input

```
puts("string constant \n")
```

Result:

```
string constant
```

Related Topics:

[putchar](#)
[printf](#)

rand

Return a random number.

Syntax

```
[result =] rand()
```

Where:

result specifies a debug variable 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.

Discussion

Return 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.

Example

The following example illustrates the [srand](#) and rand functions:

Command input:

```
define int4 card
srand(3)
card = rand()
card
```

Result:

```
16838T           // result may vary
```

Command input:

```
rand()
```

Result:

```
5758T           // result may vary
```

Related Topics:

[srand](#)

readsetting

Read settings within SourcePoint.

Syntax

```
[result =] readsetting(type, name)
```

Where

<i>result</i>	specifies an ord4 variable 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.
<i>type</i>	is an nstring or string constant specifying the type of setting.
<i>name</i>	is an nstring or string constant specifying the setting name.

Discussion

The 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 SourcePoint project file. Names can be obtained by looking in the project file in the emulator configuration section.

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:

```
00000001H      // 1 = Use adaptive TCK
```

Related Topics:

[writesetting](#)

reconnect

Reconnect the emulator to the target.

Syntax

```
reconnect
```

Discussion

The reconnect command connects the emulator to the target. The emulatorstate control variable transitions to state 2 (fully connected). This command has the same effect as pressing the Reconnect button in the Processor toolbar.

Example

Command Input:

```
disconnect          // disconnect from target
reconnect           // reconnect to target
```

Related Topics

[emulatorstate](#)

[disconnect](#)

[Target Configuration](#)

Register Access

Display or change the contents of a processor register.

Syntax

```
[[px]] reg-name [= expr]  
[[px]] reg-name.bit-name [= expr]
```

Where:

<i>[px]</i>	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.
<i>reg-name</i>	specifies the name of a register.
<i>bit-name</i>	specifies the name of a bit within a register.
<i>expr</i>	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. Register and bit names are case insensitive.

Example 1

To display the value of EIP for the current viewpoint processor:

Command input:

```
EIP
```

Result:

```
000002B0H
```

Example 2

To set the value of EIP for processor 1:

Command input:

```
[P1] EIP = 1000  
[P1] EIP
```

Result:

```
00001000H
```

Example 3

To display the value of the ZF bit in the eflags register:

Command input:

```
eflags.zf
```

Result:

```
FALSE
```

Related Topics:

[msr](#)

reload

Reload user program.

Syntax

```
reload
```

Discussion

The reload command repeats the last executed load command. The filename and arguments specified in the [load](#) command are the same.

Example

Command input:

```
load "test.axf"  
reload           // reload test.axf
```

Related Topics:

[load](#)

[unload](#)

reloadproject

Reload 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()
```

Related Topics:

[loadproject](#)

[unloadproject](#)

remove

Remove debug objects.

Syntax

```
remove name
remove {data-type | debug | alias | proc | libcalls} [name]
```

Where:

data-type specifies the variable type to remove (see [Data Types](#)).

debug specifies that all aliases, debug variables and debug procedures are removed.

alias specifies that aliases are to be removed.

proc specifies that debug procedures are to be removed.

libcalls specifies that user-defined procedures are to be removed (see [libcall](#)).

name specifies the name of the object to remove. * and ? can be used as wildcard characters.

Discussion

Use the remove command to remove debug objects created with the [define](#) command. These include debug variables, procedures and alias definitions.

If name is not specified, * is assumed.

Example 1

To remove all debug variables starting with the letters var:

Command input:

```
remove var*
```

Example 2

To remove only the debug procedure showregs:

Command input:

```
remove showregs
```

Example 3

To remove all debug objects (except user-defined procedures used with libcall):

Command input:

```
remove debug
```

Example 4

To remove all procedure definitions:

Command input:

```
remove proc
```

Related Topics:

[#define](#)
[#undef](#)
[define](#)
[libcall](#)
[show](#)

reset

Reset specified target system functions.

Syntax

```

reset emulator
reset [ target [0 | 1] ]
reset tap [(jtag-chain)]

```

Where:

<code>emulator</code>	resets the emulator.
<code>target</code>	resets the target and the target processor.
<code>tap</code>	resets the target Test Access Port (TAP) by asserting/deasserting the TRST signal.
<code>jtag-chain</code>	is 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.)

Example 1

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)

```

Example 2

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)

```

Example 3

To reset all jtag chains:

SourcePoint 7.12

Command input:

```
reset tap
```

Example 4

To reset the jtag chain 0:

Command input:

```
reset tap(0)
```

Example 5

To reset the emulator.

Command input:

```
reset emulator
```

Related Topics:

[go](#)
[halt](#)
[stop](#)

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.

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.axf" has been loaded prior to using the restart command.

Command input:

```
restart      // restart test.axf
```

Related Topics:

[load](#)

[unload](#)

return

Return from a debug procedure.

Syntax

```
return [value]
```

Discussion

Use the return command to return from a debug procedure. If the debug procedure has a return value, a value may optionally be returned.

Example

To take the average of three numbers:

Command input:

```
define proc ord4 avg(a, b, c)
define ord4 a
define ord4 b
define ord4 c
{
    return ((a + b + c) / 3)
}
avg(4, 6, 3)
```

Result:

00000004H

Related Topics:

[Debug Procedures](#)

right

Extract a number of characters from the end of a string.

Syntax

```
[result =] right(string-expr, n)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string-expr</i>	specifies an nstring variable or string constant.
<i>n</i>	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:

```
define nstring month = "January"  
define nstring temp = right(month, 3)  
temp
```

Result:

```
"ary"
```

Related Topics:

[left](#)
[mid](#)

runcontroltype

Display a string identifying the processor.

Syntax

```
[result =] runControlType
```

Where:

result specifies an nstring variable 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.

Discussion

Use the runControlType command to get a unique string that identifies the currently connected debug probe.

Example 1

Command input:

```
runControlType
```

Result:

```
"ECM-XDP3E"
```

Example 2

Command input:

```
define nstring type = runControlType  
type
```

Result:

```
"ECM-XDP3E"
```


safemode

Display or change 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 of 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.

Example 1

To display the current setting:

Command input:

```
safemode
```

Result:

```
FALSE
```

Example 2

To enable safemode:

Command input:

```
safemode=true  
safemode
```

Result:

SourcePoint 7.12

TRUE

Related Topics:

[Options Menu - Target Configuration](#)

save

Save is a synonym for [upload](#).

savebreakpoints

Save the current list of breakpoints to a file.

Syntax

```
savebreakpoints(filename)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

Discussion

The savebreakpoints command saves the current list of breakpoints (displayed in the Breakpoints window) to a file. This command is the equivalent of selecting Save from the Breakpoints window context menu.

Example

Command input:

```
savebreakpoints("c:\\temp\\mybreakpoints.brk")
```

Related Topics:

[loadbreakpoints](#)

savelayout

Save a SourcePoint window layout.

Syntax

```
savelayout(filename)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

Discussion

The savelayout command saves the current window layout. A window layout is a set of open SourcePoint windows along with their locations, sizes, docking style, etc.

Example

Command input:

```
savelayout("mylayout.lyt")
```

Related Topics:

[loadlayout](#)

savewatches

Save a set of variables to watch.

Syntax

```
savewatches(filename, tab)
```

Where:

filename specifies a filename. See [Filenames](#) for details.

tab is a constant or expression specifying the tab number (1-4).

Discussion

The savewatches command saves the specified variables currently displayed in a Watch window tab to a file. This is equivalent to selecting save in the Watch view.

Example

To save the watches in the Watch 2 tab to a file:

Command input:

```
savewatches("watchlist", 2)
```

Related Topics:

[loadwatches](#)

selectdirectory

Open a dialog to select a directory.

Syntax

```
[result =] selectdirectory([startPath])
```

Where:

result specifies an nstring variable 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.

startPath is an nstring variable or string constant specifying the directory to begin the search in.

Discussion

The selectdirectory function displays a dialog to allow the user to specify a directory. The path returned includes a backslash at the end.

If startPath is specified the search begins in that directory. If startPath is not specified the search begins in the last directory accessed.

Example

Command input:

```
define nstring strDir = selectdirectory("c:\program files")
// dialog opens, user selects a directory
strDir
```

Result:

```
"C:\Program Files\test\"
```

Related Topics:

[selectfile](#)

selectfile

Open a dialog to select a file.

Syntax

```
[result =] selectfile([startPath])
```

Where:

result specifies an nstring variable 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.

startPath is an nstring variable or string constant specifying the directory to begin the search in.

Discussion

The selectfile function displays the standard file open dialog to allow the user to specify a filename. The file can then be accessed using the standard file I/O commands.

If *startPath* is specified the search begins in that directory. If *startPath* is not specified the search begins in the last directory accessed.

Example

Command input:

```
define nstring strFile = selectfile("c:\program files")  
// dialog opens, user selects a file  
strFile
```

Result:

```
"C:\Program Files\test\test.dat"
```

Related Topics:

[selectdirectory](#)

shell

Execute an operating system command.

Syntax

```
shell [shell-command]
```

Where:

shell-command specifies any valid shell command.

Discussion

The shell and [dos](#) commands are equivalent.

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.

The shell command without an argument will open a DOS window. The DOS command is a synonym for the shell command.

Examples

Command input:

```
shell cp c:/tmp/test.list /save
```

Command input:

```
shell ls -al
```

Related Topics:

[dos](#)

show

Show definitions and values of debug objects

Syntax

```
show
show name
show {data-type | debugvar | alias | proc} [name]
show {libcalls | devices}
```

Where:

<i>data-type</i>	displays the specific variable type (see Data Types).
debugvar	specifies that only debug variables are shown.
alias	specifies that only alias definitions are shown.
proc	specifies that only debug procedures are shown.
libcalls	specifies that user-defined procedures are shown
devices	specifies that target device names are shown.
<i>name</i>	specifies the name of an existing debug object.

Discussion

Use the show command to display a list of debug objects created with the [define](#) command. These include debug variables, procedures and alias definitions.

Names can use the * and ? as wildcard characters. If name is not specified, * is assumed.

Example 1

To list all of the alias definitions currently defined:

Command input:

```
show alias
```

Result:

```
dog alias "0x1234"
cat alias "0x30000000"
```

Example 2

To display all type ord2 debug variables beginning with the letters var:

Command input:

```
show ord2 var*
```

Result:

```
var1 ord2 "0003H"  
var2 ord2 "0005H"
```

Example 3

To display only the debug procedure declaration for proc1:

Command input:

```
show proc1
```

Result:

```
ord4 proc1(ord2 arg1, ord4 arg2)
```

Related Topics:

[#define](#)

[#undef](#)

[define](#)

[Debug Procedures](#)

[proc](#)

[remove](#)

sin

Return the sine of a radian expression.

Syntax

```
[result =] sin(expr)
```

Where:

result specifies a debug object of type real8 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.

expr specifies a number or an expression of type real8 evaluated in radians.

Discussion

The sin command returns the sine of expr.

Note: Values returned by this 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.

Example

Command input:

```
sin(0)
```

Result:

```
0
```

Related Topics:

[acos](#)
[asin](#)
[atan](#)
[atan2](#)
[cos](#)
[tan](#)

sizeof

Returns the size of a program variable.

Syntax

```
[result =] sizeof(variable)
```

Where:

result 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.

variable specifies a program variable name.

Discussion

The sizeof function returns the size of a program variable. This can be useful for defining breakpoint ranges for composite variables.

Example

Command input:

```
sizeof(myStructure)
```

Result:

```
00000120H
```

Related Topics:

[last](#)

sleep

Pause a macro for a specified time.

Syntax

```
sleep(expr)
```

Where:

expr specifies the number of seconds to sleep.

Discussion

The sleep command pauses a macro for a specified number of seconds. A decimal point is allowed. Resolution is good to 1 ms. The maximum sleep time allowed is 60 seconds. A sleep command may be ended early by pressing ctrl+break.

Example 1

To sleep for 5 seconds:

Command input:

```
sleep(5)
```

Example 2

To sleep for 250 ms:

Command input:

```
sleep(.250)
```

Related Topics:

[getc](#)

[pause](#)

softbreak, softremove, softdisable, softenable

Set, clear, display, enable, and disable soft breakpoints.

Syntax

```
softbreak
softbreak = [ sts, ] location [, name ] [, proc]

softremove [all]
softremove = {name | location | proc} [,...]

softenable = {name | location | proc} [,...]

softdisable [all]
softdisable = {name | location | proc} [,...]
```

Where:

<i>sts</i>	{ e[nabled] d[isabled] }
<i>location</i>	l[ocation] = address
<i>name</i>	n[ame] = <i>breakpoint name</i>
<i>proc</i>	p[rocessor] = { P0 P1 P2 ... }

Discussion

The `softbreak` command sets and displays soft breakpoints (soft breaks). `Softbreak` with no arguments displays a list of the current soft breaks.

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

To display current soft breaks:

```
softbreak
```

To set a soft break at location 12341234:

```
softbreak = location=12341234
```

To remove all soft breaks:

```
softremove
```

To remove soft break at 12341234:

```
softremove = location=12341234
```

To disable all soft breaks:

```
softdisable
```

To disable soft break at 12341234:

```
softdisable = location=12341234
```

To enable soft break at 12341234:

```
softenable = loc=12341234
```

Related Topics:

[Breakpoints Window](#)
[cpubreak commands](#)
[dbgbreak commands](#)
[swremove](#)

sprintf

Write formatted output to an nstring variable.

Syntax

```
[result =] sprintf(nstring, format [, expr [ ,... ] ] )
```

Where:

<i>result</i>	specifies a debug object of type ord4 to which the return value is assigned. If result is not specified, the return value is displayed on the next line.
<i>nstring</i>	is an nstring debug variable.
<i>format</i>	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.
<i>expr</i>	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](#) for more information. The value returned is the number of characters of output generated.

Example

Command input:

```
define ord4 dummy
define nstring mystring
define ord4 o4test = 3
dummy = sprintf(mystring, "this is test %#d", o4test)
mystring
```

Result:

```
this is test #3
```

Related Topics:

[fprintf](#)
[printf](#)
[putchar](#)
[puts](#)

sqrt

Return the square root of an expression.

Syntax

```
[result =] sqrt(expr)
```

Where:

result 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.

expr specifies a number or an expression of type real8.

Discussion

The sqrt command returns the square root of an expression. Sqrt returns 0 (zero) when expr is negative.

Note: Values returned by this command are in real8 (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.

Example 1

Command input:

```
sqrt(64t)
```

Result:

```
8
```

Example 2

Command input:

```
define real8 answer = sqrt(102t)
answer
```

Result:

```
16.0624
```

Related Topics:

[pow](#)

srand

Set the starting point for generating a pseudo-random number using the rand command.

Syntax

```
srand(expr)
```

Where:

expr specifies a number or an expression of type ord4.

Discussion

The srand command sets the starting point for generating a pseudo-random number using the rand command.

Example

Command input:

```
srand(5)  
rand
```

Result:

```
00000036H
```

Command input:

```
rand
```

Result:

```
00007015H
```

Related Topics:

[rand](#)

step

Execute one or more instructions.

Syntax

```
[[px]] step [into | over | out | branch] [step-cnt]
[[px]] step-cmd [step-cnt]
```

Where:

<i>[px]</i>	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.
<i>step-cnt</i>	specifies the number of instructions to step (1-255).
<i>step-cmd</i>	{ stepinto stepover stepoutof bstep istep }
stepinto	step into function calls
stepover	step over function calls
stepout	step out of a function call
bstep	step til the next branch instruction
istep	step into function calls (always low level step)

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 one inst. (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
```

Related topics

[go](#)
[stop](#)

stop

Halt the processor.

Syntax

```
[[px]] stop
```

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 can be used as an override to stop all processors.

Discussion

The stop command stops target program execution.

Note: The stop command and the [halt](#) command perform the same function.

Example

Command input:

```
go til 00000288  
stop
```

Related Topics:

[go](#)
[halt](#)
[step](#)

strcat

Append one string to another.

Syntax

```
strcat(string-expr1, string-expr2)
```

Where:

string-expr1 specifies an nstring variable.

string-expr2 specifies an nstring variable, or a string constant.

Discussion

The strcat command appends the second string (*string-expr2*) to the end of the first string (*string-expr1*).

Example

Command input:

```
define nstring a = "10"  
define nstring b = "22."  
strcat(b, a)  
b
```

Result:

```
"22.10"
```

Related Topics:

[strcpy](#)
[strncat](#)
[strncpy](#)

strchr

Find a character in a string.

Syntax

```
[result =] strchr(string, ch)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string</i>	is an nstring variable, or string constant, to search.
<i>ch</i>	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
```

Related Topics:

[strpos](#)

strcmp

Compare two strings, character by character.

Syntax

```
[result =] strcmp(string-expr1, string-expr2)
```

Where:

<i>result</i>	specifies a debug object of type int2 to which the return value is assigned. If result is not specified, the return value is displayed on the next line of the screen.
<i>string-expr1</i>	specifies an nstring variable or a quoted string constant.
<i>string-expr2</i>	specifies an nstring variable or a quoted string constant.

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 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

Command input:

```
define nstring name1 = "rosenberg"
define nstring name2 = "rosenbaum"
define int4 order = strcmp(name1, name2)
order
```

Result:

```
1T
```

Command input:

```
strcmp(name2, name1)
```

Result:

```
-1T
```

Related Topics:

[strncmp](#)

strcpy

Copy one string into another.

Syntax

```
strcpy(string-expr1, string-expr2)
```

Where:

string-expr1 specifies an nstring variable.

string-expr2 specifies an nstring variable or a quoted string constant.

Discussion

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.

Example

Command input:

```
define nstring month = "October"
define nstring year = "2010"
define nstring date
strcpy(date, month)
date
```

Result:

October

Command input:

```
strcpy(date, year)
date
```

Result:

2010

Related Topics:

[strcat](#)
[strncat](#)
[strncmp](#)

_strdate

Copy the current system date to an nstring variable.

Syntax

```
[result = ] _strdate(string)
```

Where:

<i>result</i>	specifies an nstring variable to which the function return value is assigned. If <i>result</i> is not specified, the return value is displayed on the next line of the screen.
<i>string</i>	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
```

Related Topics

[_strtime](#)
[ctime](#)
[time](#)

string [] (index into string)

Return the nth character in a string.

Syntax

```
[result =] string-expr [index]
```

Where:

<i>result</i>	specifies a debug variable of type char 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.
<i>string-expr</i>	specifies an nstring variable or a quoted string constant.
<i>index</i>	is the character position to return.

Discussion

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.

Example**Command input:**

```
define nstring myString = "Hi There!"
myString[0]
```

Result:

```
'H'
```

Command input:

```
define char myChar
myChar = myString[3]
myChar
```

Result:

```
'T'
```

Related Topics:

[strchr](#)
[strpos](#)

strlen

Return the length of a string.

Syntax

```
[result =] strlen(string-expr)
```

Where:

<i>result</i>	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.
<i>string-expr</i>	specifies an nstring variable, a quoted string constant.

Discussion

The strlen command returns the length of an ASCII string, excluding any null terminating character.

Example 1

Command input:

```
define nstring month = "October"  
strlen(month)
```

Result:

```
7T
```

Example 2

Command input:

```
define nstring year = "2010"  
define int4 answer = strlen(year)  
answer
```

Result:

```
4T
```

_strlwr

Convert a string to lowercase.

Syntax

```
[result = ] _strlwr(string)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string</i>	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
```

Related Topics

[_strupr](#)

strncat

Append the specified number of characters from one string to another.

Syntax

```
strncat(string-expr1, string-expr2, expr)
```

Where:

<i>string-expr1</i>	specifies an nstring variable.
<i>string-expr2</i>	specifies 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.

Discussion

The strncat command appends the specified number of characters (*expr*) from the second string (*string-expr2*) to the end of the first string (*string-expr1*). Copying of the second string continues until a null terminating character is copied or the specified number of character have been copied. The *string-expr2* is left unchanged.

Example

Command input:

```
define nstring a = "10.86"  
define nstring b = "22."  
strncat(b, a, 2)  
b
```

Result:

```
"22.10"
```

Related Topics:

[strcat](#)
[strncat](#)

strncmp

Compare a portion of two strings.

Syntax

```
[result =] strncmp(string-expr1, string-expr2, expr)
```

Where:

<i>result</i>	specifies a debug object of type int2 to which the return value is assigned. If result is not specified, the return value is displayed on the next line of the screen.
<i>string-expr1</i>	specifies an nstring variable or a quoted string constant.
<i>string-expr2</i>	specifies an nstring variable or a quoted string constant.
<i>expr</i>	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 characters have been compared. The return value depends on the difference between the 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 1

Command input:

```
define nstring name1 = "rosenbaum"
define nstring name2 = "rosenberg"
define nstring name3 = "rosen"
strncmp(name1, name2, 7)
```

Result:

```
-1T
```

Example 2

Command input:

```
strncmp(name1, name3, 9)
```

Result:

1T

Example 3

Command input:

```
strncmp(name1, name3, 3)
```

Result:

0T

Related Topics:

[strcmp](#)

strncpy

Copies a portion of one string into another.

Syntax

```
strncpy(string-expr1, string-expr2, expr)
```

Where:

<i>string-expr1</i>	specifies an nstring variable.
<i>string-expr2</i>	specifies an nstring variable or a quoted string constant.
<i>expr</i>	specifies a number or an expression of type int4 that specifies the maximum number of characters to copy.

Discussion

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*.

Example

Command input:

```
define nstring month = "October"
define nstring date
strncpy(date, month, 3)
date
```

Result:

```
"Oct"
```

Related Topics:

[strcat](#)
[strcpy](#)
[strncpy](#)

strpos

Find a character in a string.

Syntax

```
[result = ] strpos(string, ch)
```

Where:

<i>result</i>	specifies an ord4 variable 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.
<i>string</i>	is an nstring variable, or string constant, to search.
<i>ch</i>	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

Related Topics:

[strchr](#)

strstr

Search an ASCII string for the occurrence of a given sub-string.

Syntax

```
[result =] strstr(string-expr1, string-expr2)
```

Where:

<i>result</i>	specifies a debug object of type int4 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.
<i>string-expr1</i>	specifies an nstring variable or a quoted string constant.
<i>string-expr2</i>	specifies an nstring variable or a quoted string constant.

Discussion

A case-sensitive search is performed on string-expr1, looking for string-expr2. If string-expr2 is found within string-expr1, the index of the match is returned. A return value of -1 indicates that string-expr2 was not found.

Example 1

Command input:

```
define nstring string1 = "AaBbCcDdEeFf"
define nstring string2 = "Dd"
define int4 ret_val = strstr(string1, string2)
ret_val
```

Result:

6T

Example 2

Command input:

```
strstr(string1, "DD")
```

Result:

-1T

_strtime

Copy the current system time to an nstring variable.

Syntax

```
[result = ] _strtime(string)
```

Where:

<i>result</i>	specifies an nstring variable 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.
<i>string</i>	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
```

Related Topics

[_strdate](#)

[ctime](#)
[time](#)

strtod

Convert a string into a real8 variable.

Syntax

```
[result =] strtod(string-expr)
```

Where:

<i>result</i>	specifies a debug variable of type real8 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.
<i>string-expr</i>	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.

Example 1

Command input:

```
strtod("1.2")
```

Result:

```
1.2
```

Example 2

Command input:

```
define real8 answer
answer=strtod("23.345")
answer
```

Result:

```
23.345
```

Example 3

Command input:

```
define nstring myString = "4.56 inches"  
strtod(myString)
```

Result:

4.56

Related Topics:

[strtol](#)
[strtoul](#)

strtol

Convert a string into an int4 variable.

Syntax

```
[result =] strtol(string-expr, base)
```

Where:

<i>result</i>	specifies a debug variable of type int4 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.
<i>string-expr</i>	specifies an nstring variable or a quoted string constant.
<i>base</i>	is 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.

Example 1

Command input:

```
strtol("1000", 16t)
```

Result:

```
00001000H
```

Example 2

Command input:

```
strtol("1000", 10t)
```

Result:

```
000003E8H
```

Example 3

Command input:

```
strtol("1000", 8t)
```

Result:

```
00000200H
```

Example 4

Command input:

```
strtol("1000", 2t)
```

Result:

```
00000008H
```

Related Topics:

[strtod](#)
[strtoul](#)

strtoul

Convert a string into an ord4 variable.

Syntax

```
[result =] strtoul(string-expr, base)
```

Where:

<i>result</i>	specifies a debug variable 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.
<i>string-expr</i>	specifies an nstring variable or a quoted string constant.
<i>base</i>	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

1. 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.)
2. 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.

Example 1

Command input:

```
base=10t
strtoul("123", 10)
```

Result:

```
123T
```

Example 2

Command input:

```
base=16t
define ord4 answer
answer = strtoul("0x1000", 16t)
answer
```

Result:

00001000H

Example 3

Command input:

```
base=10t
define nstring myString = "2048 cars"
strtoul(myString, 10)
```

Result:

2048T

Related Topics:

[strtod](#)
[strtol](#)

`_strupr`

Convert a string to uppercase.

Syntax

```
[result = ] _strupr(string)
```

Where:

result specifies an nstring variable 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.

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

Display or modify software breakpoints.

Syntax

```
[[px]] swbreak addr [,addr,...]
```

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.

addr specifies a virtual, physical or symbolic address.

Discussion

Use the swbreak command to set or display software emulation breakpoints. Use the swremove command to remove software breakpoints. A list of all breakpoints can be viewed in the Breakpoints window.

The emulator inserts software breakpoints into memory before entering emulation. When the target processor hits 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.

Caution: 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 hardware breakpoints instead.

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 [softbreak](#) command for enable/disable information.

Example 1

To set a software breakpoint at physical address 1000

Command input:

```
swbreak 1000p
```

Example 2

To display all software breakpoints:

Command input:

```
swbreak
```

Related Topics:

[Breakpoints Window](#)
[softbreak](#), [softremove](#), [softdisable](#), [softenable](#)
[swremove](#)

switch

Cause execution to branch to one of several case statements.

Syntax

```
switch (expr)
{
    case label-expr: [ commands ]
    [ ... ]
    [ default: commands ]
}
```

Where:

<i>expr</i>	Specifies a number or an expression. The value of <i>expr</i> is compared to the value of the label in each case statement.
case label-expr:	Specifies a number or an expression whose value is compared to <i>expr</i> . The colon (:) is required punctuation.
<i>commands</i>	Any emulator commands, including <code>break</code> (which causes an immediate exit from the switch control construct). You cannot use the <code>include</code> command.
default	Specifies the statement that is executed if none of the case statements label-expr: values match that of <i>expr</i> . 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 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.

The [include](#) command is not executable inside the switch control construct.

Example

Command input:

```
define ord4 value = 3
switch(value)
{
case 0:
    printf("saw 0\n")
    break
case 3:
    printf("saw 3\n")
    break
default
    printf("illegal value\n")
    break
}
```

Result:

saw 3

Related Topics:

[break](#)
[if](#)

swremove

Remove software breakpoints.

Syntax

```
[[px]] swremove {all | addr [,addr,...]}
```

Where:

<code>[px]</code>	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.
<code>addr</code>	specifies a virtual, physical or symbolic address.
<code>all</code>	removes all software emulation breakpoints.

Discussion

Use the swremove command to remove software breakpoints. If you specify more than one addr, use a comma as a separator.

Example 1

To remove the software breakpoint at the symbolic address main:

Command input:

```
swremove main
```

Example 2

To remove all software breakpoints:

Command input:

```
swremove
```

Related Topics:

[softbreak](#), [softremove](#), [softdisable](#), [softenable](#)
[swbreak](#)

tabs

Display or change the tab spacing.

Syntax

```
tabs [= expr]
```

Where:

expr specifies a value between 1 and 8, inclusive.

Discussion

The tabs control variable displays or changes the tab spacing for output commands (printf, puts, etc.). The default value for tab spacing is 4. This variable can also be used in an expression.

Example

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("%d\t%d\t%d\n", x, y, z)
tabs = svtabs
```

tan

Return the tangent of a radian expression.

Syntax

```
[result =] tan(expr)
```

Where:

result 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.

Discussion

Values returned by this function are in real8 (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.

Example

Command input:

```
tan(1)
```

Result:

```
1.55741
```

Related Topics:

[acos](#)

[asin](#)

[atan](#)

[atan2](#)

[cos](#)

[sin](#)

tapdatashift

Shift bits into and out of the test access port (TAP) on the JTAG chain.

Syntax

```
tapdatashift(device, register, operation, writeValue, bitCount)
tapdatashift(device, register, operation, byref readValue, bitCount)
```

Where:

<i>device</i>	is an int4 that specifies the position of the device to access. Device positions are displayed by the devicelist command.
<i>register</i>	is an int4 where 0 specifies IR and 1 specifies DR.
<i>operation</i>	is an int4 where 0 specifies write and 1 specifies read.
<i>writeValue</i>	is an int4, ord4, array of int4, or array of ord4 that supplies the bits to shift into the device. It must be large enough to contain the total number of bits specified by the bitCount argument.
<i>readValue</i>	is a reference to a debug variable of type int4, ord4, array of int4, or array of ord4 that will receive the bits shifted out of the device. It must be large enough to contain the total number of bits specified by the bitCount argument.
<i>bitCount</i>	is an int4 that specifies the number of bits to shift into or out of the device.

Discussion

Use tapdatashift to read or write an instruction register or data register of a device on the JTAG chain. The device argument specifies which device is scanned while all other devices remain in bypass. The register argument specifies whether to access the instruction register or data register. The operation argument specifies whether to read or write and whether the fourth argument is a source of bits to write or the destination of bits that are read. The bits shifted into the device are taken from the writeValue argument. The bits shifted out of the device are stored into the readValue argument. You can only read or write, not both at the same time. The bitCount argument specifies how many bits are shifted.

Example 1

To write a 16-bit value into IR of device 0, writeValue=0x55AA:

Command input:

```
tapdatashift(0, 0, 0, 0x55AA, 16t);
```

Example 2

To read a 47-bit value from DR of device 0:

Command input:

```
define ord4 readValue[2];
tapdatashift(0, 1, 1, byref readValue, 47t);
```

Related Topics

[devicelist](#)

[drscan](#)

[irscan](#)

[msgscan](#)

[tapstateset](#)

tapstateset

Manually manipulate the JTAG state machine on the target using the TMS and TCK signals.

Syntax

```
tapstateset(number, state)
```

Where:

number is an int4 that specifies the number of times that the JTAG TMS operation should be carried out.

state is an int4 that specifies what state the state machine should be placed into.

Discussion

Use tapstateset to manually transition the JTAG state machine on the target to the desired JTAG state. The current state that the state machine is in is tracked by the emulator, and a pre-canned sequence is used to move from the current state to the desired state. This sequence is the shortest path to get from the current state to the desired state.

The states and values to use are as follows:

State Name	Value
Test Logic Reset TLR	0x00
Run Test Idle RTI	0x01
Select DR Scan SDR	0x02
Capture DR CDR	0x03
Shift DR ShDR	0x04
Exit-1 DR E1DR	0x05
Pause DR PDR	0x06
Exit-2 DR E2DR	0x07
Update DR UDR	0x08
Select IR Scan SIR	0x09
Capture IR CIR	0x0A
Shift IR ShIR	0x0B
Exit-1 IR E1IR	0x0C
Pause IR PIR	0x0D
Exit-2 IR E2IR	0x0E
Update IR UIR	0x0F

Example 1

Initialize the JTAG state machine on the target by moving to state TLR. This is possible from any other state by issuing five 1's on TMS.

The command needs to be issued only once, so "number" is set to 0. By setting "state" to 0, the emulator will issue the required number bits, using the required pattern to move the target state machine to the TLR state.

Command input:

```
tapstateset(0, 0)
```

Example 2

Assuming that the target is now in the TLR state, transition the state machine from TLR to SIR, via RTI, and then back to RTI, allowing the pre-canned sequences to maneuver us through the correct states.

Command input:

```
tapstateset(0, 0x0B)  
tapstateset(0, 0x01)
```

Example 3

Perform the same operations as Example 2, but don't let the pre-canned sequences do the work. Perform each transition manually, in order to show both the path that the pre-canned sequences take above, and the level of granularity available using this command.

Command input:

```
tapstateset(0, 0x01)  
tapstateset(0, 0x02)  
tapstateset(0, 0x09)  
tapstateset(0, 0x0A)  
tapstateset(0, 0x0B)  
tapstateset(0, 0x0C)  
tapstateset(0, 0x0D)  
tapstateset(0, 0x0E)  
tapstateset(0, 0x0F)  
tapstateset(0, 0x01)
```

Related Topics

[drscan](#)
[irscan](#)
[msgscan](#)
[tapdatashift](#)

targpower

Display whether the target is powered.

Syntax

```
targpower
```

Discussion

The targpower control variable indicates whether the target is powered on. This is a read-only variable. This control variable can be used in an expression.

Example

Command input:

```
targpower
```

Result:

```
TRUE
```

Related Topics:

[targstatus](#)

targstatus

Display the status of the target.

Syntax

```
targstatus
```

Discussion

The targstatus control variable returns a string indicating the current target status. This is a read-only variable. This variable can be used in an expression.

The following is a list of possible status strings returned by targstatus:

```
NoPower
Waiting
Stopped
Running
Stepping
Flushing
Halting
Resetting
Sleeping
ShutdownPending
Shutdown
```

Example 1

Command input:

```
targstatus
```

Result:

```
Stopped
```

Example 2

Command input:

```
go
targstatus
```

Result:

```
Running
```


Related Topics:

[isrunning](#)
[targpower](#)

taskattach

Cause the debugger to attach to and control a task already running on the target operating system.

Syntax

```
[result =] taskattach(filename, pid)
```

Where:

<i>result</i>	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.
<i>filename</i>	specifies a filename. See Filenames for details.
<i>pid</i>	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

Set, clear, display, enable and disable task breakpoints.

Syntax

```
taskbreak
taskbreak = [ name, ] [ sts, ] location [, task] [, macro]

taskremove [all]
taskremove = { name | location | task | macro } [ ,... ]

taskenable = { name | location | task | macro } [ ,... ]

taskdisable [all]
taskdisable = { name | location | task | macro } [ ,... ]
```

Where:

<i>name</i>	<i>Breakpoint name</i>
<i>sts</i>	{ e[nabled] d[isabled] }
<i>task</i>	p[rogram] = <i>program name</i>
<i>location</i>	l[ocation] = <i>address</i>
<i>macro</i>	f[file] = <i>path of macro file to execute when break hits</i>

Discussion

The taskbreak command sets and displays task breakpoints. Taskbreak with no arguments displays a list of the current task breakpoints.

The taskremove command removes any or all of the task breakpoints. Arguments to this command qualify which task breakpoints are to be removed. For instance, taskremove=l=1002, n=Break01 removes the task breakpoint with the name Break01 and address = 1002. Taskremove with no arguments removes all task breakpoints.

The taskenable command selectively enables task breakpoints. Arguments to this command qualify which task breakpoints are to be affected. For instance, taskenable=f=c:\OnBreak.mac enables only task breakpoints that will run the macro C:\OnBreak.mac on break.

The taskdisable command selectively disables task breakpoints. Arguments to this command qualify which task breakpoints are to be affected. For instance, taskdisable=p=/home/hello disables only task breakpoints for the task /home/hello. If no arguments are specified, all task breakpoints are disabled.

Task breakpoints 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 breakpoint is inactive, i.e., the breakpoint does not apply to the current task. This is because if a task is out of context, its address space is not valid.

Examples

To display current task breakpoints:

```
taskbreak
```

To set a task breakpoint at address 1234 for the current task that will run the macro OnBreak.mac on break:

```
taskbreak = location=1234, f=C:\OnBreak.mac
```

To set a task breakpoint at address 1000p for the task /bin/ls:

```
taskbreak = l=1000p, p=/bin/ls
```

To remove all task breakpoints:

```
taskremove
```

To remove all task breakpoints associated with task /home/hello:

```
taskremove = p=/home/hello
```

To disable all task breakpoints:

```
taskdisable
```

To disable task breakpoint with name firstBreak:

```
taskdisable = n=firstBreak
```

To enable all task breakpoints with task /home/hello:

```
taskenable = p=/home/hello
```

To enable all task breakpoints that will run the macro file C:\OnBreak.mac on break:

```
taskenable = f=C:\OnBreak.mac
```

taskend

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

To stop debugging the target operating system program whose viewpoint is 40H.

Command input:

```
taskend (40H)
```

Result:

```
0001H
```

Related Topics:

[taskattach](#)
[taskgetpid](#)
[taskstart](#)

taskgetpid

Retrieve 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

To get the PID from the target operating system program with viewpoint 40H.

Command input:

```
taskgetpid(40H)
```

Result:

```
0007H
```

Related Topics:

[taskattach](#)

[taskend](#)

[taskstart](#)

taskstart

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:

<i>vp</i>	specifies the debug object of type ord4 to which the function return value is assigned. If <i>vp</i> 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.
<i>filename</i>	specifies a filename. See Filenames for details.
<i>targetpath</i>	is a string that specifies the path and filename of the program on the target operating system to start.
<i>arguments</i>	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 (argument) 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

Display or change the emulator's current JTAG clock rate.

Syntax

```
tck [= clockrate]
```

Where:

clockrate is 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 box under Options | Emulator Configuration. Click on the JTAG Clock tab.

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, see "[Options Menu-Emulator Configuration](#)."

Example 1

To display the current setting:

Command input:

```
tck
```

Result:

```
2.0 MHz
```

Example 2

To set the JTAG clock rate to 12.0 MHz:

Command input:

```
tck = "12.0 MHz"  
tck
```

Result:

12.0 MHz

Related Topics:

[readsetting](#)
[writesetting](#)

time

Return the current calendar time as an ord4 value.

Syntax

```
[result =] time()
```

Where:

result specifies a 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.

Discussion

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 an ord4 data type.

Example

To time an operation (in seconds):

Command input:

```
define ord4 startTime = time  
  
(operation to time)  
  
printf("time = %d seconds\n", time - startTime)
```

Related Topics:

[ctime](#)

uncoreconfigure

Synchronize the uncore configuration between SourcePoint and the emulator.

Syntax

```
[result =] uncoreconfigure([force])
```

Where:

<i>result</i>	specifies a boolean variable 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.
<i>force</i>	indicates whether the uncore configuration should be forced into the emulator when the existing emulator configuration differs. Default = true.

Discussion

The uncoreconfigure function synchronizes the uncore configuration between SourcePoint and the emulator. If the "force" flag is true, SourcePoint's uncore configuration replaces any existing configuration in the emulator. If the "force" flag is false the configurations are verified for consistency. In the event of a mismatch, the configurations are presented to the user to select which configuration is to be used.

Example

Command Input:

```
uncoreConfigure()                // send uncore configuration to emulator
```

Result:

```
TRUE                             // command succeeded
```

Related Topics

[autoconfigure](#)
[num_uncore_devices](#)
[uncorescan](#)
[Target Configuration](#)

uncorescan

Direct the emulator to perform uncore discovery on the JTAG chain.

Syntax

```
[result =] uncorescan([chain])
```

Where:

<i>result</i>	specifies a boolean variable 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.
<i>chain</i>	JTAG chain number {0 1 -1 = all (default)}

Discussion

The uncorescan command causes the emulator to scan the target JTAG chain for uncore devices. If the chain argument is omitted, all chains are scanned. A return value of true indicates the command was successful.

This command is acceptable only when the emulatorstate control variable is set to 1, following a jtagconfigure operation.

Example

Command Input:

```
uncorescan( )           // scan all JTAG chains for uncore devices
```

Result:

```
TRUE                     // scan succeeded
```

Related Topics

[autoconfigure](#)
[num uncore devices](#)
[uncoreconfigure](#)
[Target Configuration](#)

#undef

Remove a debug alias definition.

Syntax

```
#undef alias-name
```

Where:

alias-name specifies a previously defined alias.

Discussion

The #undef command is used to remove an alias definition that was created with the [#define](#) command. The [remove](#) command can also be used to remove multiple alias definitions.

Example

Command input:

```
#define ld load c:\src\targdev      // define alias
#undef ld                          // remove alias
```

Related Topics:

[#define](#)
[define](#)
[include](#)
[remove](#)
[show](#)

unload

Unload program(s) from SourcePoint.

Syntax

```
[[px]] unload [all | filename]
```

Where:

<i>px</i>	specifies an optional viewpoint override. If the override is omitted, the current viewpoint is assumed.
<i>all</i>	specifies that all programs are to be unloaded.
<i>filename</i>	specifies a filename. See Filenames for details.

Discussion

The unload command removes one or more programs from SourcePoint. This includes all associated source and symbol information.

Example 1

To unload text.axf:

Command input:

```
unload c:\test\test.axf
```

Example 2

To unload text.axf from processor 1:

Command input:

```
[p1] unload c:\test\test.axf
```

Example 3

To remove all programs from all processors:

Command input:

```
unload all
```

Related Topics:

[load](#)
[reload](#)

unloadproject

Unload the current SourcePoint project file.

Syntax

```
unloadproject
```

Discussion

The unloadproject command unloads the current project file. All windows, except for the Log and Command views, 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
```

Related Topics:

[loadproject](#)
[reloadproject](#)

upload

Save target memory to a file.

Syntax

```
[[px]] upload filename [addr to addr | addr length expr] [overwrite]
```

Where:

<i>[px]</i>	is the viewpoint override, including punctuation (<code>[]</code>), specifies that the viewpoint is temporarily set for this command to the specified processor. The processor can be specified as <code>px</code> (where <code>x</code> is the processor ID), or an alias you have defined for a given processor ID.
<i>filename</i>	specifies a filename. See Filenames for details.
<i>addr</i>	specifies the address of the first byte.
<i>to addr</i>	specifies the ending address of a range of addresses. The <code>addr</code> after the "to" reserved word must be greater than the <code>addr</code> before the "to" reserved word.
<i>length expr</i>	specifies the number of bytes desired.
<i>overwrite</i>	specifies that filename is to be overwritten, if it exists.

Discussion

Use the upload command to transfer the contents of target memory to a host file. The contents can be saved as a binary file, or as an axf file.

Saving as an axf file has advantages when performing trace disassembly in the Trace View. If the "Disassembly Uses" setting is set to "Use Cached Program", then when an axf file is loaded, SourcePoint will read memory from the file rather than the target. This is useful when the area traced has been overlaid and is no longer present in the target.

If the file specified by filename already exists, upload does not overwrite the file nor prompt for an overwrite, but instead an error is reported. To overwrite an existing file, you must specify the overwrite option. The overwrite option causes the file to lose its original contents.

Example 1

To save the contents of 0x300 bytes of target memory starting at 1000p to a file named mypatch.bin, and then reload it at the same address:

Command input:

```
upload mypatch.bin 1000p length 0x300
load mypatch.bin at 1000p nopsp noinit
```

Example 2

To save 1 Mb of memory at address 0x80000000:

SourcePoint 7.12

Command input:

```
upload range1.axf 0x80000000 length 0x100000
```

Related Topics:

[load](#)

[reload](#)

use

Set 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.

Note: The use control variable is identical in function to the [asmmode](#) control variable.

Example

To set the use control variable to interpret addresses as 32-bit:

Command input:

```
use = use32
```

Related Topics

[asmmode](#)

verify

Verify writes to target memory.

Syntax

```
verify [= true | false]
```

Where:

true	specifies that memory writes to target memory are verified.
false	specifies that target memory writes are 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. 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.

Example 1

To display the current value of verify:

Command input:

```
verify
```

Result:

```
false
```

Example 2

To change verify to true:

Command input:

```
verify = true
```

Related Topics:

[asm](#)
[load](#)
[Memory Access](#)

verifydeviceconfiguration

Verify that SourcePoint and emulator device configurations match.

Syntax

```
[result =] verifydeviceconfiguration()
```

Where:

result specifies a boolean variable 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.

Discussion

The verifydeviceconfiguration function verifies that SourcePoint and emulator device configurations match. In the event of a mismatch, the configurations are presented to the user to select which configuration is to be used. This operation is valid when the emulatorstate control variable is set to 1 or 2.

Related Topics

[deviceconfigure](#)

[devicelist](#)

[devicescan](#)

[Target Configuration](#)

verifyjtagconfiguration

Verify that SourcePoint and emulator JTAG configurations match.

Syntax

```
[result =] bool verifyjtagconfiguration()
```

Where:

result specifies a boolean variable 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.

Discussion

The verifyjtagconfiguration function verifies that SourcePoint and emulator JTAG configurations match. In the event of a mismatch, the configurations are presented to the user to select which configuration is to be used. This operation is valid when the emulatorstate control variable is set to 1 or 2. If the user elects to use the SourcePoint configuration, the emulatorState control variable transitions to state 1.

Example

Command Input:

```
verifyjtagconfiguration
```

Result:

```
TRUE           // JTAG configurations match
```

Related Topics

[emulatorstate](#)
[jtagconfigure](#)
[jtagscan](#)
[num_jtag_devices](#)
[Target Configuration](#)

version

Return the current version of SourcePoint and emulator flash.

Syntax

```
version
```

Discussion

Use the version command to display the current SourcePoint and emulator firmware version numbers.

Example

Command input:

```
version
```

Result:

```
SourcePoint v6.8.0.5959 Build 1816 Official;  
Emulator: Boot: v4.00.00.01, Flash: v6.08.07.190
```


viewpoint

Display or change the current viewpoint.

Syntax

```
viewpoint [= {Pn | expr | alias-name}]
```

Where:

<i>Pn</i>	specifies the nth processor of the boundary scan chain.
<i>expr</i>	is an expression resolving to a processor number (0-n4)
<i>alias-name</i>	is a processor alias name (see <code>vpalias</code>)

Discussion

Use the viewpoint command 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 window.

Code, Memory and Register windows can be set to display data from a particular processor, or set to track the viewpoint processor. Changing the viewpoint causes all tracking windows to update.

Example 1

To display the current viewpoint:

Command input:

```
viewpoint
```

Result:

```
0001H
```

Example 2

To change the viewpoint to processor 2:

Command input:

```
viewpoint = P2
```

Related Topics:

[vpalias](#)

[Viewpoint Window Introduction](#)

vpalias

Display or change a viewpoint alias

Syntax

```
[[px]] vpalias [=expr]
```

Where:

<code>[px]</code>	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.
<code>expr</code>	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.

The first character of an alias name must be a letter. Subsequent characters can be letters or numbers. Alias names are case-insensitive. 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 processor p3 "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>
```

Related Topics:

[Options Menu - Target Configuration
viewpoint](#)
[Viewpoint Window Introduction](#)

wait

Suspend command execution until a breakpoint is encountered.

Syntax

```
wait [time]
```

Where:

time specifies a number of seconds to wait.

Discussion

Use the wait command to prevent the emulator from accepting commands until a breakpoint occurs. This command is 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.

If a time is specified, then command execution is suspended until a breakpoint hits, or until the time to wait is exceeded.

The wait command has no effect when emulation is stopped.

Related Topics:

[go](#)
[Debug Procedures](#)
[proc](#)
[sleep](#)

wbinvd

Write back and invalidate the processor's internal caches.

Syntax

```
[[px]] wbinvd
```

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.

Discussion

Use the wbinvd command to write back and invalidate the processor's internal caches. In a multiprocessor system, only the caches for the current viewpoint are invalidated. The wbinvd command can only be used when the target is stopped.

Examples

To write back and invalidate the viewpoint processor's cache:

Command input:

```
wbinvd
```

Related Topics

[flush](#)

[invd](#)

while

Group and execute commands while a condition is true.

Syntax

```
while(bool-cond) {commands}
```

Where:

<i>bool-cond</i>	specifies a number or an expression that is evaluated and tested. The loop repeats until <i>expr</i> evaluates to false (zero). The parentheses are required.
<i>commands</i>	specifies any emulator commands. When you enter more than one command, you must enclose them in braces (<code>{ }</code>).

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). To break out of a loop press ctrl+break.

Note: The [include](#) command is not executable inside a while control.

Example 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
```

Related Topics:

[break](#)
[continue](#)
[for](#)
[if](#)

windowrefresh

Control timed refresh of windows.

Syntax

```
windowrefresh([time])
```

Where:

time is an ord4 value specifying a refresh interval in seconds.

Discussion

Use the windowrefresh command to control automatic refresh of all windows in SourcePoint. When windows are refreshed, processor state (e.g. memory, registers, etc) are re-read from the target.

If a time (refresh interval) is not specified, then the windows are refreshed once. A time of 0 disables automatic refresh.

Example 1

To refresh all windows once:

Command input:

```
windowrefresh()
```

Example 2

To enable automatic window refresh of all windows every 5 seconds:

Command input:

```
windowrefresh(5)
```

Example 3

To disable automatic window refresh:

Command input:

```
windowrefresh(0)
```


wport

Display or change the contents of a 16-bit I/O port.

Syntax

```
[result =] [[px]] wport(io-addr) [= expr]
```

Where:

<i>result</i>	specifies an debug variable of type ord2 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.
<i>[px]</i>	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.
<i>io-addr</i>	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.
<i>expr</i>	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.

Example 1

To assign a 16-bit value to a port:

Command input:

```
wport 88h = 4321h
```

Example 2

To assign one port value to another port:

Command input:

```
wport 90h = wport 88
```

Example 3

To create a debug variable named portvar and assign a port value to it:

Command input:

SourcePoint 7.12

```
define ord2 portvar  
portvar = wport 90  
portvar
```

Result:

4321H

Related Topics

[dport](#)
[port](#)

writesetting

Modify settings within SourcePoint.

Syntax

```
writesetting(type, name, value)
```

Where:

type is an nstring or string constant specifying the type of setting.
name is an nstring or string constant specifying the setting name.
value is an ord4 specifying the value to assign to the setting.

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 SourcePoint 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 sets 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)      // 0 = Don't use adaptive TCK
```

Related Topics:

[readsetting](#)

yield

Allow another process to run inside macro files.

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.

Related Topics:

[include](#)
[yieldflag](#)

yieldflag

Yield on each loop iteration.

Syntax

```
yieldflag [= true | false]
```

Where:

true	specifies that macros yield at every loop (for, while) iteration.
false	specifies that macros yield control only when an explicit yield command is encountered.

Discussion

The state of yieldflag affects macro file execution. When a macro is running, generated output only appears in the Command window when the macro yields. This allows windows to repaint.

When yieldflag is true, macros yield automatically 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.

Example 1

To display the current setting:

Command input:

```
yieldflag
```

Result:

```
TRUE
```

Related Topics:

[include](#)
[yield](#)

Index

#

#define.....386

#undef663

-

_strlwr.....621

_strtime.....629

_strupr637

A

aadump317

abort318

abs.....319

acos.....321

advanced.....322

AET Tab205, 215

Array.....299

asin.....323

asm.....324

asmmode.....332

ASSET InterTech - contacting 1

atan333

atan2334

autoconfigure335

B

base.....336

bell (beep).....339

bits340

Bookmarks282

bool294, 517

boolean294, 517

break342

Break on.....105

breakall343

Breakpoints105

 Adding/editing.....108

 Bus analyzer.....105

 Emulator105

 Opening.....105

 Processor110

 Set Breakpoints From Other SourcePoint
 Windows112

 Software110

 Types.....105, 110

Breakpoints Tab.....48

BTS Tab.....214

Bus analyzer (breakpoint type).....105

Bus analyzer sequence105

byte294, 517

C

cachememory345

cause347

char	294, 517	cos	358
character functions	349	cpubreak	359
Character strings	292	cpudisable	359
clock	356	cpuenable	359
Code disassembly	125	cpuid_eax.....	361
Code Tab.....	50	cpuid_ebx.....	363
Code window.....	115	cpuid_ecx.....	365
Icon definitions.....	118	cpuid_edx.....	367
Menu	119	cpuremove	359
Opening	115, 124, 144, 200, 202, 203	createprocess	369
Preferences	123	cscfg.....	370
Saving contents	127	CScripts	264
Saving settings	126	csr	377
Colors Tab.....	54	ctime	379
Command Files	307	Ctrl+break	129
Command Line Arguments	99	Customize the Registers Window.....	191
Command window.....	129	Cut	38
Comments	291	cwd.....	380
Confidence tests	79, 133	D	
Confidence test failures and symptoms	139	Data qualification	105
Confidence tests details	136	Data Types.....	294
Constants	292	DbC	285
continue.....	357	dbgbreak	382
Control Variables.....	305	dbgdisable.....	382
Copy	20	dbgenable	382

dbgremove	382	Docking windows	15
DCI	285	dos	401
Debug pointer types	298	double	294, 517
Debug Procedures	301	dport	402
Debug registers	105	drscan	403
Debug Variable Arrays	299	E	
Debug Variables	298	ECM-XDP(3) Tab	69
defaultpath	384	edit	38
define	387	Edit menu	38
definemacro	389	editor	406
Descriptor tables	141	EFI Framework debug	251
Cache	249	emubreak	407
Menu	144	emudisable	407
Opening	141	emuenable	407
Replacing	146	Emulator - flash update	34
deviceconfigure	391	Emulator - resetting	78
devicelist	392	Emulator Configuration	64
Devices window	147	ECM-XDP(3) Tab	69
Accessing in Command window	159	General	64
Creating - an example	161	Information Tab	72
Menu	157	JTAG Clock Tab	67
devicescan	396	JTAG Tab	65
disconnect	397	Network Tab	71
displayflag	398	Switches Tab	71
do while	399	Target Reset Tab	68

Emulator connection - adding	83	Flash - updating emulator	34
Emulator connection - verifying	103	Flash - updating target	57
Emulator connections	76	flist	429
Emulator Tab - Preferences	47	float	294, 517
Emulator Tab - Trace Configuration	211	flush	431
emulatorstate	409	fopen	432
emuremove	407	for	434
encrypt	410	forward	436
error	411	fprintf	438
eval	412	fputc	439
evalprogramsymbol	413	fputs	440
execution point (\$)	415	fread	441
exit	36, 417	fseek	443
exp	418	ftell	445
Expressions	296	Functions	301
F		fwrite	446
fc 419		G	
fclose	420	GDT	141
feof	422	getc	448
fgetc	424	getchar	449
fgets	426	getnearestprogramsymbol	450
File - closing	420	getprogramsymboladdress	452
File menu	20	gets	454
Filenames	308	globalsourcepath	455
first_jtag_device	428	go	44, 457

H

halt.....460

help.....461

homepath462

I

Icon groups - editing.....94

Icons.....196

Icons - arranging80

Icons - displaying text.....93

idcode.....463

IDT.....141

if 465

include467

Index into.....619

Information Tab72

int1.....294, 517

int16.....294, 517

int2.....294, 517

int4.....294, 517

int8.....294, 517

Intel PT Memory Tab.....222

Intel PT Tab.....205, 219

Interrupt acknowledge (breakpoint type) 105, 110

invd.....469

IPC55, 266

irscan470

isalnum.....349

isalpha.....349

isascii349

isctrl.....349

isdebugsymbol472

isdigit.....349

isem64t473

islower349

isprint349

isprogramsymbol.....474

ispunct.....349

isrunning476

issleeping478

issmm.....480

isspace.....349

isupper349

isxdigit349

itpcompatible482

J

JTAG (CCC) Tab73

JTAG Clock Tab.....67

JTAG Tab.....65

jtagchain.....483

jtagconfigure485

jtagdeviceadd	486	loadproject	509
jtagdeviceclear	487	loadtarget	510
jtagdevices	488	loadwatches	511
jtagscan	489	local_cscfg	370
jtagtest	490	log	512
K		Log window	163
keys	492	Icon definitions	165
L		Menu	166
last	493	log10	513
lastjtagdevice	495	loge	514
Layout	22	logmessage	515
LBR Tab	212	M	
LDT	141	Machine level step into	44
LDTR	141	Machine level step over	44
left	496	Macro commands	20
libcall	497	macropath	516
license	500	Macros	307
License File Dialog	81	Configure Macros	28
Licensing	271	Event Macros Tab	28
linear	501	User-Defined Macros Tab	28
list	503	Load Macro	28
load	505	Memory Access	517
Load target configuration files	62	Memory casting	257
loadbreakpoints	507	Memory map	57
loadlayout	508	Memory Map Tab	57

Memory Tab51

Memory window 169

 Changing values 176

 Display fields 169

 Menu 171

 Opening 174

 Preferences 173

 Size and radix 171

 Target memory - copying.....517

 Target memory - filling.....517

 View memory at address 175

messagebox522

mid.....524

msgclose525

msgdata.....526

msgdelete.....528

msgdr529

msgdump.....531

msgir.....533

msgopen.....535

msgreturndatasize.....536

msgscan538

msr540

Multi-clustering261

N

Network Tab..... 71

New project wizard..... 21

 Using 100

nolist.....503

nolog512

nstring 294, 517

num_activeprocessors542

num_all_devices543

num_devices544

num_jtag_chains.....546

num_jtag_devices547

num_processors548

num_uncore_devices.....549

O

openipc550

Operands 296

Operators 296

Options menu item..... 45

ord1 294

ord12 294, 517

ord16 294, 517

ord2 294

ord4 294

ord8 294

P

Page translation window	177	Print preview.....	32
Opening	177	Print setup	32
Paste	38	Print a Register List.....	191, 192
pause.....	551	print cycles	558
PCI devices window	179	printf	559
Menu	183	proc	562
Opening PCI registers view	184	Procedures, debug.....	301
PCI registers	183	Processor determination	283
Refreshing	185	Processor menu	44
PE format support	258	Processor overrides	309
physical	553	processorcontrol	563
point.....	294, 517	processorfamily.....	565
pointer	294, 517	processormode	566
port	555	processors	567
pow	557	procesortype	568
Preferences	46	Program commands.....	23
Breakpoints Tab	48	Program Flash Tab	59
Code Tab	50	Program Tab	52
Colors Tab	54	Project commands	21
Emulator Tab	47	projectpath	569
General	46	putchar	570
Memory Tab	51	puts	571
Program Tab.....	52	PVT	55, 266
Print.....	32	Python/CScripts	264

Q

Qualified Symbol Names.....315

Quick watch.....241

R

rand572

readsetting.....573

real10294, 517

real4294, 517

real8294, 517

reconnect.....574

Register access.....575

Register lists - printing.....187

Registers187

Customizing window.....187

Keyword table.....268

Modify187

Opening187

reload577

reloadproject578

remove579

reset44, 581

restart583

return584

right585

runcontroltype586

S

safemode587

save589

Save Trace.....230

savebreakpoints.....590

savelayou.....591

savewatches592

selectdirectory.....593

selectfile594

shell.....595

show.....596

sin598

sizeof.....599

sleep600

SMM.....105

softbreak601

softdisable.....601

softenable601

softremove601

Source path.....20

SourcePoint15

Menu80

New features12

Refreshing windows97

Toolbar18

Customizing	41	swbreak.....	638
SourcePoint Licensing	271	switch	640
SourcePoint online help		Switches Tab	71
Help command	461	swremove.....	642
Menu	81	Symbolic References	310
sprintf.....	603	Symbolic text format	276
sqrt	604	Symbols window	193
srand	606	Adding/expanding registers in a Watch view	
step.....	607	246
stop.....	609	Changing values.....	204
strcat.....	610	Classes view	199
strchr	611	Globals view	200
strcmp.....	613	Locals view	202
strcpy.....	615	Menus.....	197
string.....	294, 517	Stack view	203
string [] (index into string).....	619	Syntax Notation.....	290
strlen.....	620	T	
strncat.....	622	tabs	643
strncmp.....	623	tan	644
strncpy.....	625	tapdatashift	645
strpos.....	626	tapstateset	647
strstr	628	Target Configuration	57
strtod	631	Loading.....	62
strtol.....	633	Overview	278
strtoul.....	635	Saving	63
		Target Devices Tab.....	60

Target Reset Tab	68	Emulator Tab.....	211
targpower	649	Intel PT Memory Tab.....	222
targstatus.....	650	Intel PT Tab.....	219
taskattach.....	652	LBR Tab	212
taskbreak.....	653	Trace Hub Tab	223
taskdisable	653	Trace Display Settings	227
taskenable	653	Trace Hub Metadata	233
taskend.....	655	Trace Hub Tab	223, 233
taskgetpid.....	656	U	
taskremove.....	653	uncoreconfigure	661
taskstart.....	657	uncorescan	662
tck.....	658	unload	664
Textsym.....	276	unloadproject	666
time.....	660	upload	667
Timestamp.....	231	use	669
toascii	349	V	
toint.....	349	verify	670
tolower.....	349	verifydeviceconfiguration	672
toupper	349	verifyjtagconfiguration	673
Trace	205	version	674
Opening	205	View menu commands	41
Printing.....	229, 558	viewpoint	675
Trace Configuration.....	211	Viewpoint Processor	309
AET Tab.....	215	Viewpoint Window.....	239
BTS Tab.....	214	Menu	240

vpalias676

W

wait678

Watch window241

 Adding symbols248

 Adding/expanding registers in246

 Menu244

wbinvd679

while680

windowrefresh682

wport683

writesetting685

Y

yield687

yieldflag688