



UEFI & EDK II TRAINING

EDK II Debugging w/ Intel® System Studio

tianocore.org

LESSON OBJECTIVE

* Intel® System Studio Debugger Overview

Identify the Intel® System Studio Debugger host and target

* Getting started with Intel® System Studio Debugger

Using IPCCLI

TCF GUI

XDB GUI

* Configure when target has UEFI debug agent

* Using the INT1 ICEBP instruction as a debug break point

SYSTEM STUDIO DEBUGGER

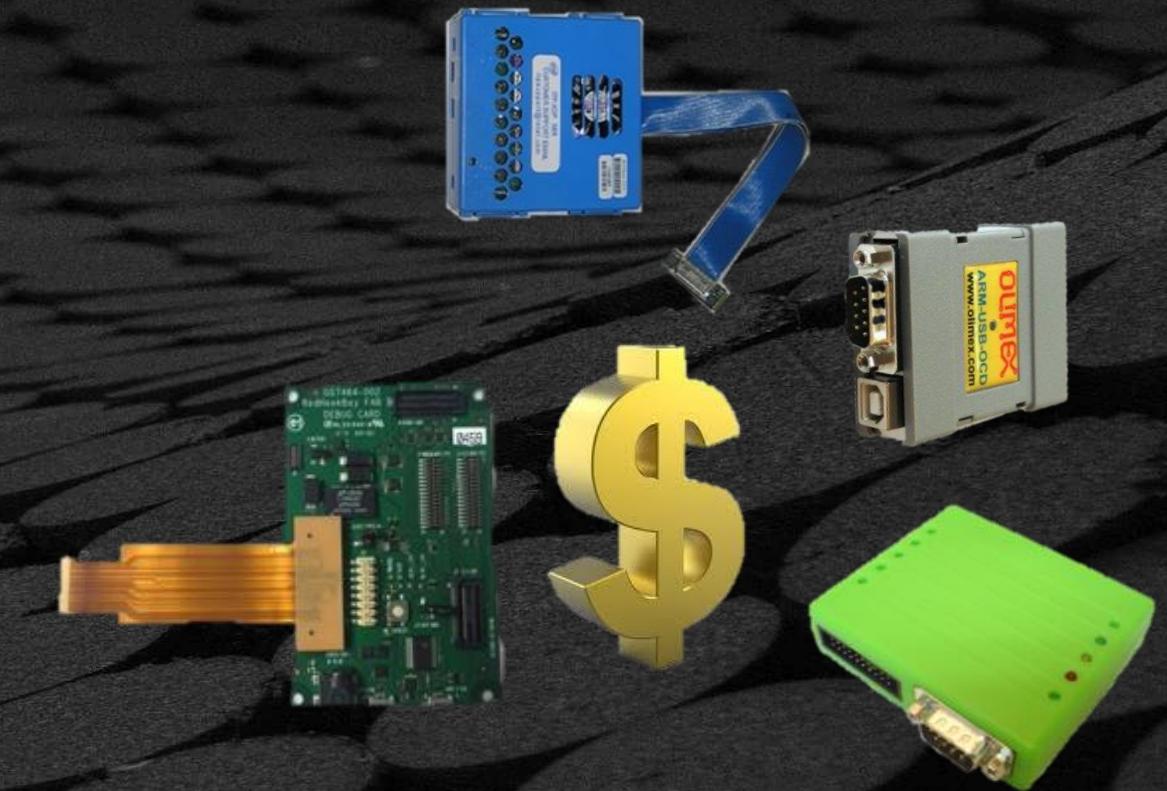
Intel® System Studio Debugger Overview

Debugging Closed Chassis Product



No debug connector

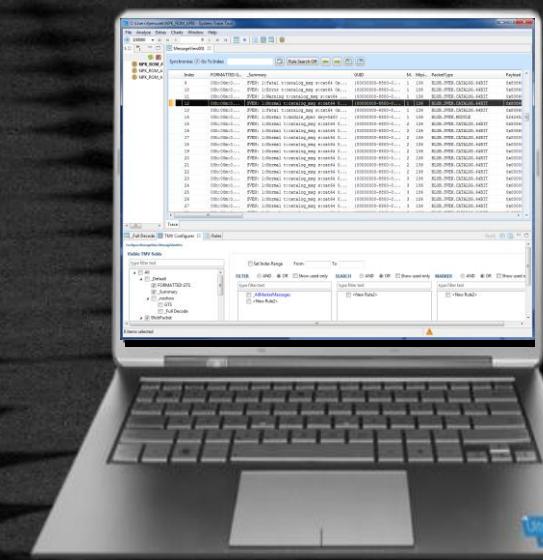
Target: closed chassis



Debug equipment

Debug Interface - DCI

Host

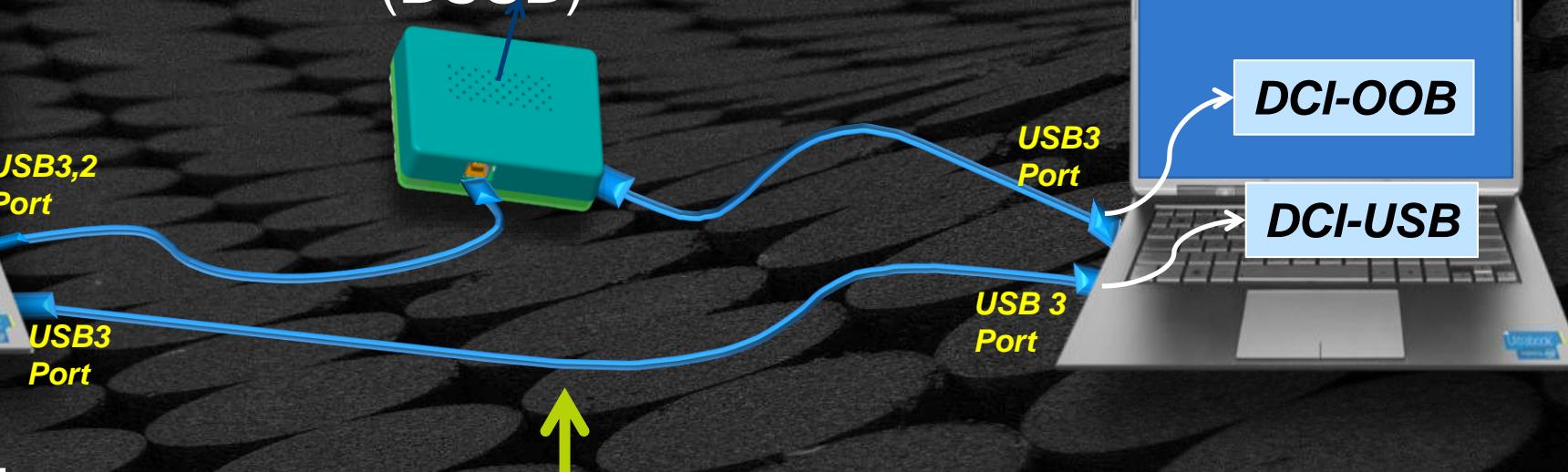


Software required:
Intel® System Debugger

DCI tool-transport technology that allows connecting to a closed-chassis target system using USB 3.0.

Option(1)
Hardware required:
Intel® SVT Closed
Chassis Adapter
(BSSB)

Target (Closed chassis)
6th generation



Option(2)
USB3 Debug cable type A-to-A cable

System-wide Closed Chassis Debugging

JTAG*-based system debug and system trace over low-cost USB* connections

- Eliminate hours of developer time working on actual production hardware instead of expensive engineering samples
- Save hundreds of dollars on JTAG equipment & probes - debug over a standard USB* connection instead of expensive JTAG probes
- Developers no longer need to wait for a turn at a shared JTAG debugging station
- Design flexibility alleviates the requirement for an accessible hardware JTAG port

USB CABLE



Debug & Trace OS boot

INTEL® SVT CLOSED CHASSIS ADAPTER¹



Debug & Trace from CPU reset

(1) SVT = Silicon View Technology – more details: https://designintools.intel.com/product_p/itpxdpsvt.htm

Intel® SVT CCA And USB3.0 Debug Cable



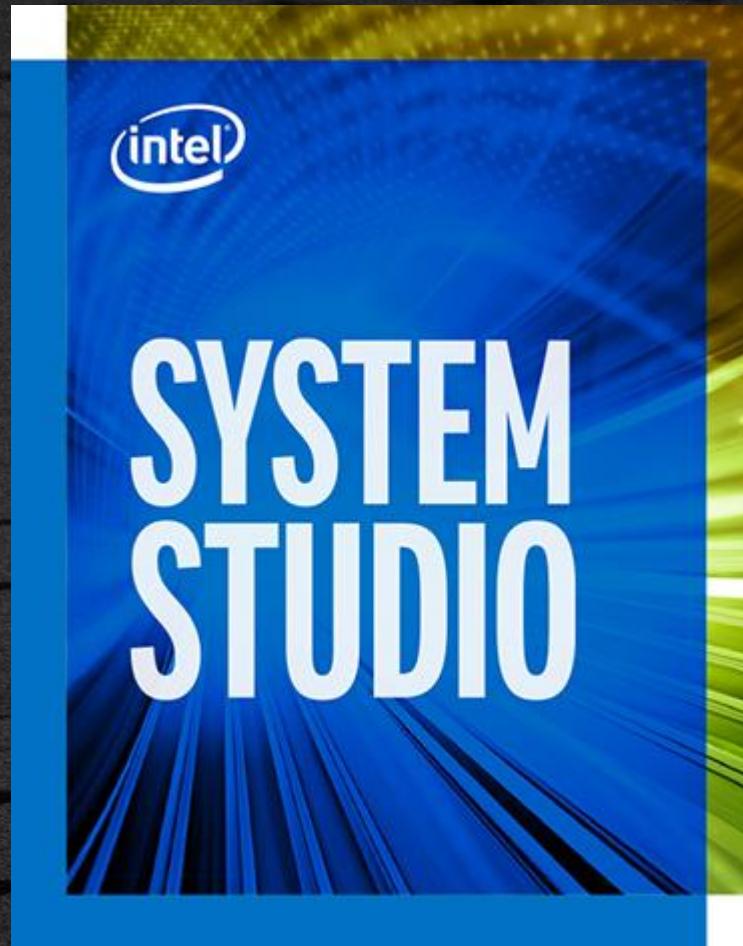
Intel® SVT Closed Chassis Adapter (CCA) can be purchased through Intel® Design-In Tools Store at
https://designtools.intel.com/product_p/itpxdpsvt.htm



Intel® SVT DCI DbC2/3 A-to-A Debug Cable :
https://designtools.intel.com/SVT_DCI_DbC2_3_A_to_A_Debug_Cable_1_Meter_p/itpdciama1m.htm

Download Intel® System Studio

GET STARTED NOW



Free 90-Day Renewable Community License

Use Intel® System Studio with a free community license backed by community forum support. This license allows you to use the software for 90 days. You can refresh the license an unlimited number of times, allowing you to use the latest version. Convert to a paid license at any time, which provides Priority Support for one year.

Supported host operating systems:

Windows*

Linux*

macOS*

Register & Download

<https://software.intel.com/en-us/system-studio/choose-download>

Intel® System Debugger Overview

The screenshot shows the Intel System Debugger's graphical user interface. On the left, there is a large window displaying assembly code for a file named 'DeleteFile.c'. The code includes comments like 'Notify deletion, if this is an alternate Data Stream be' and 'check if the whole file was deleted by calling DfIsFileDeleted()'. Below the assembly code is a memory dump window showing hex values and ASCII representation. At the bottom, there are tabs for 'Registers', 'Locals', and 'Disassembly'.



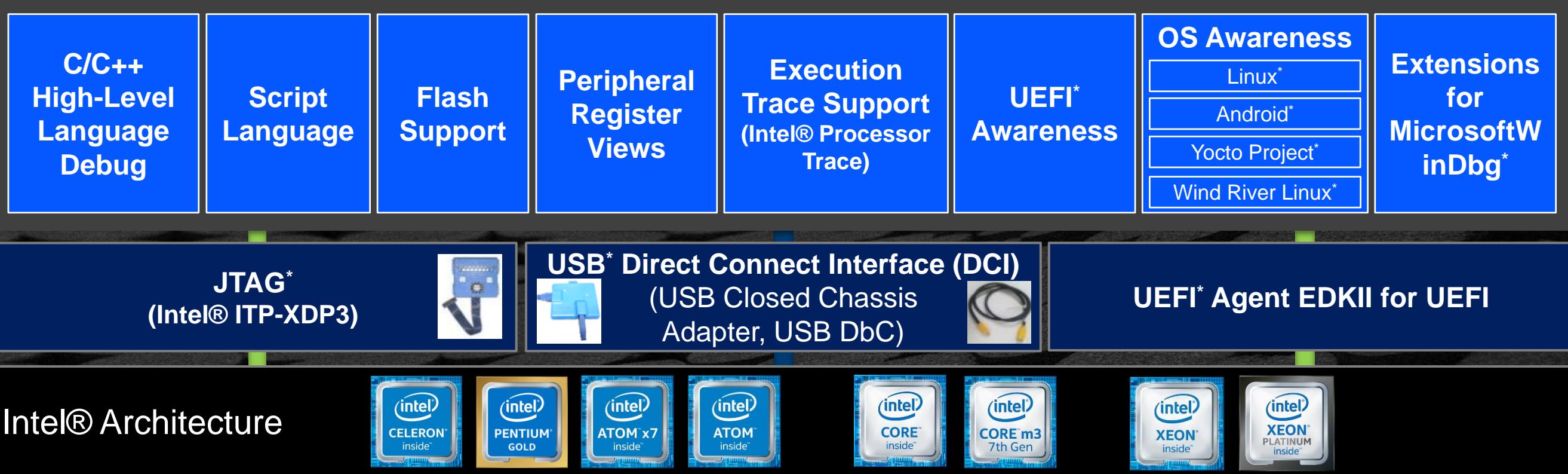
¹ System Trace is only Supporting 6th generation core

Key Features

- Linux* and Windows* host
- JTAG debug for Intel® Atom, Core™, Xeon® & Quark SoC-based platforms
- EFI/UEFI Firmware, bootloader debug, Linux* OS awareness and Kernel module debug
- Full CPU register description and bitfield editor
- Access to page translation and descriptor tables
- LBR, IPT On-Chip instruction trace support, SMP run control support
- JTAG debug & instruction trace to Microsoft* WinDbg* kernel debugger *NEW*
- System Trace: System-wide hardware and software event trace *NEW*

Intel® System Debugger

Debug & Trace Intel® Architecture-based System Software & IoT applications



- Speed up debug and validation with deep hardware and software insight for faster time-to-market
- Ideal for platform bring-up and debug of firmware, UEFI/BIOS, operating systems, and device drivers
- Debug Windows* and Linux* kernel sources, and dynamically loaded drivers & kernel modules

General UEFI Features:

- Source-level debug in any phase of UEFI, from reset to OS boot
- Load symbols for all or selected modules
- Simultaneous debug of MSVC and GCC-built modules

Passive mode (interrogation-based):

- Inspect target memory to locate modules, load symbols
- Can be used on production BIOS
- Requires JTAG or DCI cable

Active Mode (agent-based):

- Receive notifications from agent as modules are loaded/unloaded
- Break at init of a named module, regardless of load position
- Requires debug agent “SourceLevelDebugPkg” in EDKII
- Only use on Debug version of FW

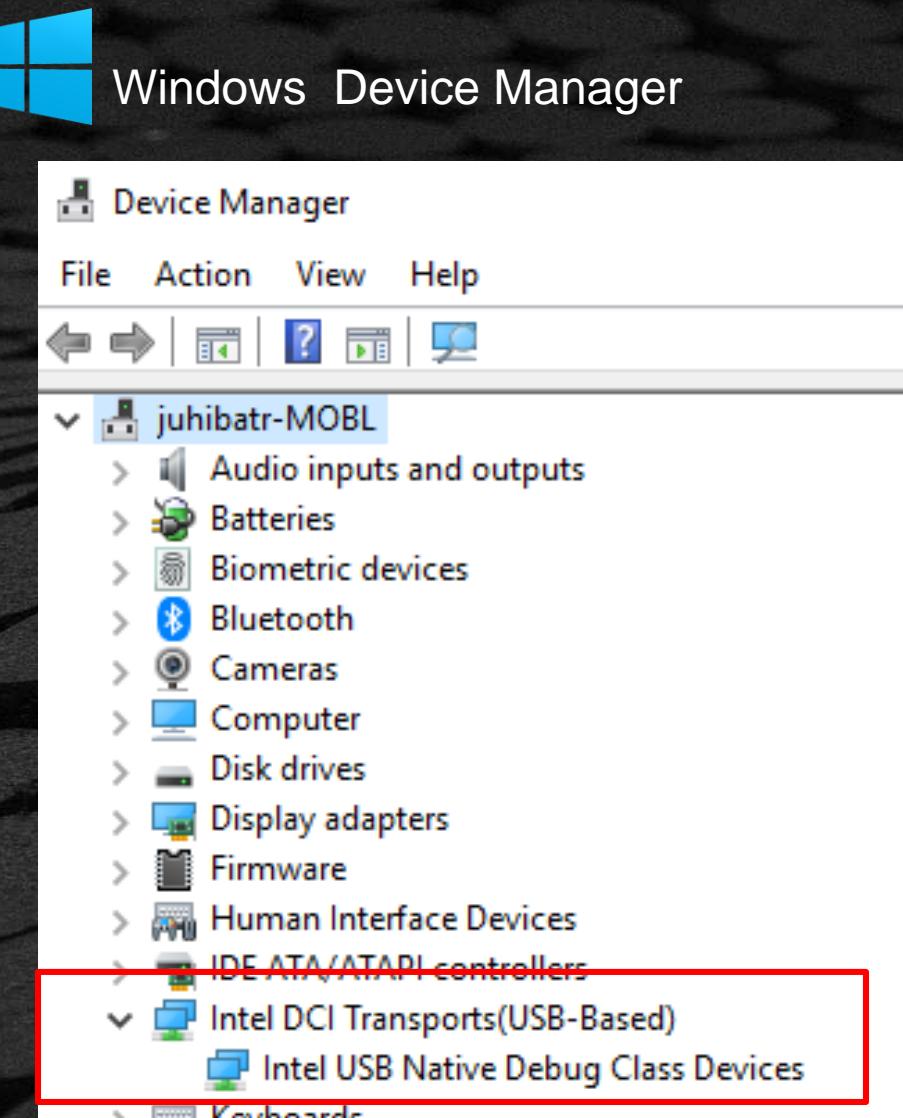
GETTING STARTED

Getting started with Intel® System Studio Debugger

Power up the Target

Power Up the target, make sure the Intel® Debug Cable - DCI probe is connected the target and host

Check for the connection by opening the Device Manager and looking for Intel DCI Transports (USB based) – Intel USB Native Debug Class Devices as shown in the picture.



Intel® System Studio Debugger- IPCCLI

Ipccli - Tool that provides a Python command line interface to use OpenIPC

First edit the config file **OpenIpcConfig.xml**

Go to folder `c:\IntelSWTools\system_studio_2019\tools\OpenIPC_1.1915.3701.200\Config` and open file **OpenIpcConfig.xml**

Edit it to reflect `<DefaultIpcConfig Name="BXTP_DCI_OpenRC" />` for UP Squared board.

Open command line prompt as administrator and setup the environment using the `iss_env.bat` command

```
$> cd C:\IntelSWTools\system_studio_2019  
$> iss_env.bat
```

Execute `ipccli` (make sure python and python/scripts are in the path)

```
$> ipccli
```

Intel® System Studio Debugger- IPCCLI

- Halt the target using `itp.halt()`
- Set reset break using following command `itp.cv.resetbreak`
- Set `itp.cv.resetbreak=1` if it is set to 0
- Then use this command to for reset `itp.threads[0].port(0xcf9, 0x6)`
- Check the status of the target threads and halt them if not already halted using `itp.status()` and `itp.halt()` resp.
- Enable INT1(ICEBP) handler over ISD: `itp.breaks.int1exception = 1`
- Use `itp.go()` to resume processors to running
- Use `itp.help()` for help manual

Setting IO breakpoint from IPCCLI

Open Ipccli Command window and use instruction `itp.brget()` for any breakpoints

Use command `itp.brnew("0x80", "io")` to create a IO breakpoint on Port 80 Hex

Go back to the ISD GUI and resume the target

The target will halt on hitting the IO breakpoint

Once the Breakpoint is hit , click on Loadthis in the GUI to load the source files which will be stopped at Postcode function call.

Use the command `itp.brdisable(#)` where “#” is the breakpoint to disable

Intel® System Studio Debugger

- 2 GUI Versions

TCF based
debugger

- Target Communication Framework(TCF) works with the Eclipse IDE
- TCF provides a complete modern debugger for C/C++

XDB Based
debugger

- Intel ITP based debugger supporting JTAG
- Moving to Legacy debugger by Intel® System Studio

The underlying debugging is similar but the user interface is different

GETTING STARTED

1. Download latest Intel® System Studio :

<http://intel.ly/system-studio>

2. Run installer until installer completes installation

TCF - Launch script for Intel® System Studio

-  : C:\Program Files (x86)\IntelSWTools\system_studio_2019\iss_ide_eclipse-launcher.bat
-  : opt/intel/system_studio_2019/IntelSWTools/system_studio_2019/iss_ide_eclipse-launcher.sh

XDB - Launch script for Intel® System Studio Debugger – Legacy

-  : C:\Program Files (x86)\IntelSWTools\system_studio_2019\System Debugger 2019\system_debug\xdb.bat
-  : opt/intel/system_studio_2019/system_debugger_2019/system_debug/xdb.sh

Intel® System Studio Debugger

- 2 GUI versions

TCF based debugger

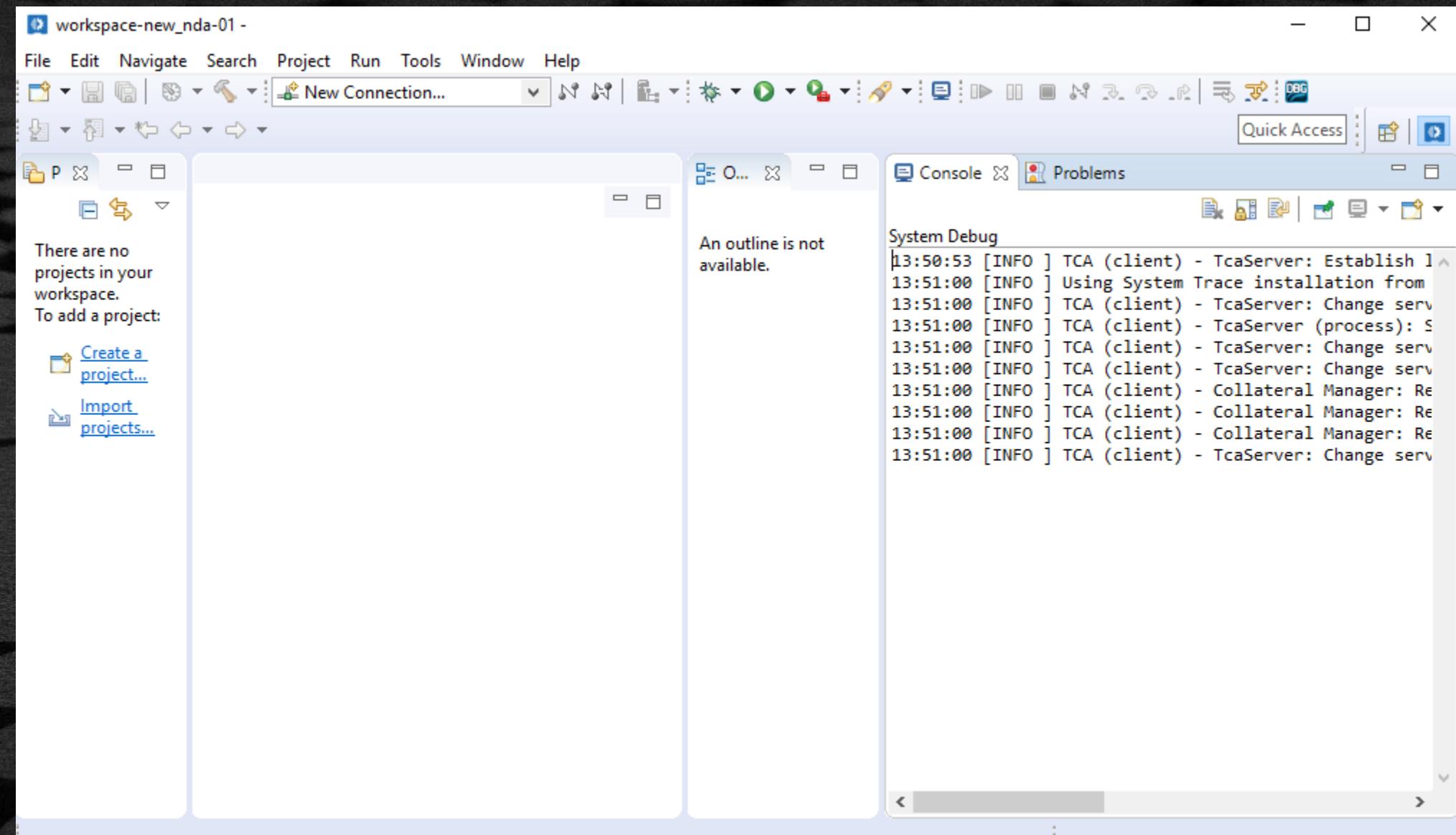
- Target Communication Framework(TCF) works with the Eclipse IDE
- TCF provides a complete modern debugger for C/C++

XDB Based debugger

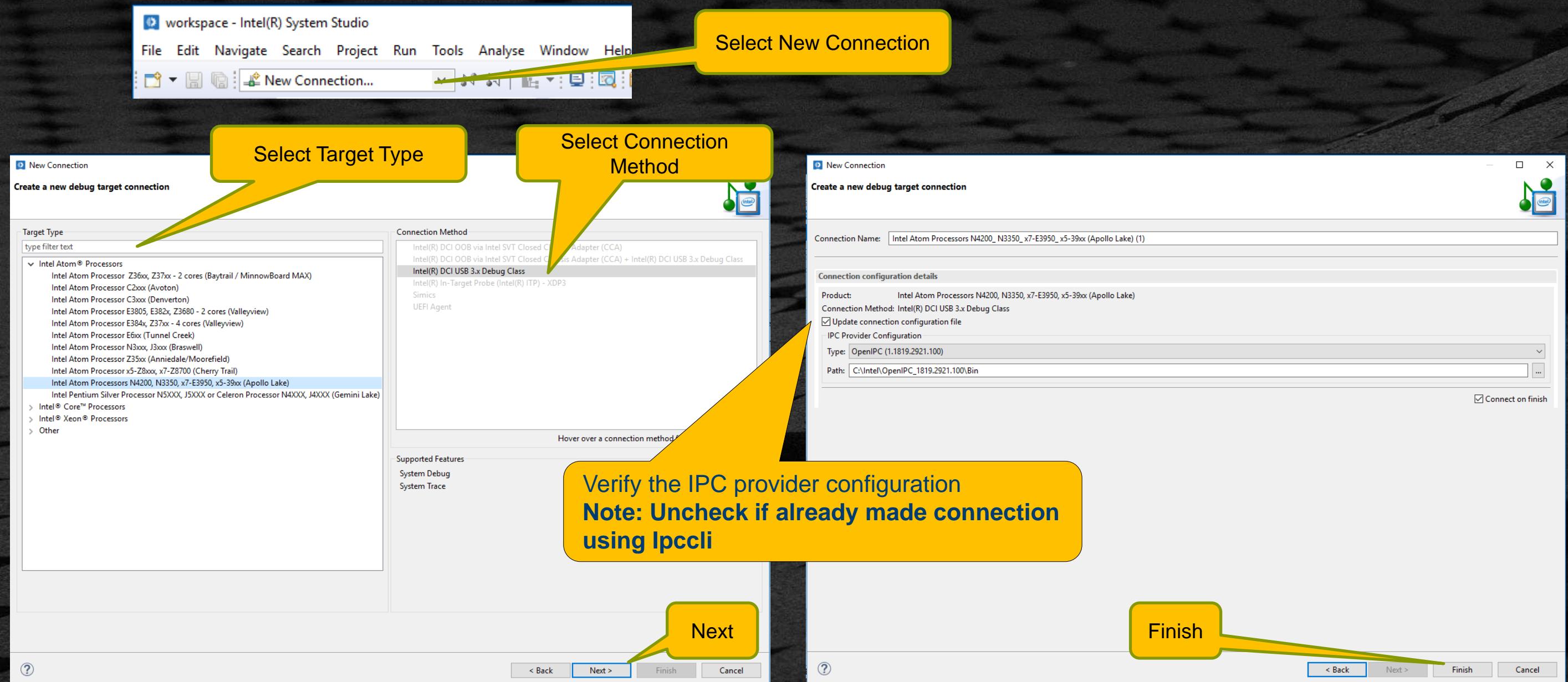
- Intel ITP based debugger supporting JTAG
- Moving to Legacy debugger by Intel® System Studio

The underlying debugging is similar but the user interface is different

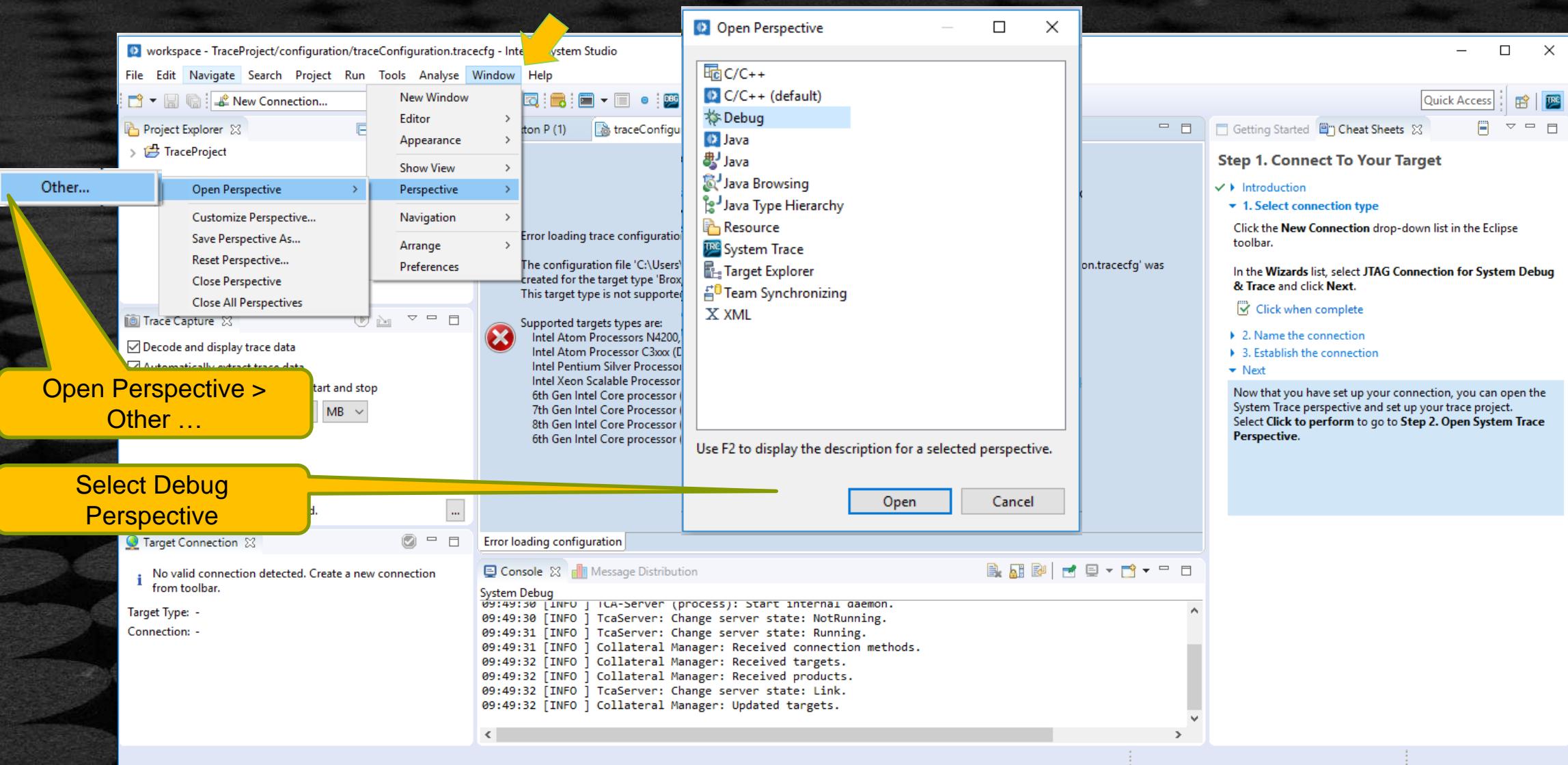
Launch Eclipse IDE



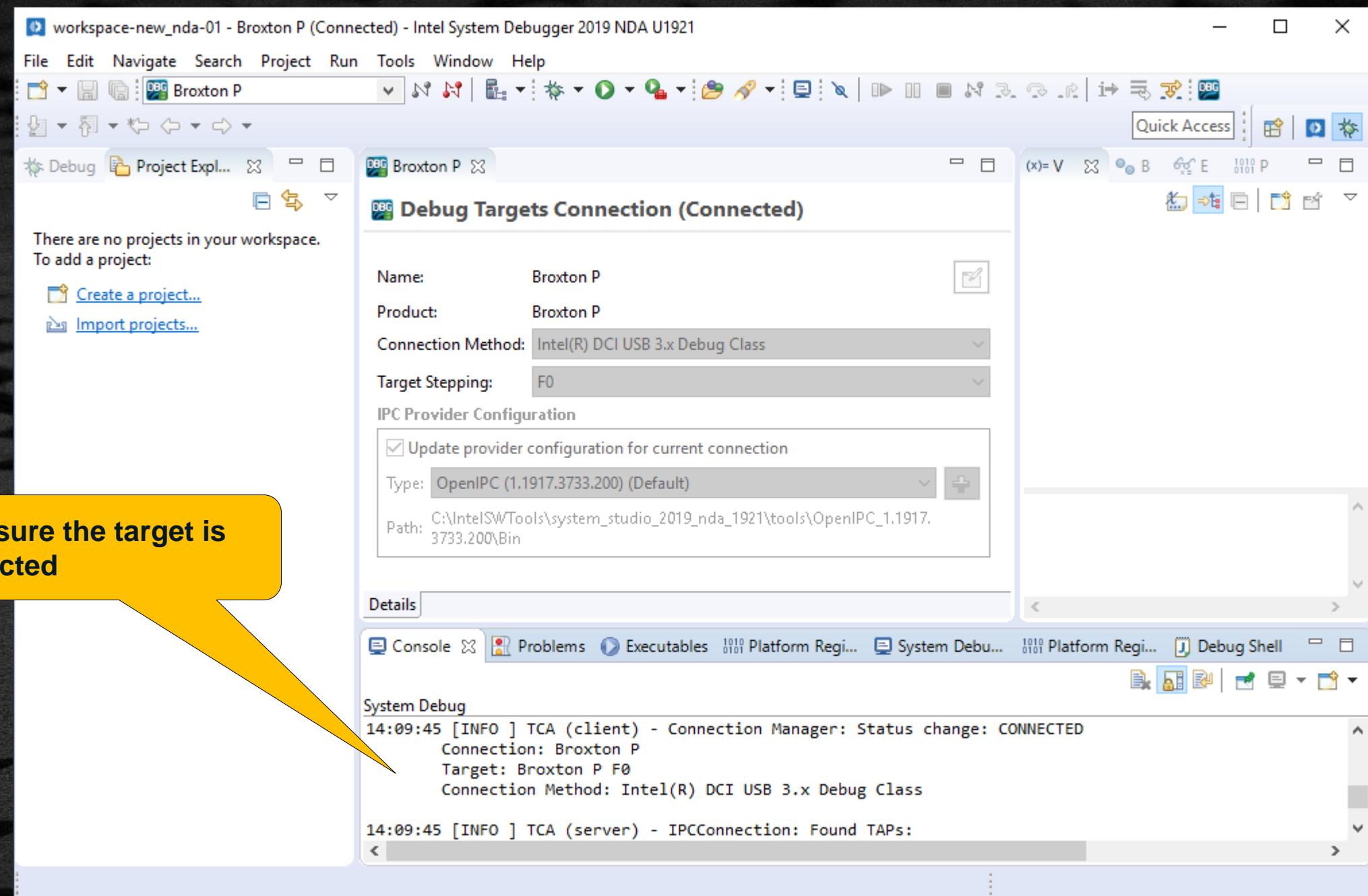
Create a New Debug Target Connection



Open Perspective Debug

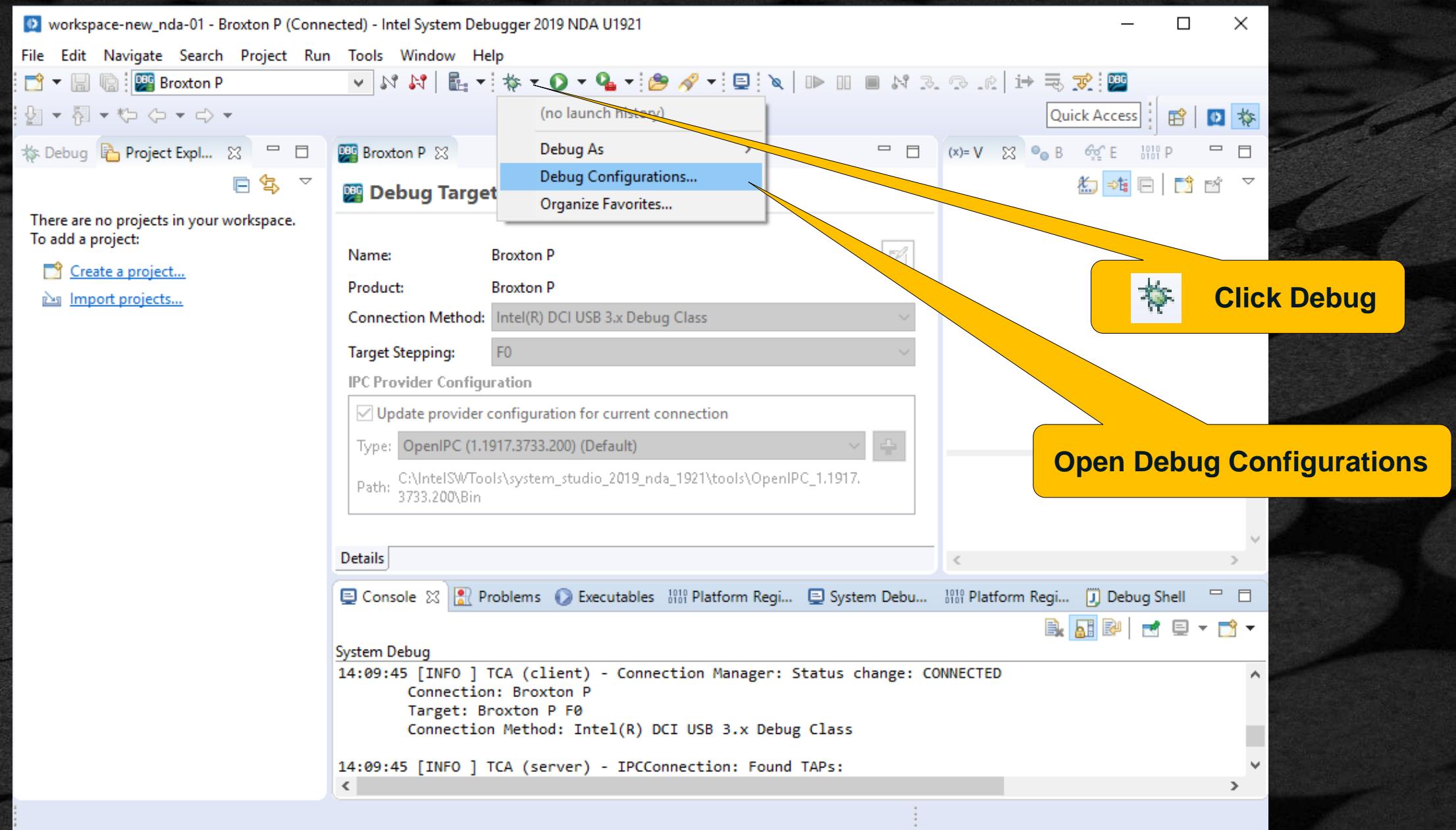


Debug Perspective

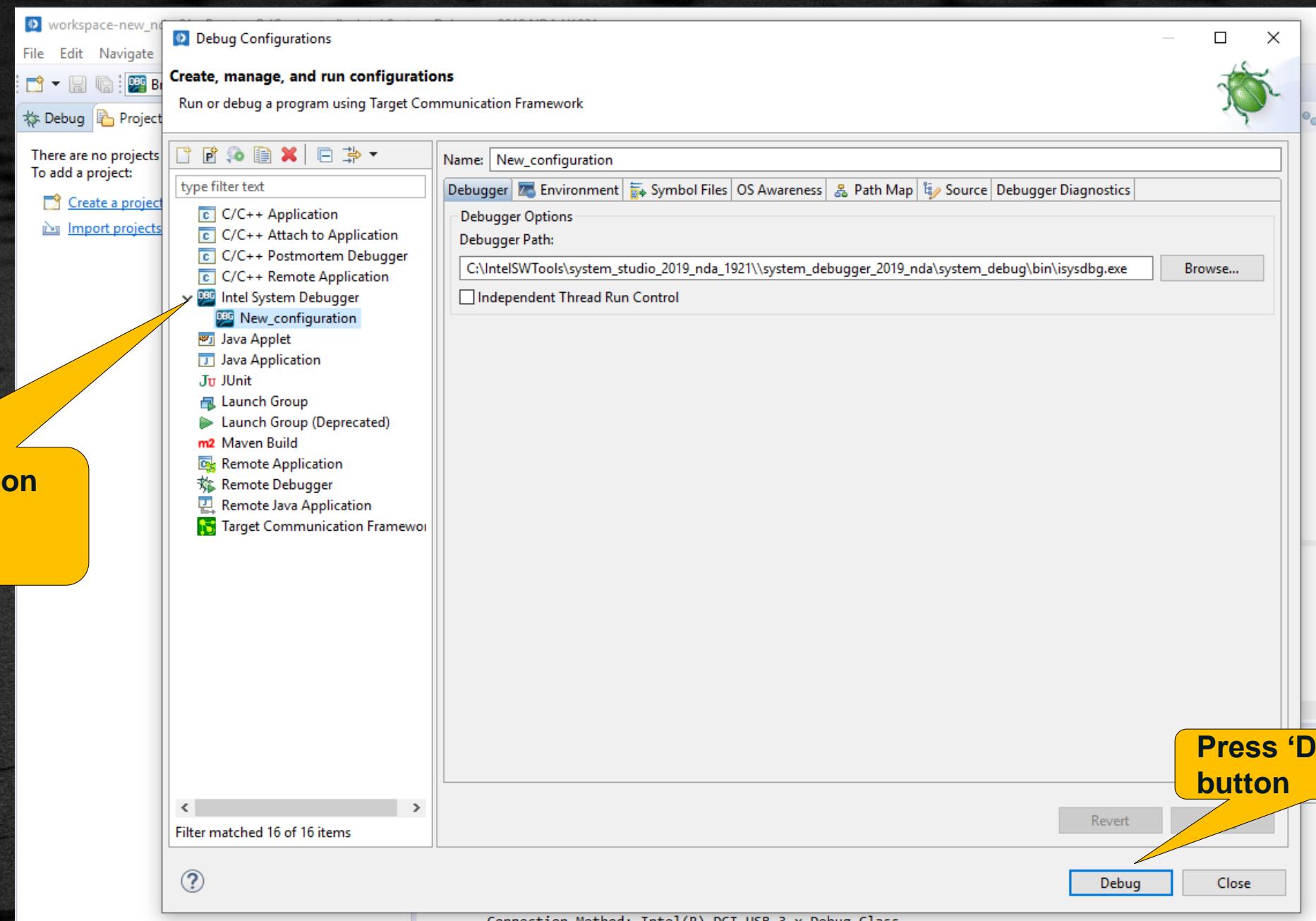


Launch System Debug - TCF

Select Debug Configurations

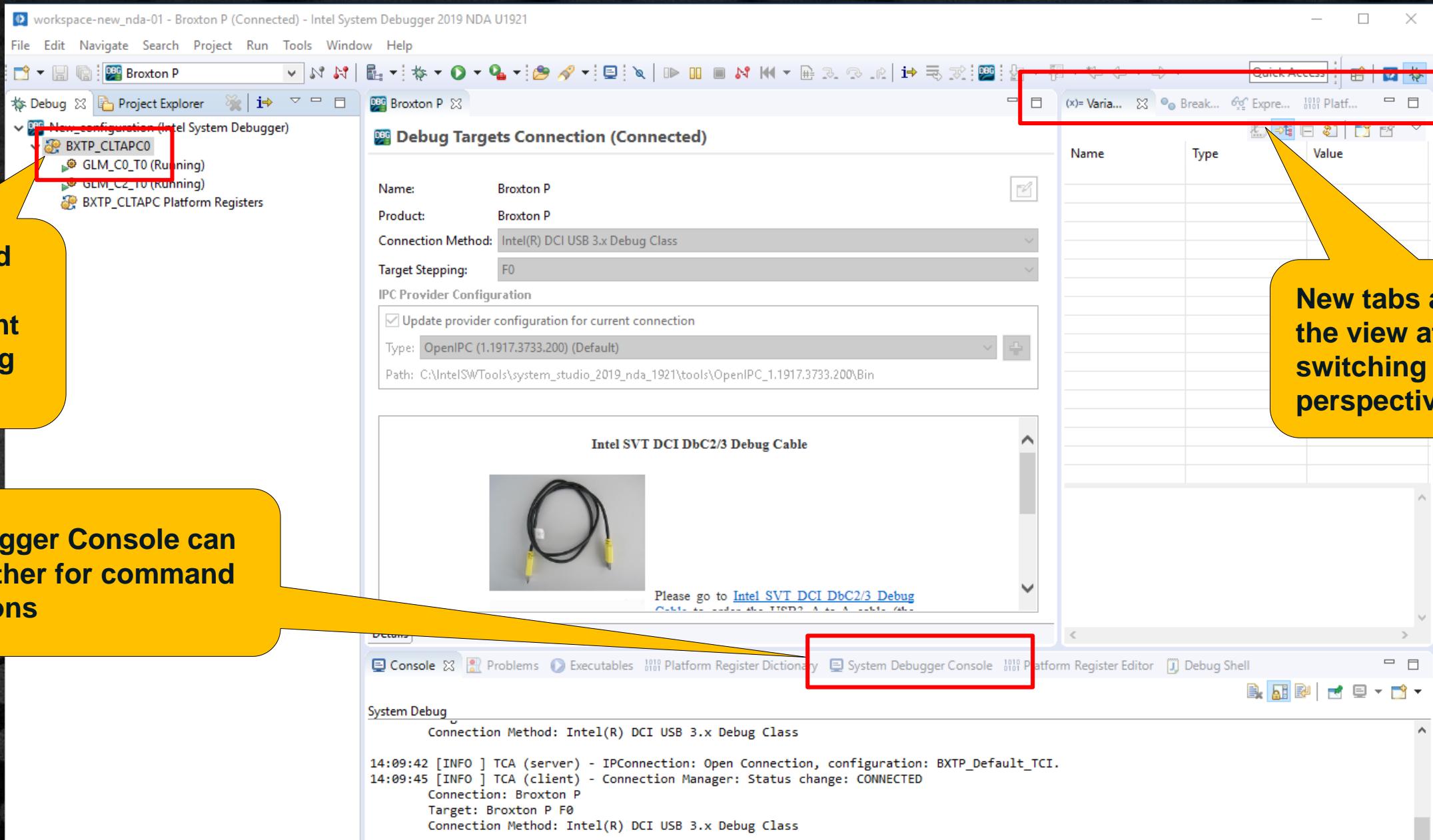


Launch System Debug - TCF



Debug Perspective - TCF

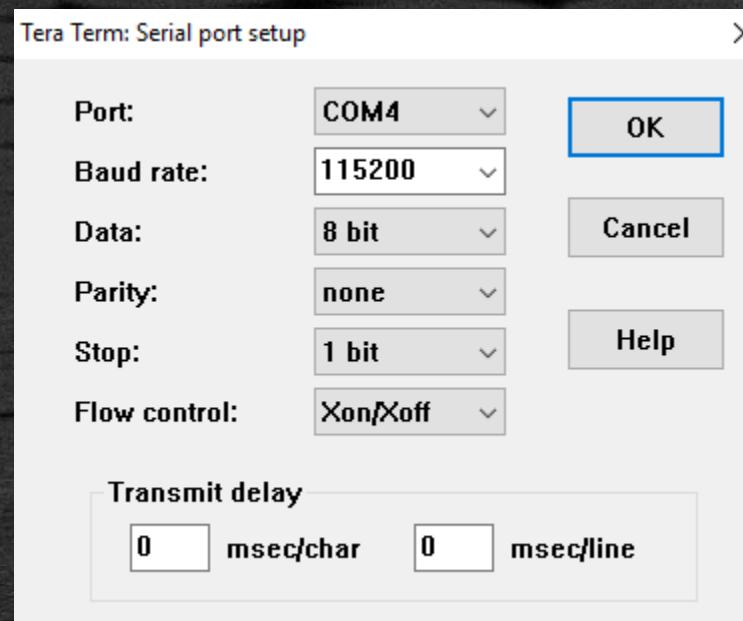
Now Debugging Target



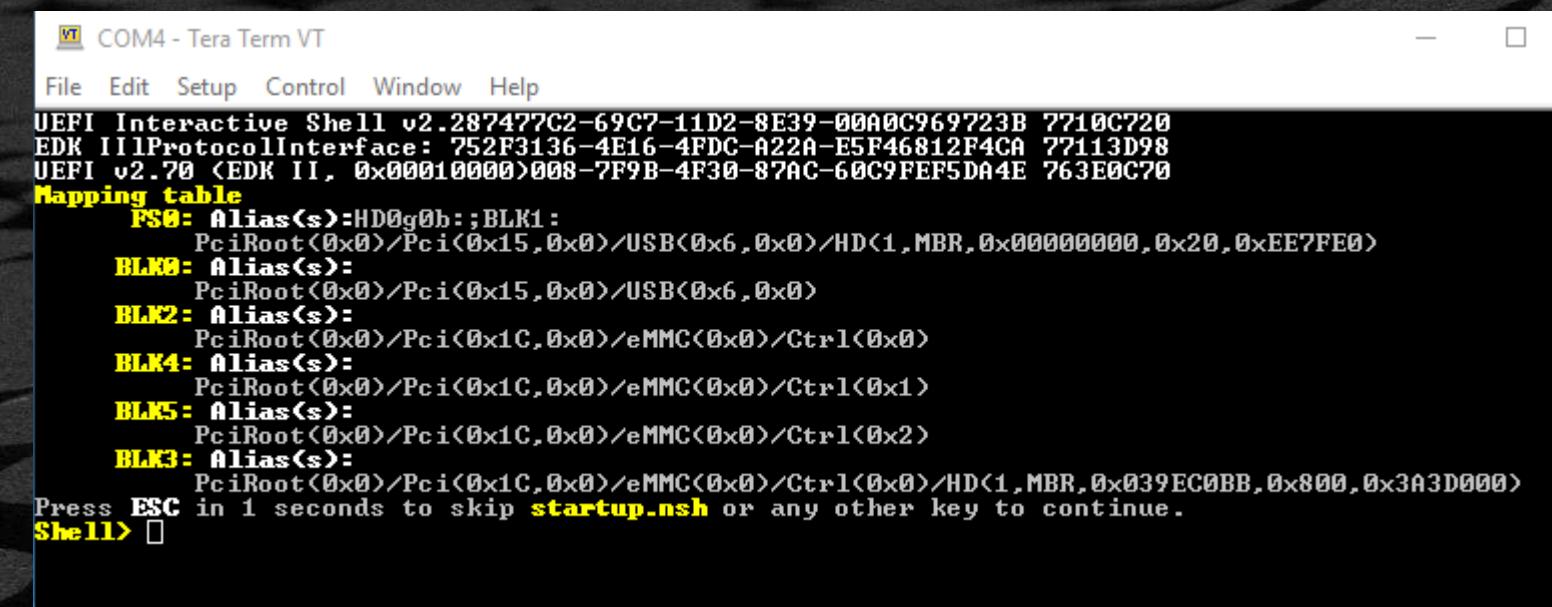
Serial Debug Output Example

Tera Term used as example:

Shows target currently at the UEFI Shell Prompt.



Serial port configuration on Host

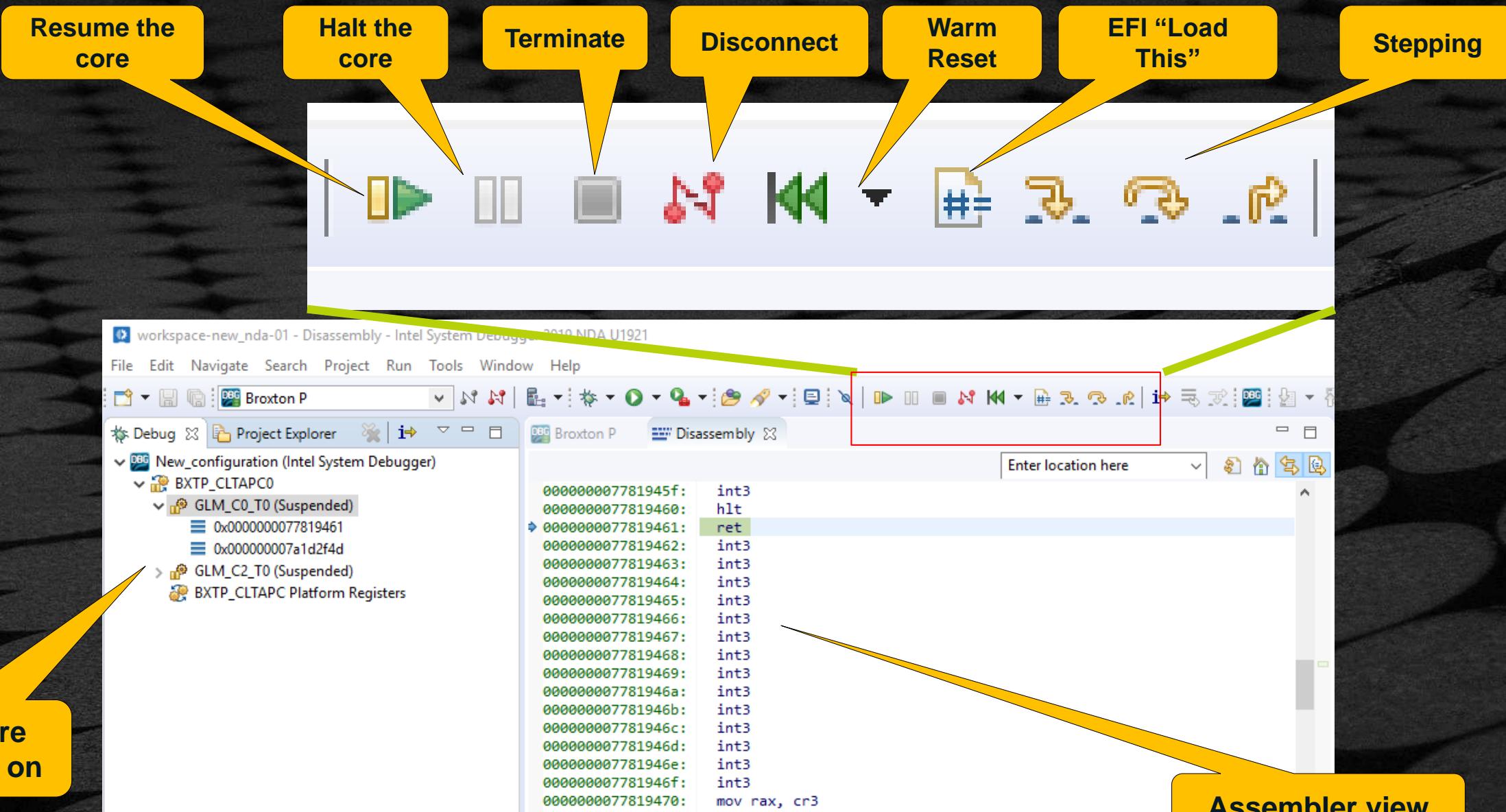


The screenshot shows the Tera Term window titled 'COM4 - Tera Term VT'. The window displays the UEFI Interactive Shell prompt. The output includes:

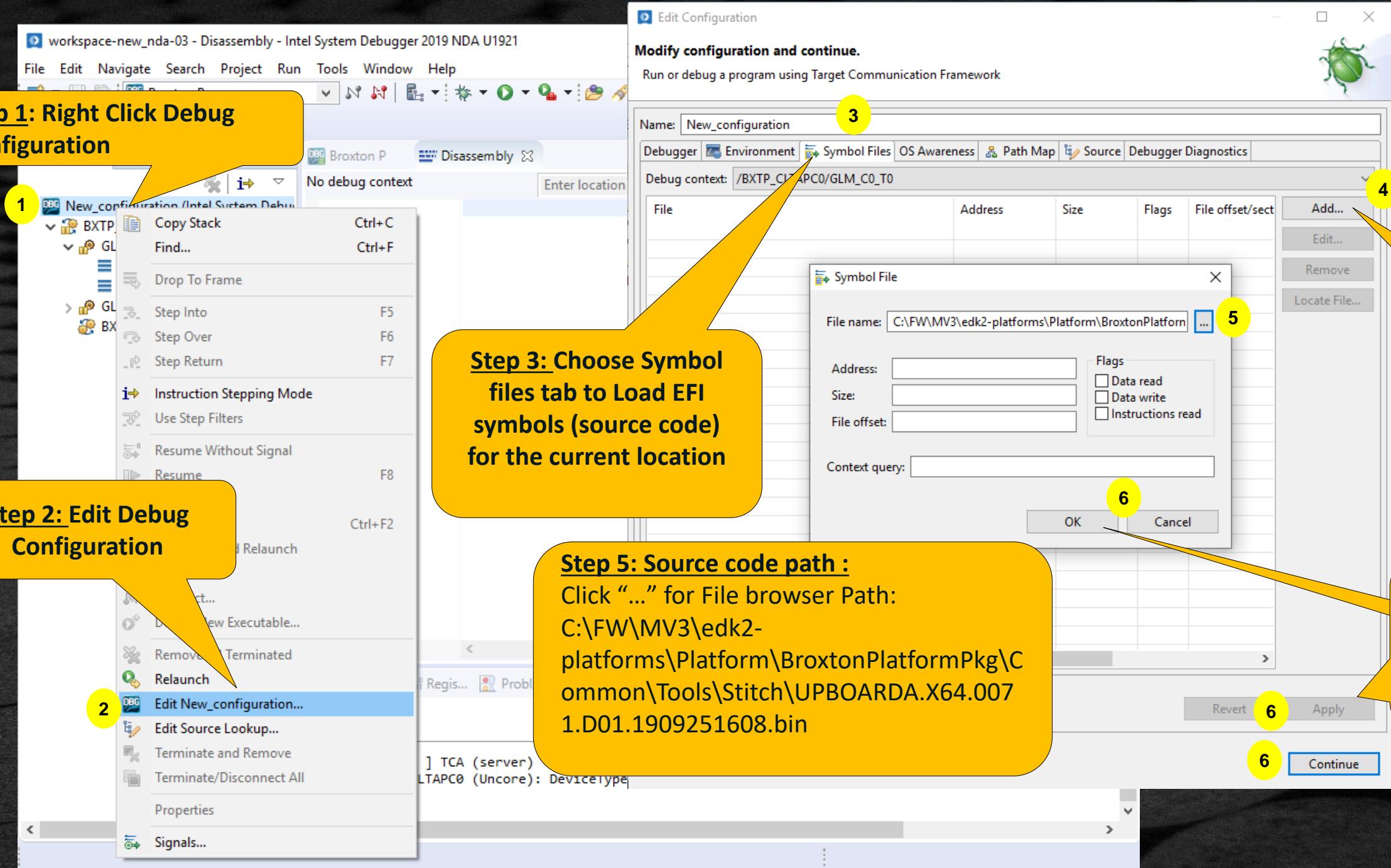
```
UEFI Interactive Shell v2.287477C2-69C7-11D2-8E39-00A0C969723B 7710C720
EDK II!ProtocolInterface: 752F3136-4E16-4FDC-A22A-E5F46812F4CA 77113D98
UEFI v2.70 (EDK II, 0x00010000)008-7F9B-4F30-87AC-60C9FEF5DA4E 763E0C70
Mapping table
  FS0: Alias<s>:HD0g0b::BLK1:
    PciRoot<0x0>/Pci<0x15,0x0>/USB<0x6,0x0>/HD<1,MBR,0x00000000,0x20,0xEE?FE0>
  BLK0: Alias<s>:
    PciRoot<0x0>/Pci<0x15,0x0>/USB<0x6,0x0>
  BLK2: Alias<s>:
    PciRoot<0x0>/Pci<0x1C,0x0>/eMMC<0x0>/Ctrl<0x0>
  BLK4: Alias<s>:
    PciRoot<0x0>/Pci<0x1C,0x0>/eMMC<0x0>/Ctrl<0x1>
  BLK5: Alias<s>:
    PciRoot<0x0>/Pci<0x1C,0x0>/eMMC<0x0>/Ctrl<0x2>
  BLK3: Alias<s>:
    PciRoot<0x0>/Pci<0x1C,0x0>/eMMC<0x0>/Ctrl<0x0>/HD<1,MBR,0x039EC0BB,0x800,0x3A3D000>
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Shell>
```

Basic Debug Buttons - TCF

Core status and Disassembly

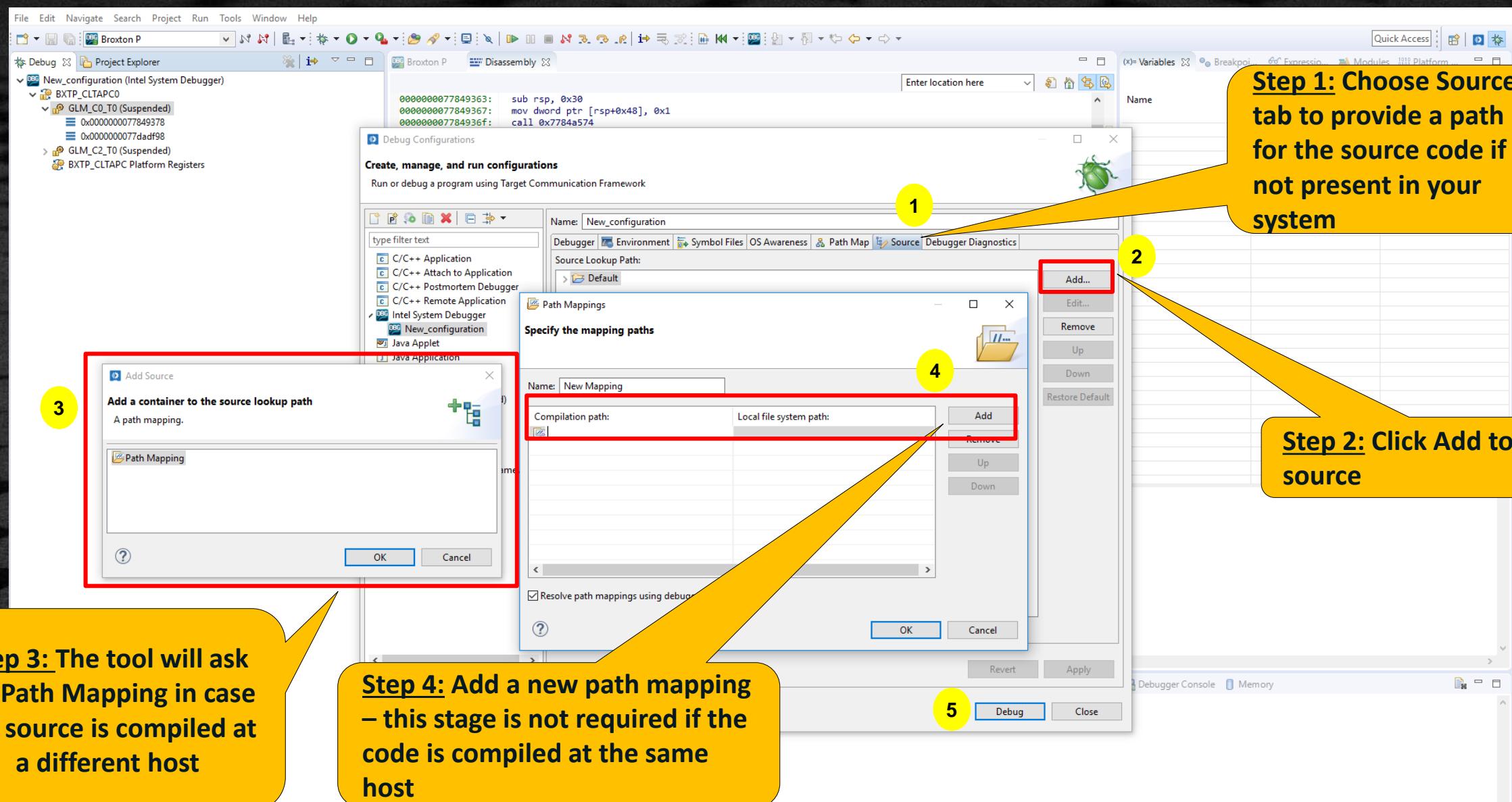


Load source codes



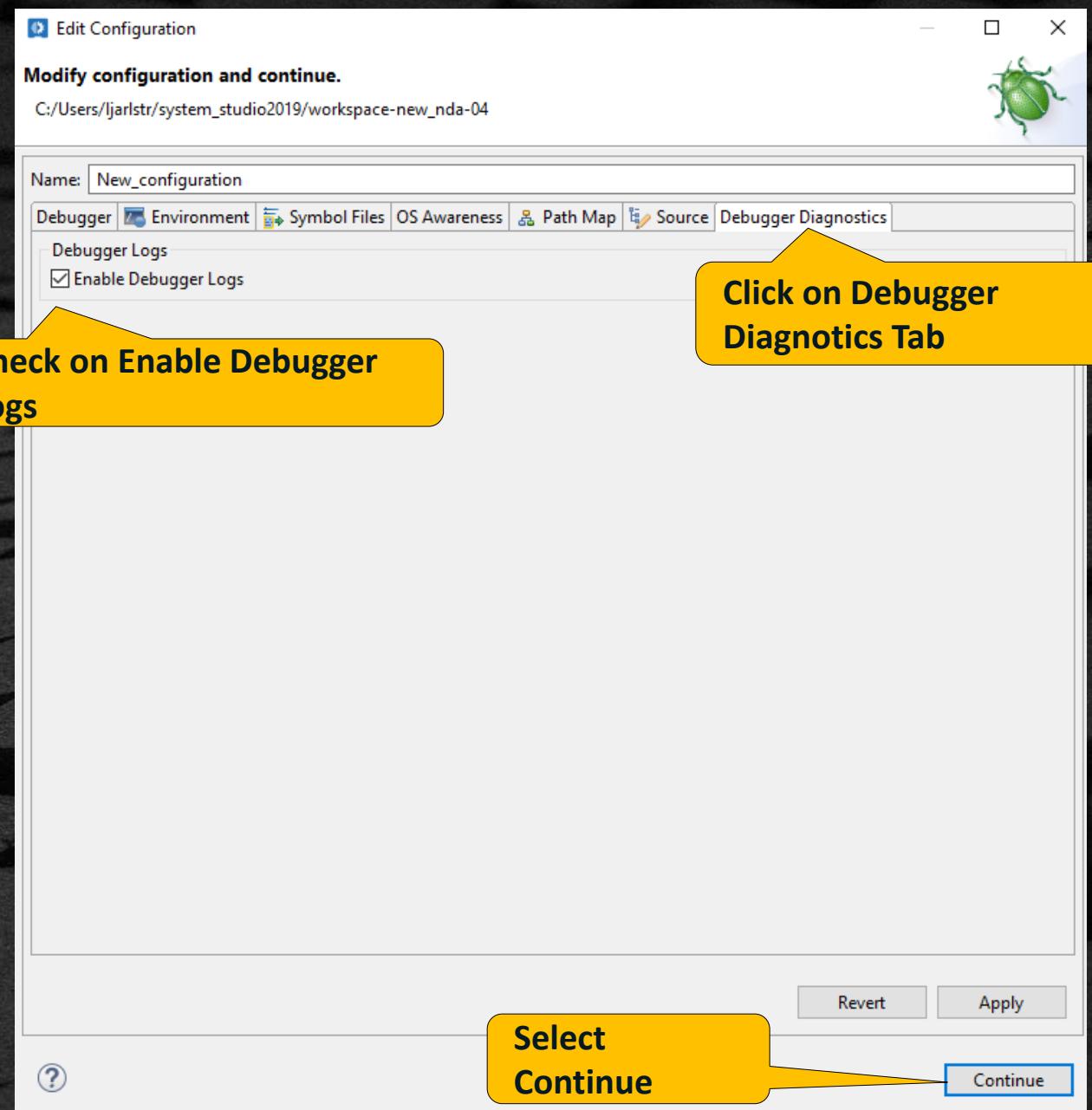
Load Source Code – Not Built on Host

Source code compiled on a different host system

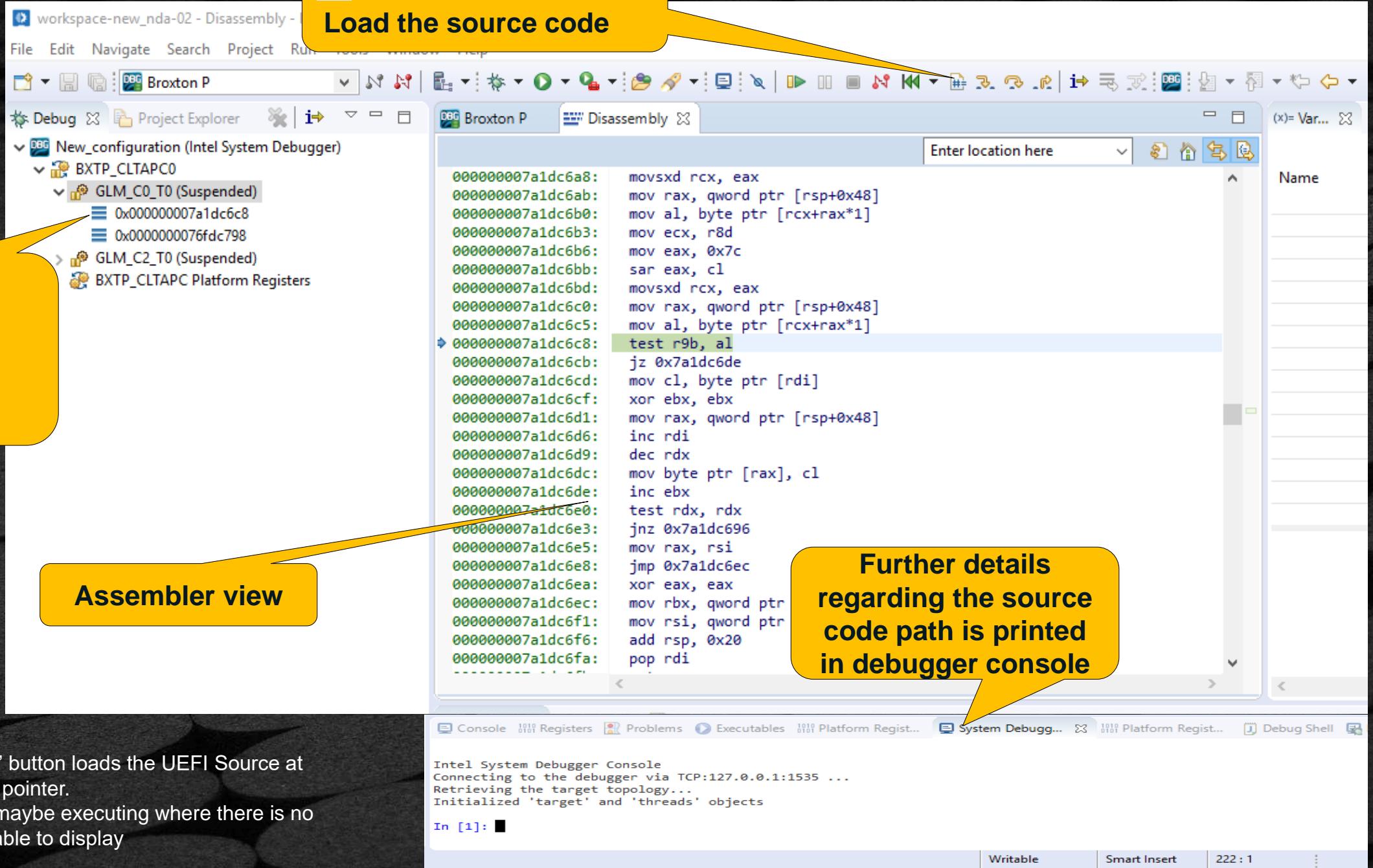


Enabling Debugger Diagnostic

- Enabling Debugger logs are necessary in case you face a problem while connecting to the target. Will help the engineering team to debug that issue
- Debugger logs can be enabled in the "Debugger Diagnostics" tab in the launch configuration. The debugger log files (isysdbg-<timestamp>.log) will be created in the Eclipse workspace folder.



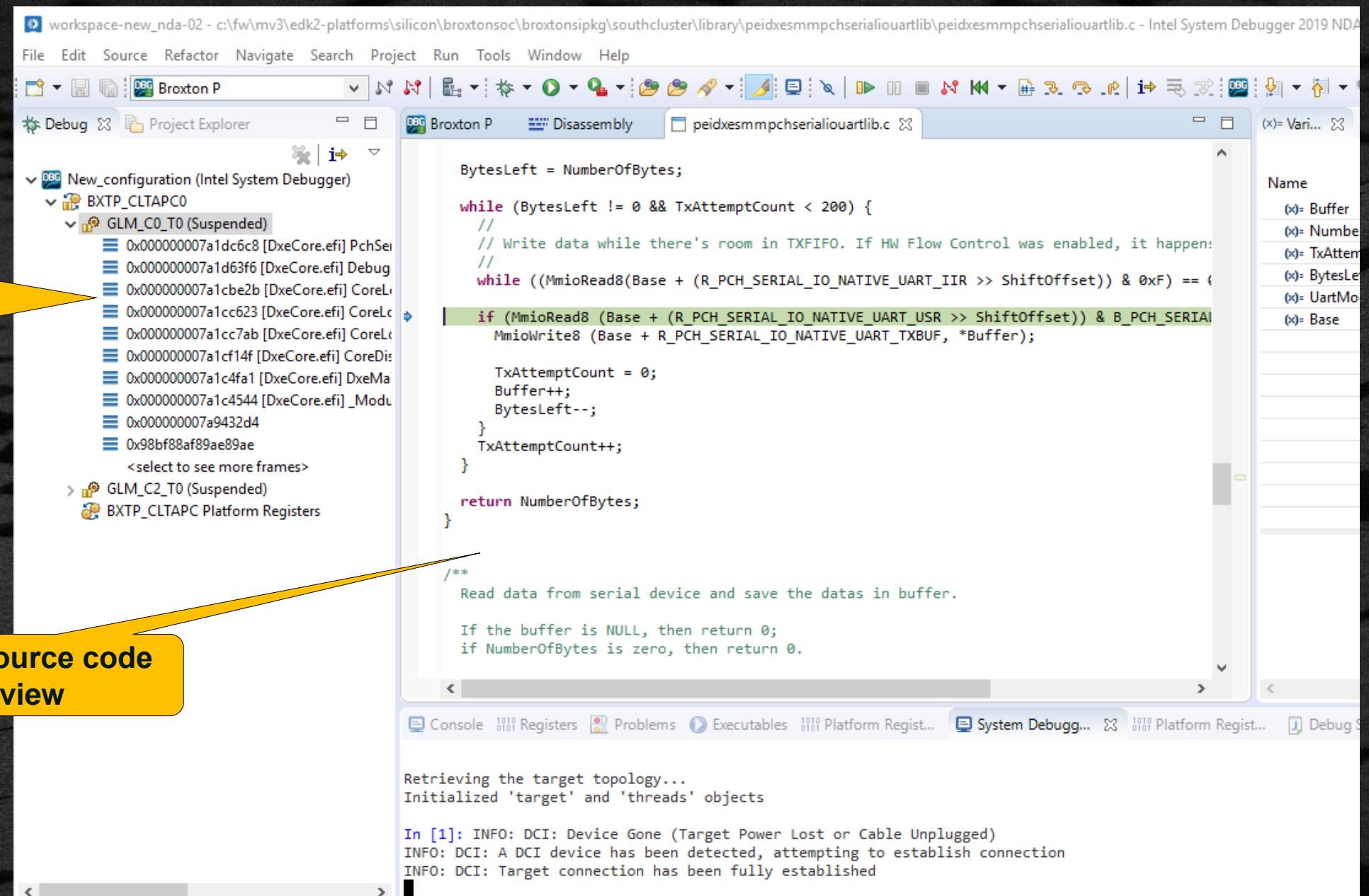
“Load This” Button



Note: The “Load This” button loads the UEFI Source at the current Instruction pointer.
 Some UEFI Modules maybe executing where there is no “C” source code available to display

Load the Source for UEFI

After “Load This” Button – notice the stack now has information on “DxeCore.efi”



Stack information Now shows UEFI Modules

“C” Source code view

The screenshot shows the Intel System Debugger interface. The Project Explorer view displays a tree structure of UEFI modules, including 'New_configuration (Intel System Debugger)', 'BXTP_CLTAPC0', and 'GLM_C0_T0 (Suspended)'. The 'GLM_C0_T0' node is expanded, showing numerous memory frames, some of which are labeled as being from 'DxeCore.efi'. A yellow arrow points from the text 'Stack information Now shows UEFI Modules' to this expanded tree. Another yellow arrow points from the text '“C” Source code view' to the main code editor area.

```

BytesLeft = NumberOfBytes;

while (BytesLeft != 0 && TxAttemptCount < 200) {
    //
    // Write data while there's room in TXFIFO. If HW Flow Control was enabled, it happens here.
    //
    while ((MmioRead8(Base + (R_PCH_SERIAL_IO_NATIVE_UART_IIR >> ShiftOffset)) & 0xF) == 0)
        if (MmioRead8 (Base + (R_PCH_SERIAL_IO_NATIVE_UART_USR >> ShiftOffset)) & B_PCH_SERIAL_IIR_RXFIFO_EMPTY)
            MmioWrite8 (Base + R_PCH_SERIAL_IO_NATIVE_UART_TXBUF, *Buffer);

    TxAttemptCount = 0;
    Buffer++;
    BytesLeft--;
}
TxAttemptCount++;
}

return NumberOfBytes;
}

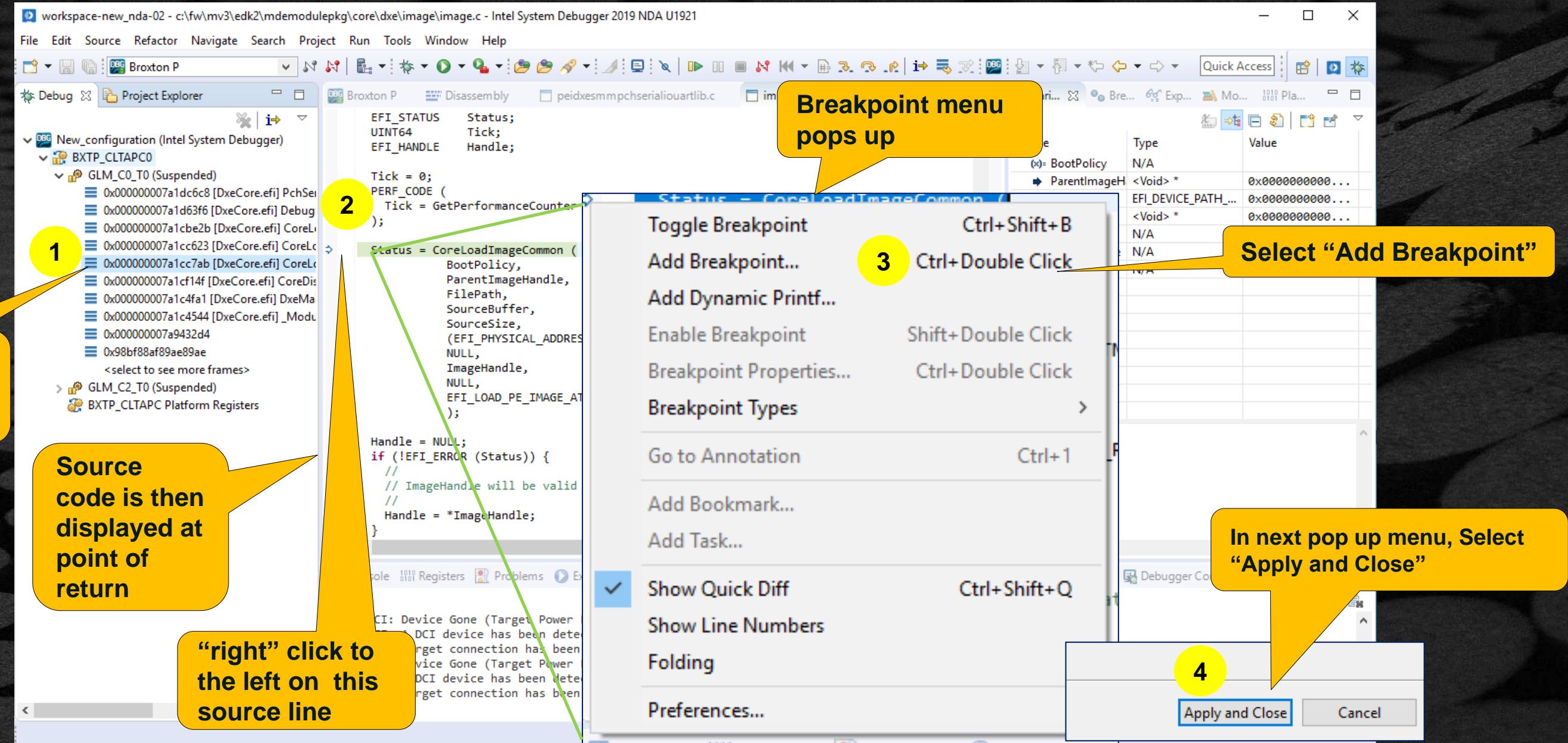
/**
 * Read data from serial device and save the datas in buffer.
 *
 * If the buffer is NULL, then return 0;
 * if NumberOfBytes is zero, then return 0.
 */

```

Retrieving the target topology...
Initialized 'target' and 'threads' objects

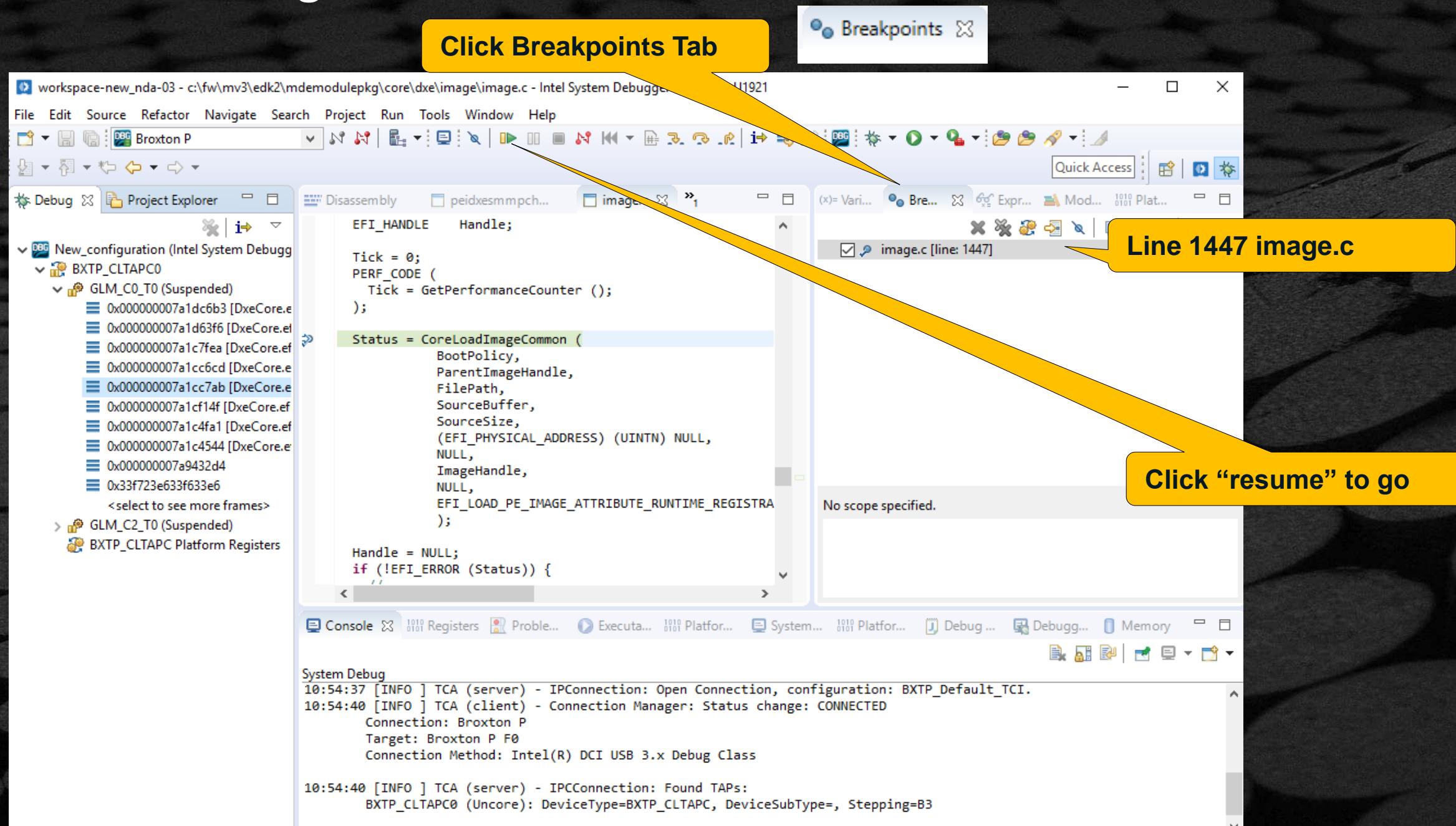
In [1]: INFO: DCI: Device Gone (Target Power Lost or Cable Unplugged)
INFO: DCI: A DCI device has been detected, attempting to establish connection
INFO: DCI: Target connection has been fully established

Set a Breakpoint Using the Stack



Check the Breakpoints Tab

Breakpoint set in image.c line 1447



Observe at the Breakpoint

Serial Debug Output shows execution of each DXE module

The diagram illustrates the workflow for observing DXE module execution. It consists of two main windows: a Serial Debug Output window (Tera Term VT) and an Intel System Debugger interface.

Serial Debug Output Window (Left):

```

VT COM4 - Tera Term VT
File Edit Setup Control Window Help
PROGRESS CODE: U3040003 I0
Loading driver BB65942B-521F-4EC3-BAF9-A92540CF60D2
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 770CB0C0
Loading driver at 0x00077002000 EntryPoint=0x00077002384 SataController.efi
InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 770CBA18
ProtectUefiImageCommon - 0x770CB0C0
- 0x000000077002000 - 0x00000000000003CC0
PROGRESS CODE: U3040002 I0
InstallProtocolInterface: 18A031AB-B443-4D1A-A5C0-0C09261E9F71 770058F0
InstallProtocolInterface: 6A7A5CFF-E8D9-4F70-BADA-75AB3025CE14 77005920
PROGRESS CODE: U3040003 I0
Loading driver 19DF145A-B1D4-453F-8507-38816676D7F6
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 770CB540
Loading driver at 0x00076FF2000 EntryPoint=0x00076FF2384 AtaBusDxe.efi
InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 770CAF98
ProtectUefiImageCommon - 0x770CB540
- 0x000000076FF2000 - 0x0000000000007660
PROGRESS CODE: U3040002 I0
InstallProtocolInterface: 18A031AB-B443-4D1A-A5C0-0C09261E9F71 76FF8C80
InstallProtocolInterface: 107A772C-D5E1-11D4-9A46-0090273FC14D 76FF8EA8
InstallProtocolInterface: 6A7A5CFF-E8D9-4F70-BADA-75AB3025CE14 76FF8E90
PROGRESS CODE: U3040003 I0
Loading driver 5E523CB4-D397-4986-87BD-A6DD8B22F455
InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 770CA840
Loading driver at 0x00076FDA000 EntryPoint=0x00076FDA3A4 AtaAtapiPassThruDxe.efi
InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 770CA718
ProtectUefiImageCommon - 0x770CA840
- 0x000000076FDA000 - 0x0000000000000000
PROGRESS CODE: U3040002 I0
InstallProtocolInterface: 18A031AB-B443-4D1A-A5C0-0C09261E9F71 76FE4650
InstallProtocolInterface: 107A772C-D5E1-11D4-9A46-0090273FC14D 76FE4680
InstallProtocolInterface: 6A7A5CFF-E8D9-4F70-BADA-75AB3025CE14 76FE4698
PROGRESS CODE: U3040003 I0
Loading driver 0167CCC4-D0F7-4F21-A3EF-9E64B7CDCE8B

```

Intel System Debugger Interface (Bottom Left):

```

\mdemodulepkg\core\dxe\image\image.c - Intel System Debugger 2019 NDA U1921
Search Project Run Tools Window Help
Disassembly peidxesmmpch...

```

Annotations:

- Last Dxe Module Loaded and Executed**: Points to the last module listed in the Serial Debug Output.
- Press “resume”**: Points to the resume button in the Intel System Debugger toolbar.
- Next Dxe Module executed**: Points to the next module listed in the Serial Debug Output.

Observe at the Breakpoint

Variable tab

The screenshot shows the Intel System Debugger interface during a debug session. A yellow callout points to the 'Local Variables' table in the Variables tab, which lists several variables with their types and values.

A yellow box highlights the text 'Stopped at line 1447' in the Project Explorer, indicating the current breakpoint location.

The code editor shows a snippet of C code from 'image.c' at line 1447:

```
EFI_HANDLE Handle;  
Tick = 0;  
PERF_CODE (  
    Tick = GetPerformanceCounter ();  
);  
  
Status = CoreLoadImageCommon (  
    BootPolicy,  
    ParentImageHandle,  
    FilePath,  
    SourceBuffer,  
    SourceSize,  
    (EFI_PHYSICAL_ADDRESS) (UINTN) NULL,  
    NULL,  
    ImageHandle,  
    NULL,  
    EFI_LOAD_PE_IMAGE_ATTRIBUTE_RUNTIME_REGISTRATION | EFI_LOAD_PE_IMAGE_ATI  
);  
  
Handle = NULL;  
if (!EFI_ERROR (Status)) {  
    /  
}
```

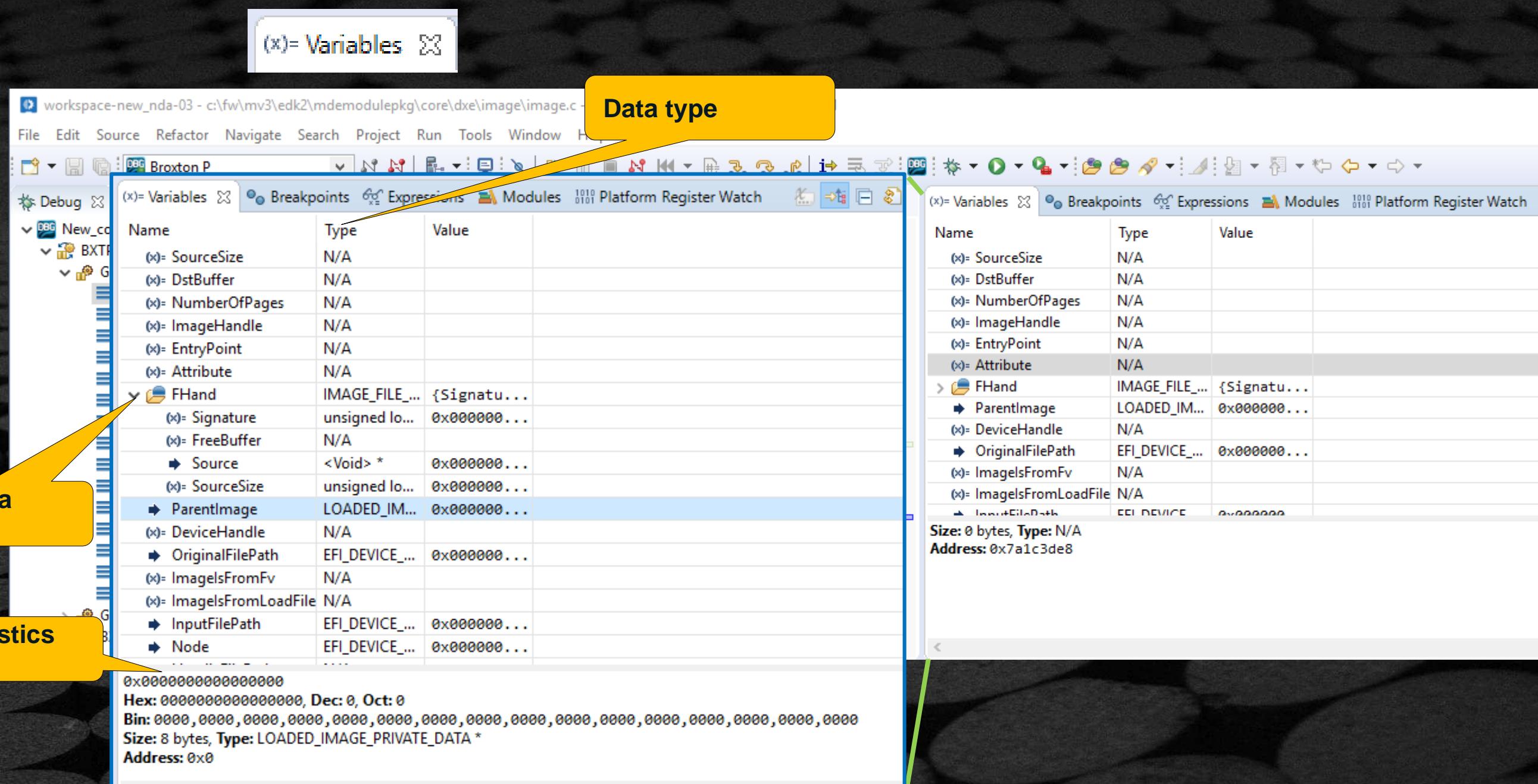
The Variables tab displays the following local variables:

Name	Type	Value
(x)= BootPolicy	N/A	N/A
(x)= ParentImageHandle	N/A	N/A
(x)= FilePath	N/A	N/A
(x)= SourceBuffer	N/A	N/A
(x)= SourceSize	N/A	
(x)= ImageHandle	N/A	
(x)= Status	unsigned lo...	0x000000...

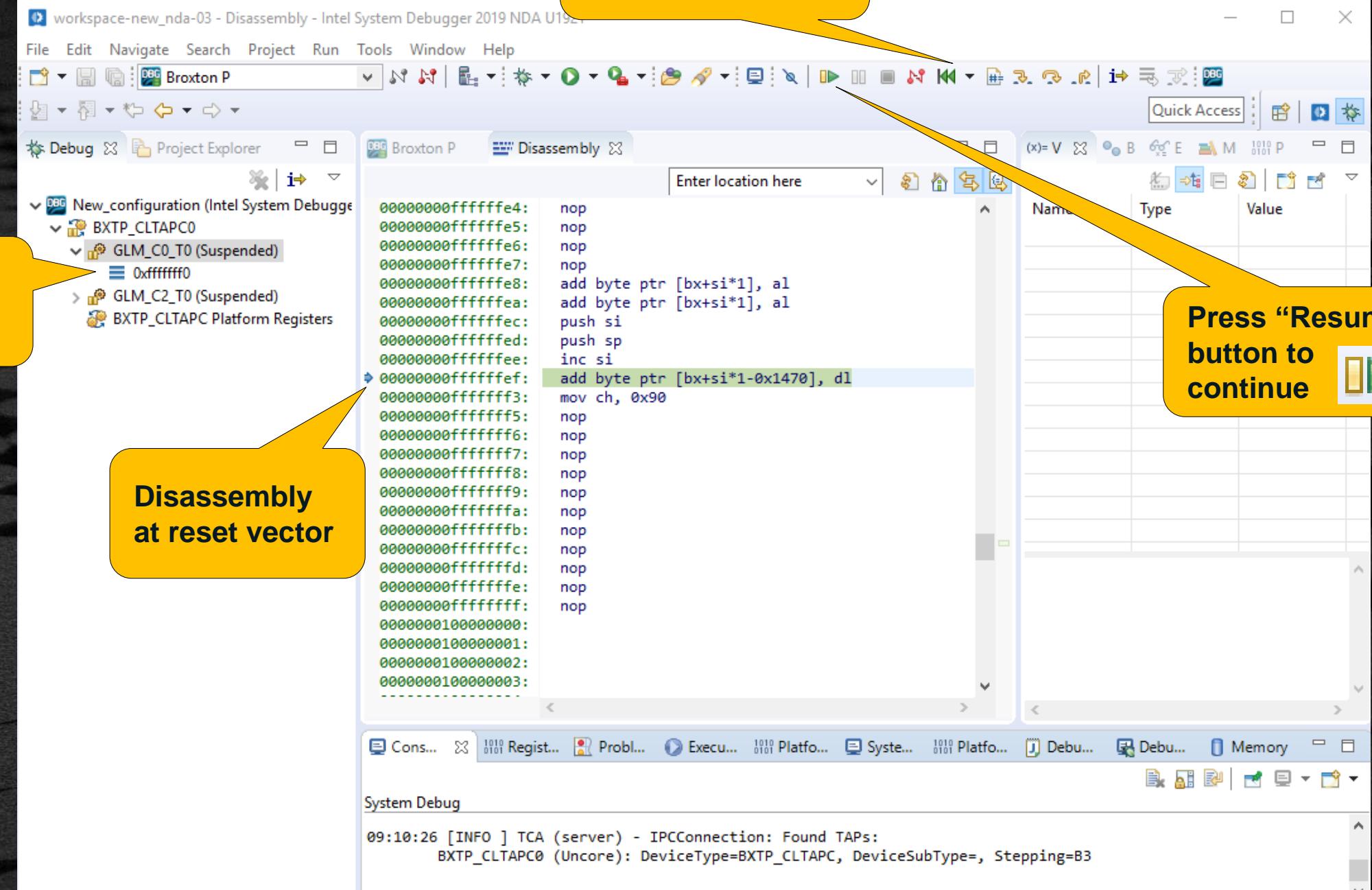
The System Debug console at the bottom shows log messages:

```
10:54:37 [INFO ] TCA (server) - IPConnection: Open Connection, configuration: BXTP_Default_TCI.  
10:54:40 [INFO ] TCA (client) - Connection Manager: Status change: CONNECTED  
    Connection: Broxton P  
    Target: Broxton P F0  
    Connection Method: Intel(R) DCI USB 3.x Debug Class
```

Local Variables



Reset the Target



The screenshot shows the Intel System Debugger interface. The left pane displays the Project Explorer with a configuration named "New_configuration" containing a target "BXTP_CLTAPC0" with two suspended threads: "GLM_C0_T0" and "GLM_C2_T0". The right pane shows the Disassembly view for the "GLM_C0_T0" thread, which is suspended at the reset vector. The assembly code for the first few instructions is:

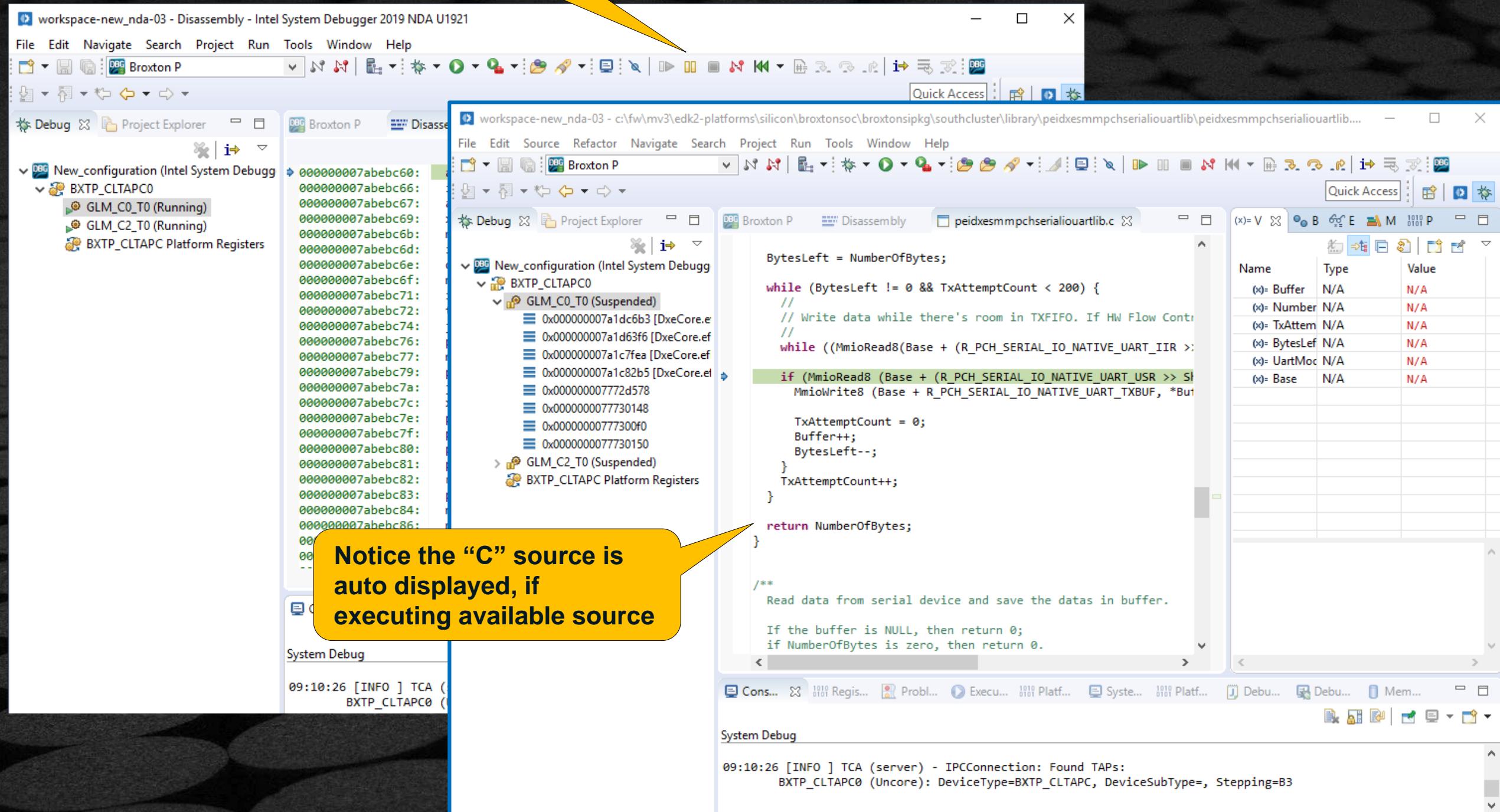
```
00000000fffffe4:    nop
00000000fffffe5:    nop
00000000fffffe6:    nop
00000000fffffe7:    nop
00000000fffffe8:    add byte ptr [bx+si*1], al
00000000fffffea:    add byte ptr [bx+si*1], al
00000000fffffec:    push si
00000000fffffed:    push sp
00000000fffffee:    inc si
00000000fffffef:    add byte ptr [bx+si*1-0x1470], dl
00000000fffffff3:    mov ch, 0x90
00000000fffffff5:    nop
00000000fffffff6:    nop
00000000fffffff7:    nop
00000000fffffff8:    nop
00000000fffffff9:    nop
00000000ffffffa:    nop
00000000ffffffb:    nop
00000000ffffffc:    nop
00000000ffffffd:    nop
00000000ffffffe:    nop
00000000fffffff:    nop
0000000100000000:
0000000100000001:
0000000100000002:
0000000100000003:
```

Annotations in yellow callouts point to specific areas:

- "Stack at reset vector" points to the stack frame in the Project Explorer.
- "Disassembly at reset vector" points to the assembly code in the Disassembly view.
- "Press ‘reset’ button" points to the "Reset" button in the toolbar.
- "Press ‘Resume’ button to continue" points to the "Resume" button in the toolbar.

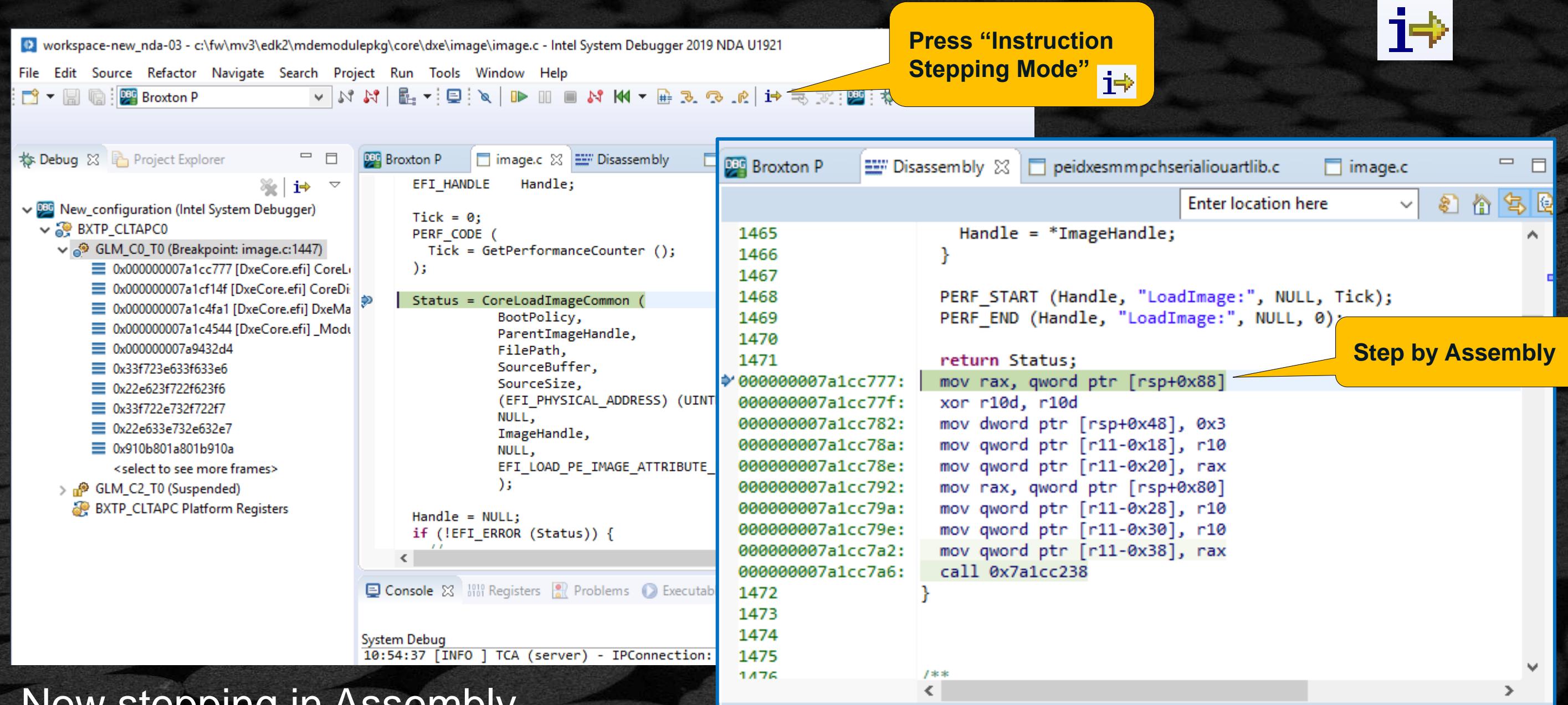
At the bottom, the System Debug window shows a message from TCA (server) indicating it found TAPs for the BXTP_CLTAPC0 target.

Press “suspend”
button to halt

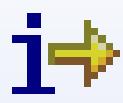


Run then “suspend”, Halt

Switch Between Stepping Assembly & “C” Source



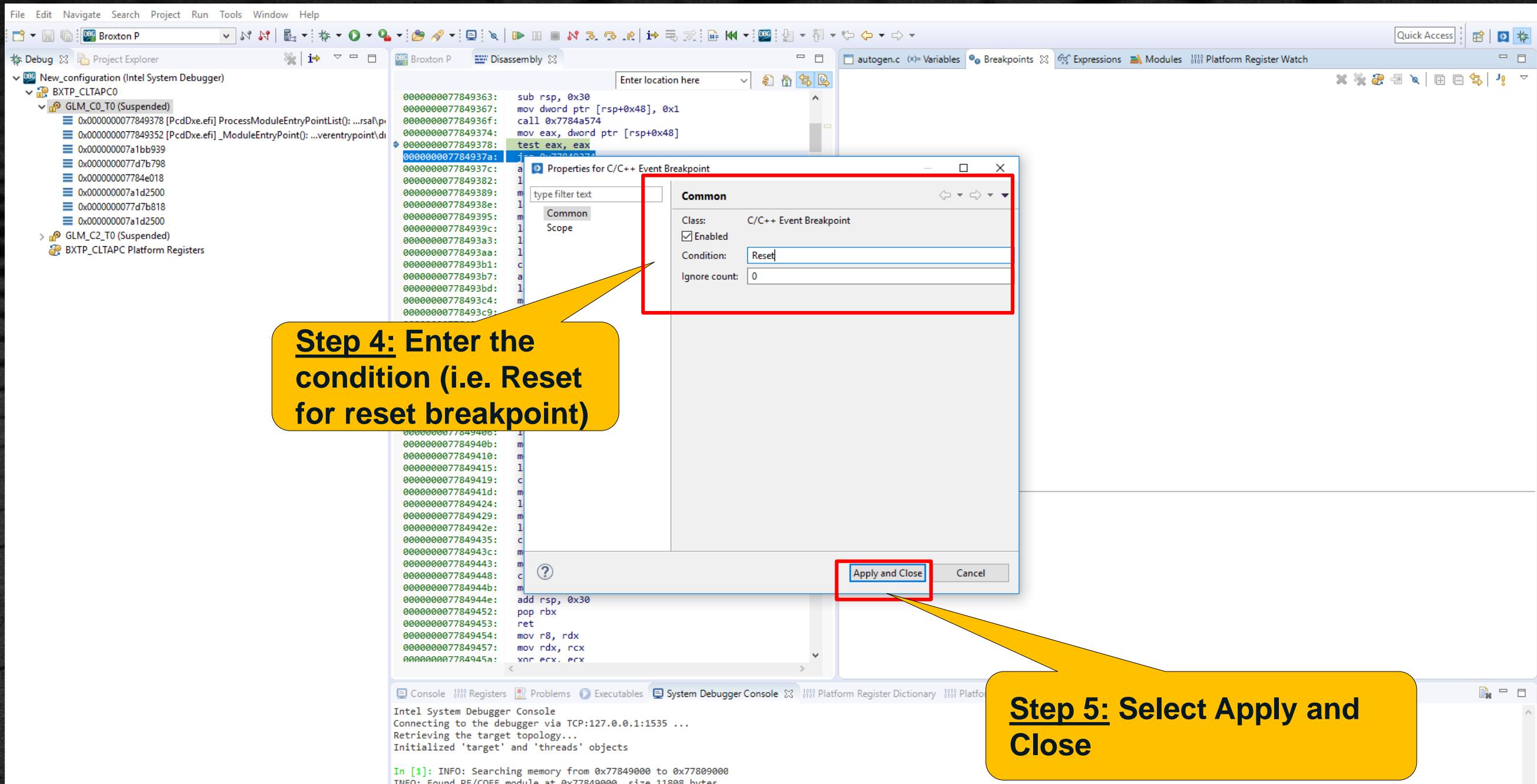
Now stepping in Assembly



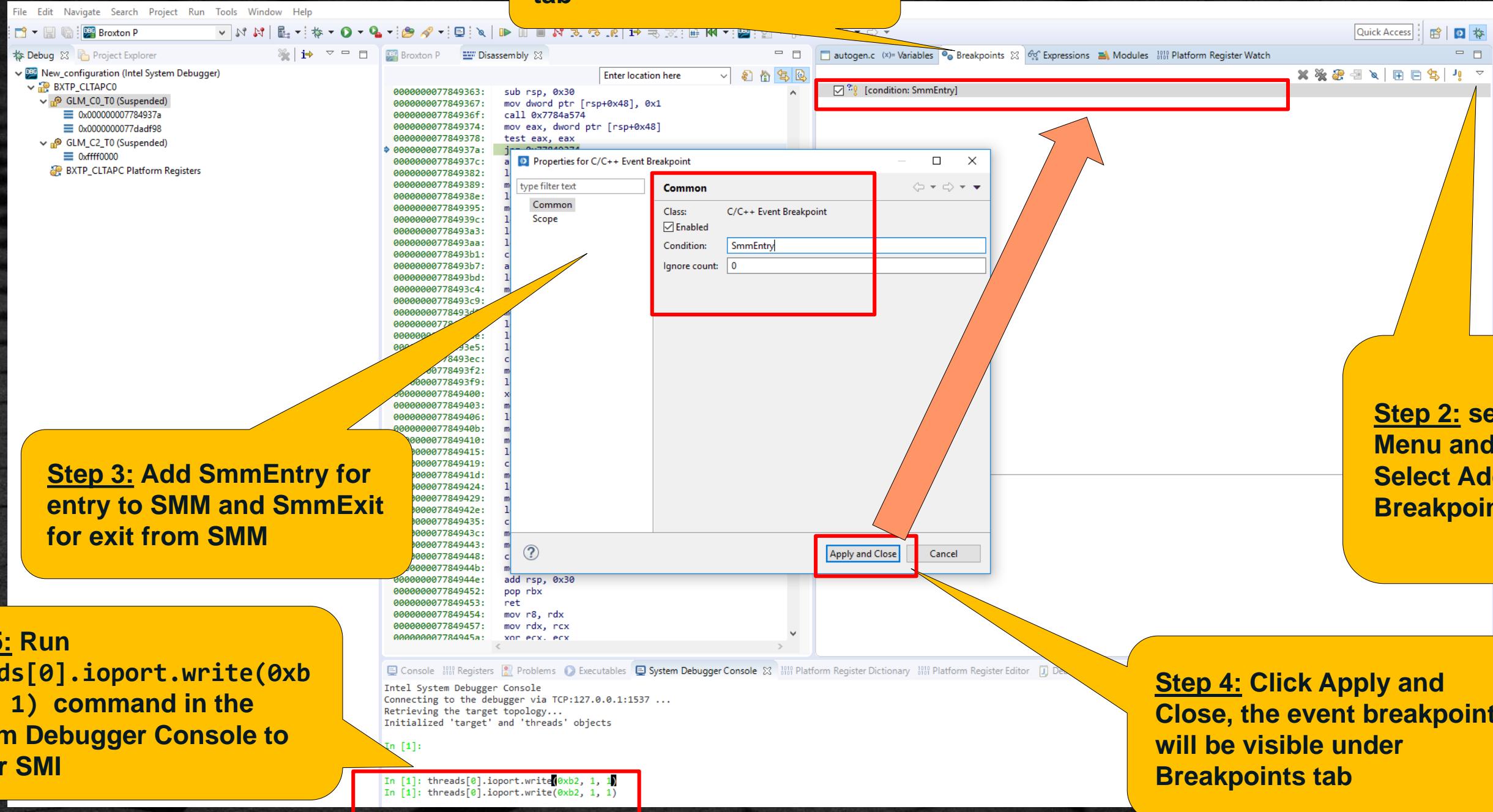
Press “Instruction
Stepping Mode”

Step by Assembly

Set Event Breakpoints



SMM Debugging



Intel® System Studio Debugger

- 2 GUI versions

TCF based
debugger

- Target Communication Framework(TCF) works with the Eclipse IDE
- TCF provides a complete modern debugger for C/C++

XDB Based
debugger

- Intel ITP based debugger supporting JTAG
- Moving to Legacy debugger by Intel® System Studio

The underlying debugging is similar but the user interface is different

Getting Started The System Debugger -XDB legacy

Step 1:

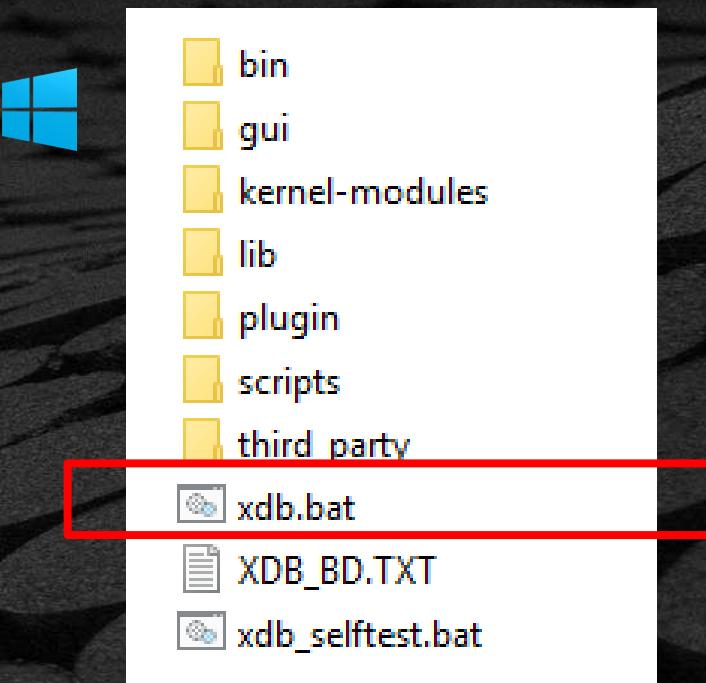
Open the folder where Intel® System Debugger was installed

Step 2:

Select a startup script and launch it - Batch files.

Step 3:

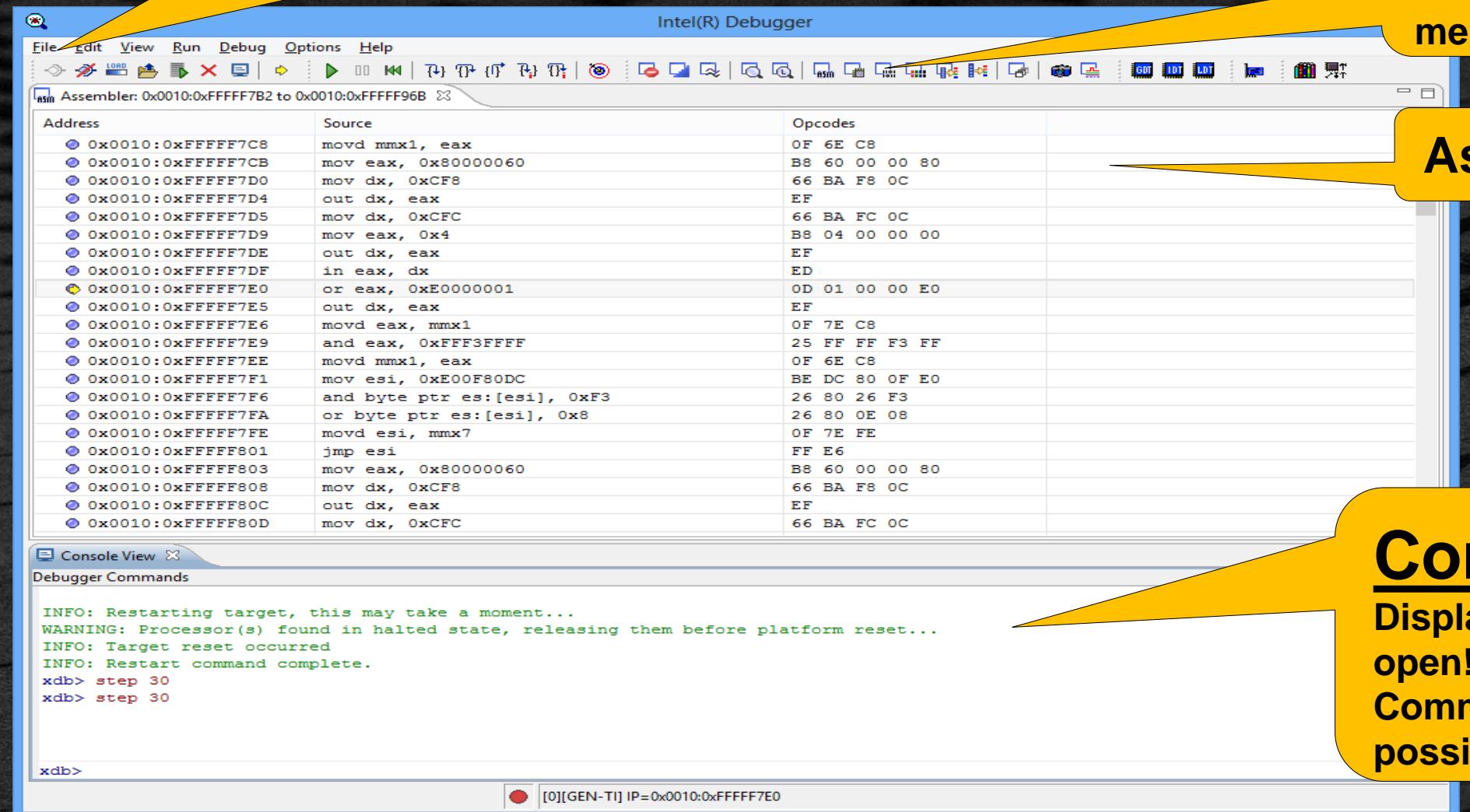
Connect to your target using the “Connect” button (or Ctrl-R)



:
opt/intel/system_stud
io_2019/system_debugg
er_2019/system_debug/
xdb.sh

Initial Startup & overview

Connect/Disconnect



Toolbars / Menus

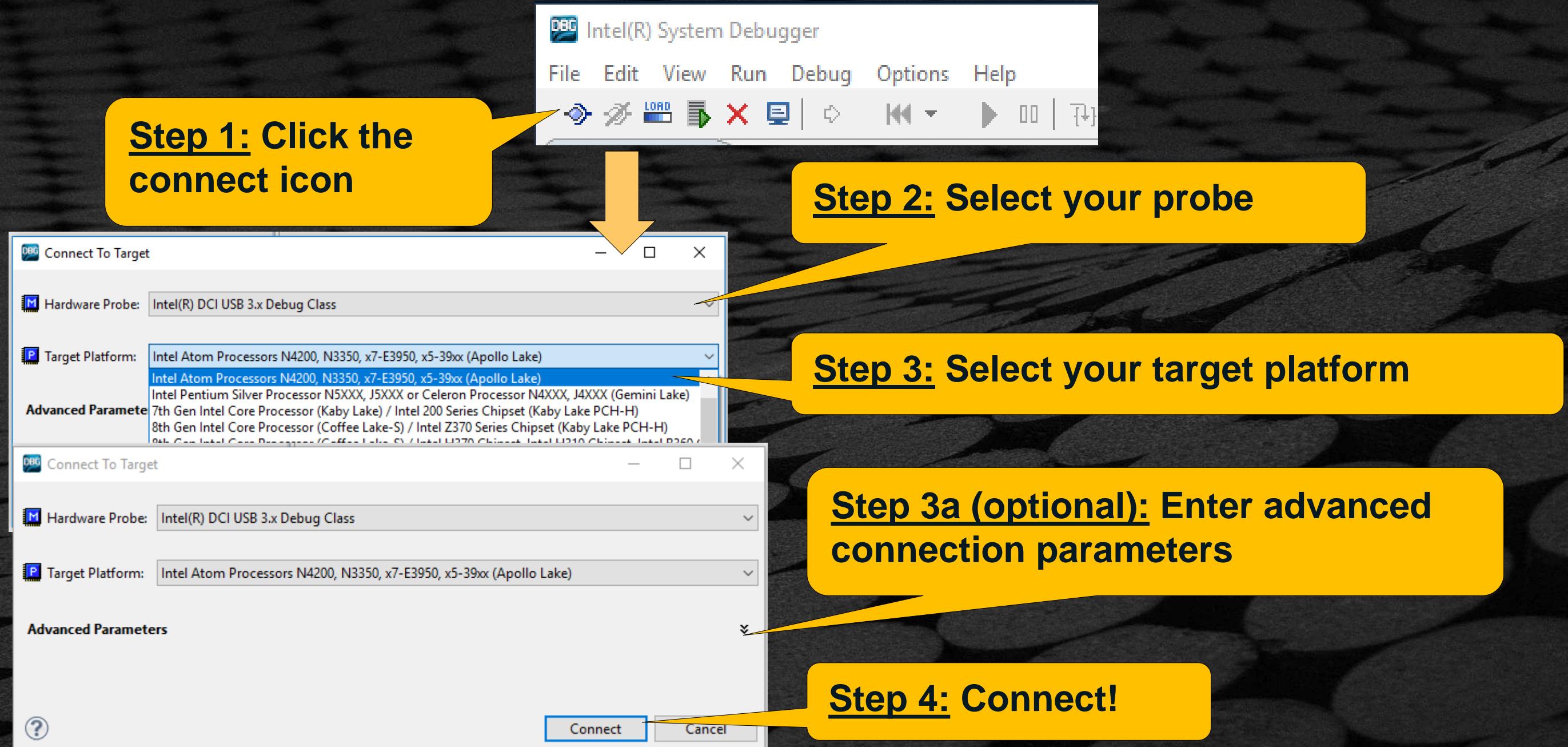
Most functions available from menus & buttons

Assembly Window

Console Window

Displays status messages (keep it open!)
Command-driven operation also possible

Connecting To Targets



Standard Debugger Features

Linux & EFI Support

```

Intel(R) Debugger
File Name: CpuIoDxe.efi
Assembler: 0x0038:0x00000000A19722EA to 0x0038:0x00000000A197249B
234     if (Event->NotifyLink.ForwardLink != NULL) {
235         RemoveEntryList (&Event->NotifyLink);
236         Event->NotifyLink.ForwardLink = NULL;
237     }
238
239     // Queue the event to the pending notification list
240     //
241     //
242
243     InsertTailList (&gEventQueue[Event->NotifyTpl], &Event->NotifyLink);
244     gEventPending |= (UINTN)(1 << Event->NotifyTpl);
245
246
247
248
249 /**
250  Signals all events in the EventGroup.
251
252  @param EventGroup           The list to signal
253
254 */
255 VOID
256 VOID
257 CoreNotifySignalList (

```

CPU Structures

Expression	Value
Event	0x00000000A2883D18
*Event	
Signature	
Type	
SignalCount	1
SignalLink LIST EN	
ForwardLink	0x00000000A2883E28
BackLink	0x00000000A2883C28

Program State

ID	Address	Function	File
0	0x00000000A19722DA	CoreNotifyEvent(class ...)	event.c:234
1	0x00000000A2902B78	CpuIoServiceRead(class ...)	cpuio.c:413
2	0x00000000A290300C	DebugPrint(unsigned lo...)	debuglib.c:93
3	0x00000000A197230A	CoreNotifyEvent(class ...)	event.c:234

Instruction Trace [LBR]

[0][GEN-TI] IP=0x0038:0x00000000A197230A [DxeCore.efi] EV

Multiple Source Files

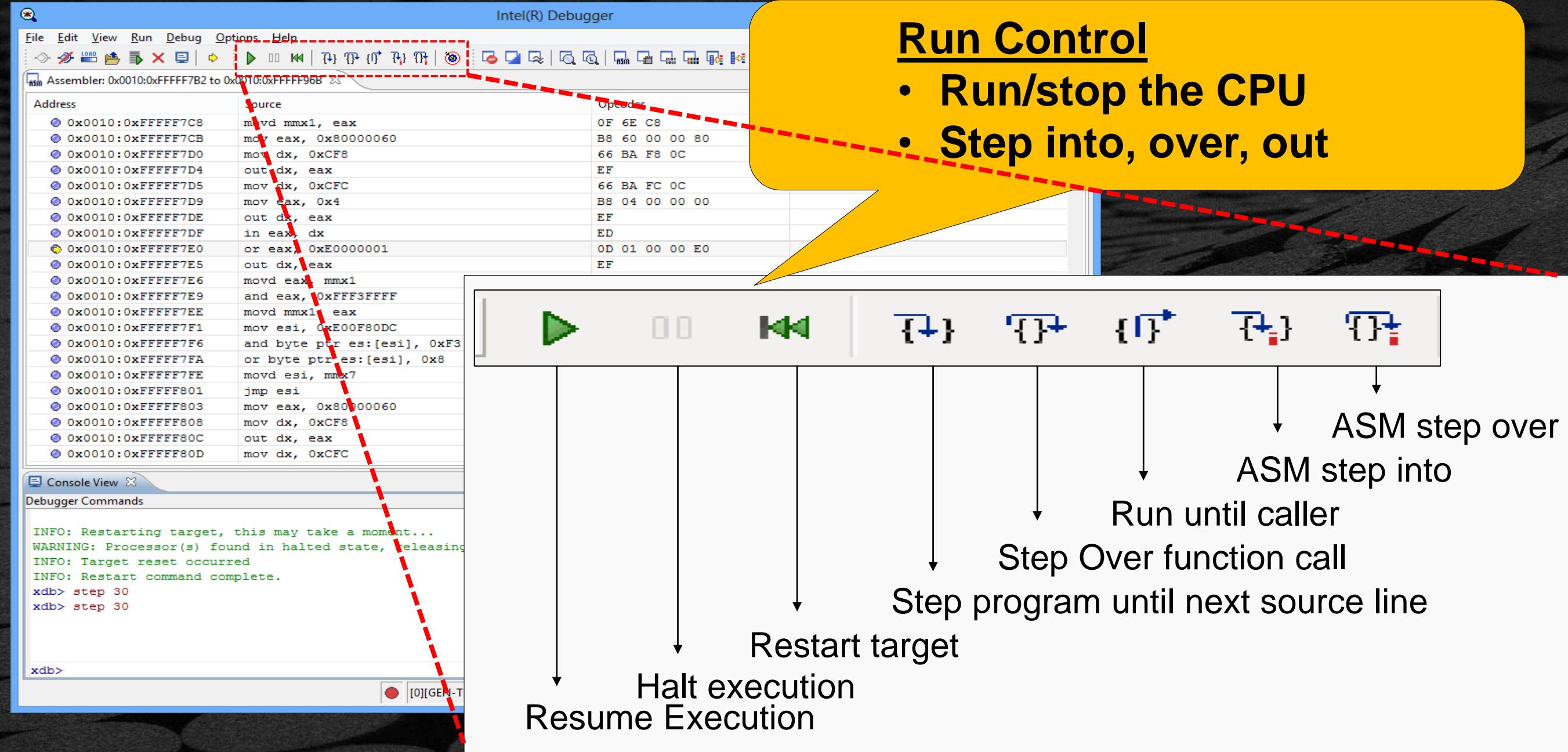
Syntax-highlights

Execution Trace

Breakpoints

And more!

Target Run Control



Multiple HW threads



Select View → Hardware Threads

The debugger can display how the multiple logical cores are used and indicate → which logical core is used by the current code displayed.

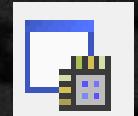
You can select a logical core and continue debugging the SW running there.

A screenshot of a debugger's "Hardware Threads" window. The window has tabs for "Hardware Threads" and "Breakpoints", with "Hardware Threads" selected. It displays a table with columns: Name, Id, State, Address, Location, and File. There is a hierarchical tree view on the left under "Name". Under the root node "IA", two threads are listed: "GLM_C0_T0" (Id 0, stopped at address 0x0010:0x...) and "GLM_C2_T0" (Id 1, stopped at address 0xF000:0x...). The "GLM_C0_T0" row is highlighted with a yellow arrow pointing to it, indicating it is the current thread being debugged.

Name	Id	State	Address	Location	File
IA					
GLM_C0_T0	0	stopped	0x0010:0x...		
GLM_C2_T0	1	stopped	0xF000:0x...		

Modify Register Flags

View → Registers Double left click “EFL”



DEB Modify Register EFL

Register Value: 0x00000206 Original Value: 0x00000206
Group Value: 0x00 CF 0:0

Register Layout:

The diagram shows the bit layout of the EFLAGS register. Bit 31 is reserved. Bits 30-28 are reserved. Bits 27-25 are VIP and AC. Bits 24-22 are RF and NT. Bits 21-19 are OF. Bits 18-16 are IOPL (VM, VIF, ID). Bits 15-13 are DF, TF, ZF, AF, PF, and CF. Bits 12-10 are SF and IF. Bits 9-8 are reserved. Bits 7-6 are reserved. Bits 5-4 are reserved. Bits 3-2 are reserved. Bits 1-0 are reserved.

0 = no carry out from most significant bit

Description:

EFLAGS-Register

System Flags

The system flags of the EFLAGS register control I/O, maskable interrupts, debugging, task switching, and the virtual-8086 mode. An application program should ignore these system flags, and should not attempt to change their state. In some systems, an attempt to change the state of a system flag by an application program results in an exception. The flag registers CF, PF, AF, ZF, SF and OF are set or cleared by arithmetic operations to reflect results of the operation.

Set Restore Close

Registers X

Register	Value
RCX	0x0000000077827E18
RDX	0x0000000000000000
RSI	0x0000000077827E18
RDI	0x0000000000000004
RSP	0x000000007A1C3878
RBP	0x0000000000000010
R8	0x000000007808D3B0
R9	0x000000000000001F
R10	0x0000000000000000
R11	0x0000000076F0E018
R12	0x0000000000000014
R13	0x0000000000000014
R14	0x000000007A1E4180
R15	0x0000000000000007
RIP	0x0000000077819461
RFL	0x0000000000000206 R
EAX	0x00000000
EBX	0x77DAC098
ECX	0x77827E18
EDX	0x00000000
ESI	0x77827E18
EDI	0x00000004
ESP	0x7A1C3878
EBP	0x00000010
CS	0x0038
DS	0x0030
SS	0x0030
ES	0x0030
FS	0x0030
GS	0x0030
EIP	0x77819461
EFL	0x00000206 E

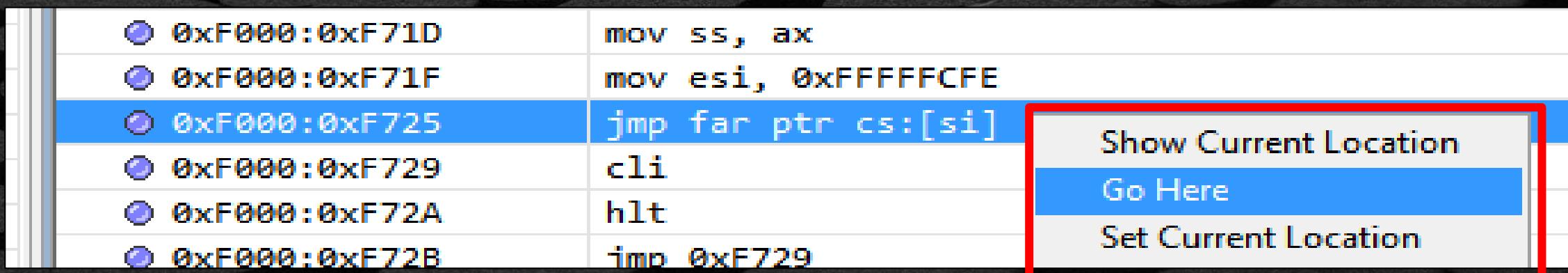
Stepping into UEFI / BIOS code...

Before we can load symbols/source for UEFI we need to be in protected mode

Using the Assembly-level Run Control features you can step from the reset vector to the jump to protected mode:



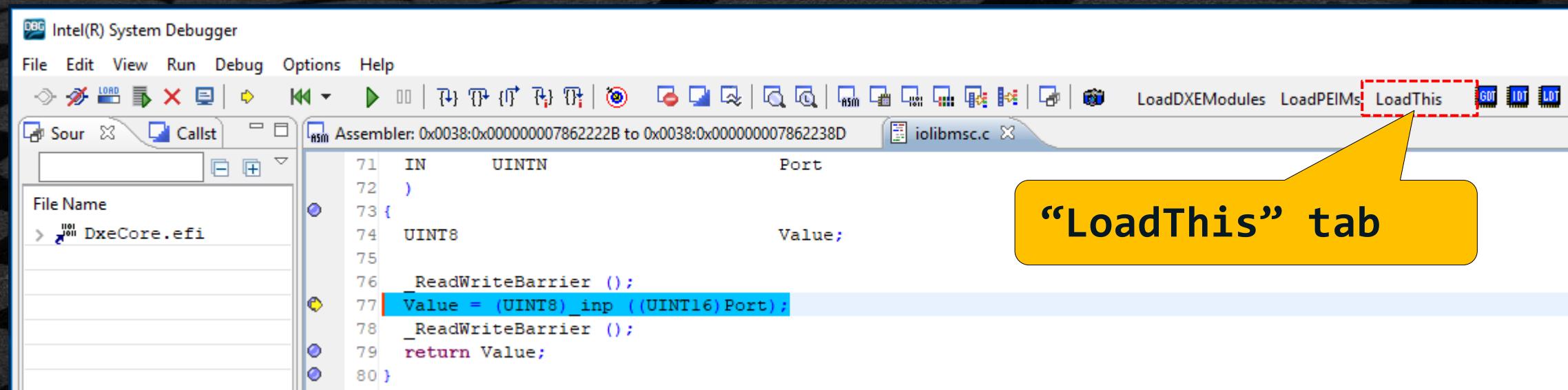
Use the context menu in the ASM window:



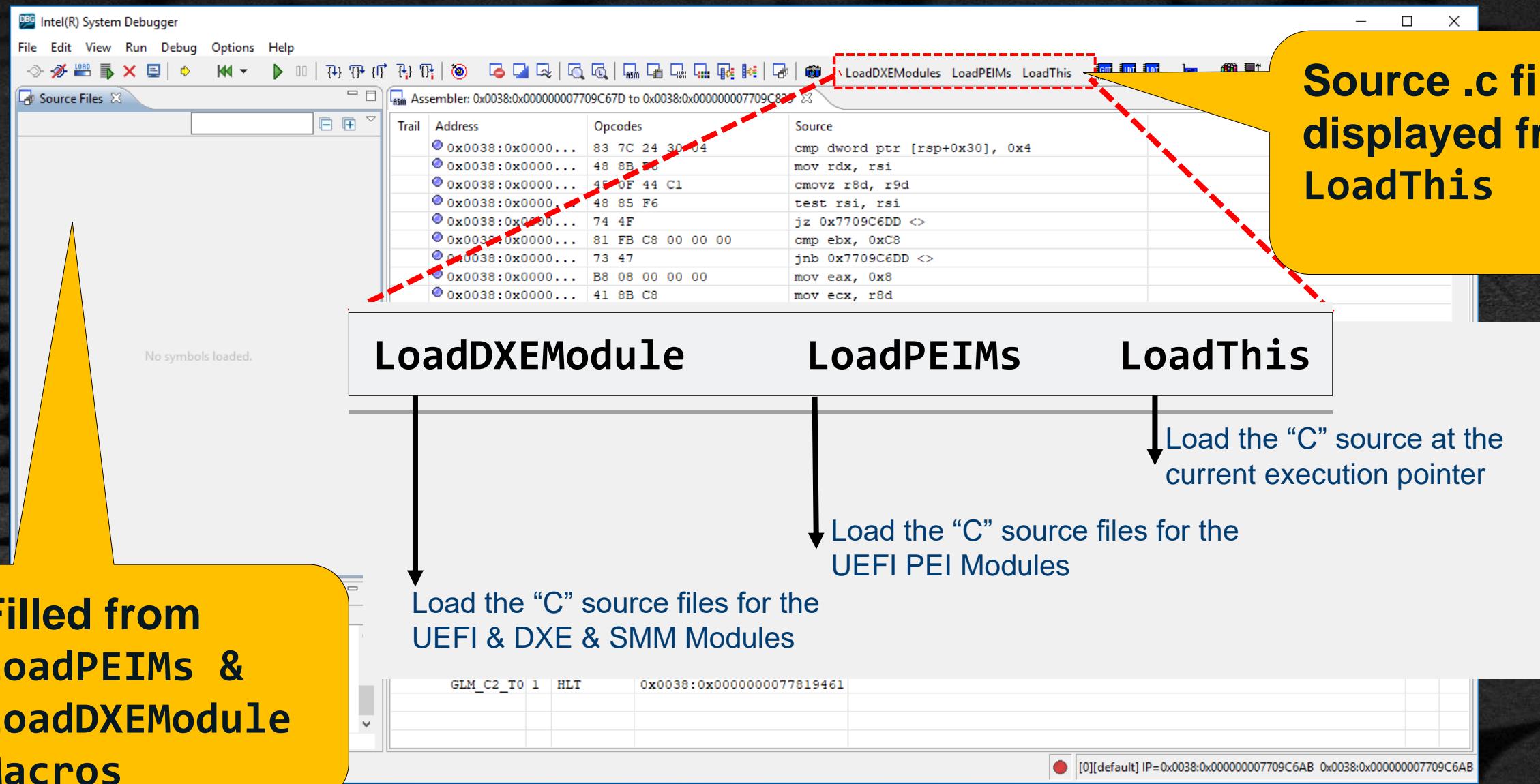
Loading Symbols for UEFI BIOS

Once in protected mode the System Debugger function can load symbols for UEFI using: “loadthis” command or “LoadThis” tab

- Key to UEFI debug: searches memory for a relocatable UEFI module, uses module metadata to locate debug symbols & sources
- Works in all phases of UEFI (SEC, PEI, DXE) (only PEI and DXE with Debug Agent Mode)
- Searches from the current IP or from the a supplied address



UEFI Viewing Source

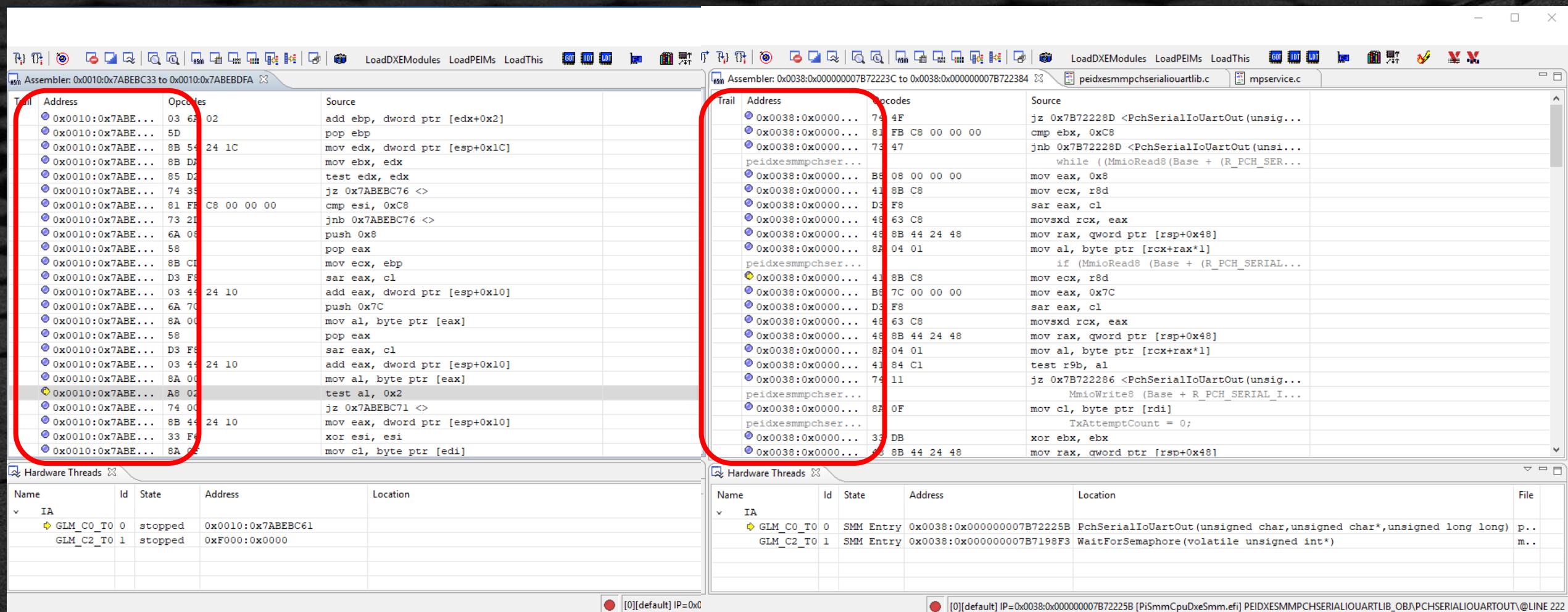


Filled from
LoadPEIMs &
LoadDXEModule
Macros

UEFI Viewing Source PEI vs. DXE

PEI segment at 0x0010:0xnnnn

DXE segment at 0x0038:0xnnnn



The image shows two side-by-side UEFI debugger windows. Both windows have a toolbar at the top with various icons for file operations, assembly, memory, and registers.

Left Window (PEI Segment):

- Title bar: Assembler: 0x0010:0x7ABEBC33 to 0x0010:0x7ABEBDFA
- Table:

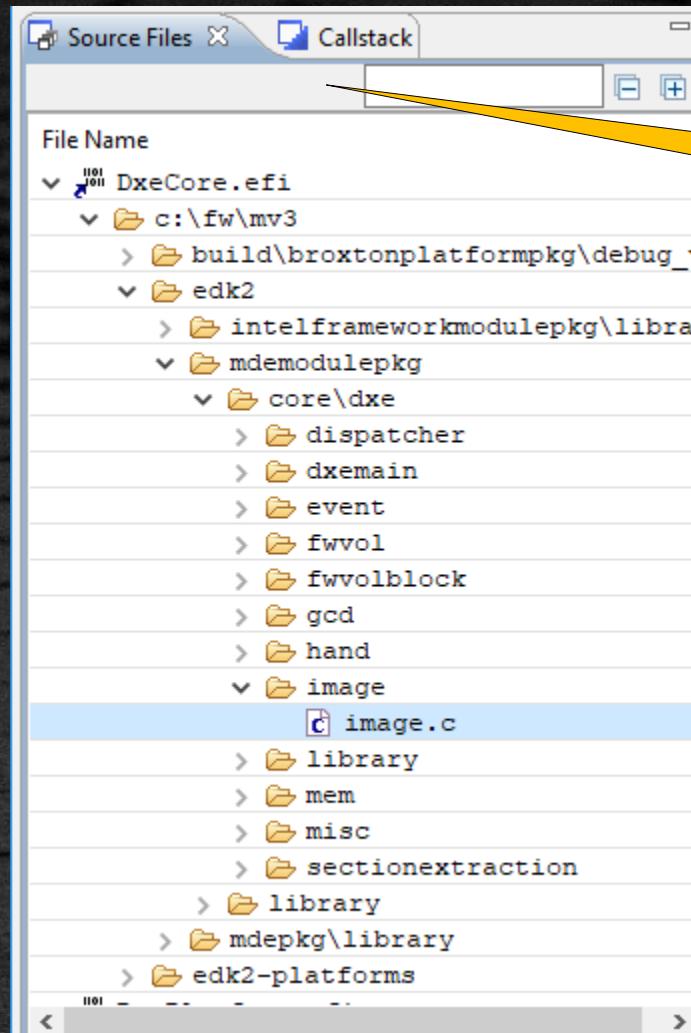
Trail	Address	Opcodes	Source
0x0010:0x7ABE...	03 6B 02		add ebp, dword ptr [edx+0x2]
0x0010:0x7ABE...	5D		pop ebp
0x0010:0x7ABE...	8B 54 24 1C		mov edx, dword ptr [esp+0x1C]
0x0010:0x7ABE...	8B D8		mov ebx, edx
0x0010:0x7ABE...	85 D2		test edx, edx
0x0010:0x7ABE...	74 35		jz 0x7ABEBC76 <>
0x0010:0x7ABE...	81 FF C8 00 00 00		cmp esi, 0xC8
0x0010:0x7ABE...	73 21		jnb 0x7ABEBC76 <>
0x0010:0x7ABE...	6A 08		push 0x8
0x0010:0x7ABE...	58		pop eax
0x0010:0x7ABE...	8B C1		mov ecx, ebp
0x0010:0x7ABE...	D3 F8		sar eax, cl
0x0010:0x7ABE...	03 44 24 10		add eax, dword ptr [esp+0x10]
0x0010:0x7ABE...	6A 70		push 0x7C
0x0010:0x7ABE...	8A 00		mov al, byte ptr [eax]
0x0010:0x7ABE...	58		pop eax
0x0010:0x7ABE...	D3 F8		sar eax, cl
0x0010:0x7ABE...	03 44 24 10		add eax, dword ptr [esp+0x10]
0x0010:0x7ABE...	8A 00		mov al, byte ptr [eax]
0x0010:0x7ABE...	A8 02		test al, 0x2
0x0010:0x7ABE...	74 00		jz 0x7ABEBC71 <>
0x0010:0x7ABE...	8B 44 24 10		mov eax, dword ptr [esp+0x10]
0x0010:0x7ABE...	33 F8		xor esi, esi
0x0010:0x7ABE...	8A F8		mov cl, byte ptr [edi]

Right Window (DXE Segment):

- Title bar: Assembler: 0x0038:0x00000007B72223C to 0x0038:0x00000007B722384
- Table:

Trail	Address	Opcodes	Source
0x0038:0x0000...	74 4F		jz 0x7B72228D <PchSerialIoUartOut(unsigned char,unsigned char*,unsigned long long)
0x0038:0x0000...	81 FB C8 00 00 00		cmp ebx, 0xC8
0x0038:0x0000...	73 47		jnb 0x7B72228D <PchSerialIoUartOut(unsigned char,unsigned char*,unsigned long long)
peidxesmmpchser...			while ((MmioRead8(Base + (R_PCH_SERIAL_I...)) & 0x01) == 0)
0x0038:0x0000...	B8 08 00 00 00		mov eax, 0x8
0x0038:0x0000...	41 8B C8		mov ecx, r8d
0x0038:0x0000...	D3 F8		sar eax, cl
0x0038:0x0000...	48 63 C8		movsxsd rcx, eax
0x0038:0x0000...	48 8B 44 24 48		mov rax, qword ptr [rsp+0x48]
0x0038:0x0000...	8B 04 01		mov al, byte ptr [rcx+rax*1]
peidxesmmpchser...			if (MmioRead8 (Base + (R_PCH_SERIAL_I...)) & 0x01) {
0x0038:0x0000...	41 8B C8		mov ecx, r8d
0x0038:0x0000...	B8 7C 00 00 00		mov eax, 0x7C
0x0038:0x0000...	D3 F8		sar eax, cl
0x0038:0x0000...	48 63 C8		movsxsd rcx, eax
0x0038:0x0000...	48 8B 44 24 48		mov rax, qword ptr [rsp+0x48]
0x0038:0x0000...	8B 04 01		mov al, byte ptr [rcx+rax*1]
0x0038:0x0000...	41 84 C1		test r9b, al
0x0038:0x0000...	74 11		jz 0x7B722286 <PchSerialIoUartOut(unsigned char,unsigned char*,unsigned long long)
peidxesmmpchser...			MmioWrite8 (Base + R_PCH_SERIAL_I...)
0x0038:0x0000...	8B OF		mov cl, byte ptr [rdi]
peidxesmmpchser...			TxAttemptCount = 0;
0x0038:0x0000...	33 DB		xor ebx, ebx
0x0038:0x0000...	8B 8B 44 24 48		mov rax, qword ptr [rsp+0x48]

Select source files to view

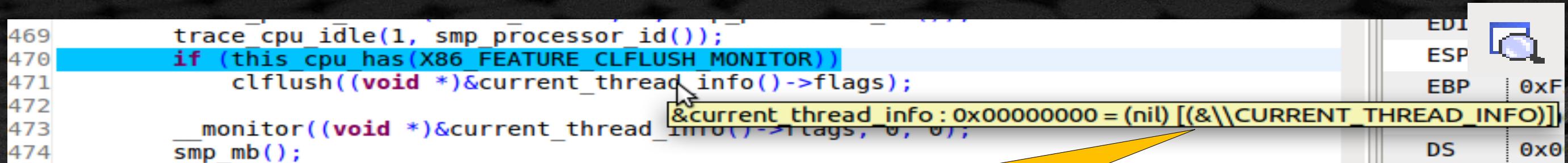


With the 'Source Files' icon you can open a window which contains the source tree as found in the binary file loaded in the debugger.
Just double click on any of the source file names to open the file in the source window

The screenshot shows a source code editor window titled 'image.c'. The code is written in C and includes assembly-like syntax for performance counter ticks and EFI loading. The code snippet shown is:

```
1442 Tick = 0;
1443 PERFORMANCE_CODE (
1444     Tick = GetPerformanceCounter ();
1445 );
1446
1447 Status = CoreLoadImageCommon (
1448     BootPolicy,
1449     ParentImageHandle,
1450     FilePath,
1451     SourceBuffer,
1452     SourceSize,
1453     (EFI_PHYSICAL_ADDRESS) (UINTN) NULL,
1454     NULL,
1455     ImageHandle,
1456     NULL,
1457     EFI_LOAD_PE_IMAGE_ATTRIBUTE_RUNTIME_REGISTRATION | EFI_LOAD_PE_IMAGE_ATTRIBUTE_DEBUG_IMAGE_INFO_TABLE_REGIS
1458 );
1459
1460 Handle = NULL;
1461 if (!EFI_ERROR (Status)) {
1462     //
```

Evaluate symbols



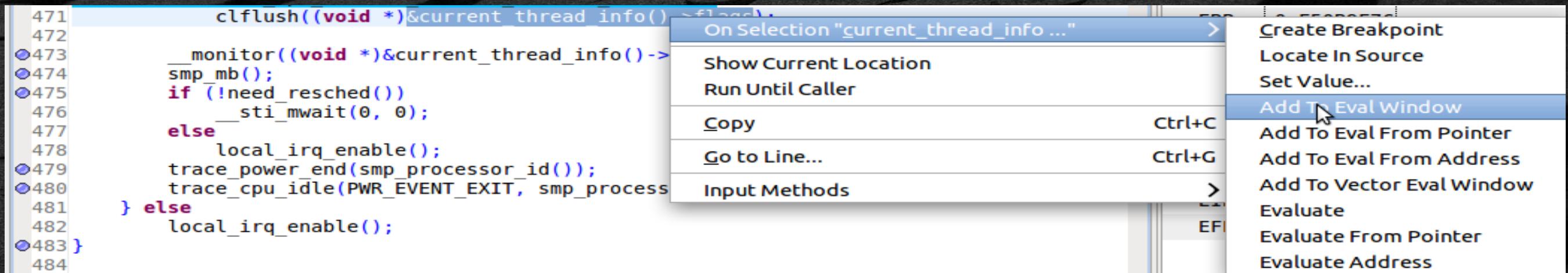
```

469     trace_cpu_idle(1, smp_processor_id());
470     if (this cpu has(X86 FEATURE_CLFLUSH_MONITOR))
471         clflush((void *)&current_thread_info()->flags);
472         monitor((void *)&current_thread_info()->flags, 0, 0);
473     smp_mb();
474

```

Hover the cursor over a variable and the debugger will show you its value

Highlight a variable and use the ‘right mouse click’ – additional options are now available



471 clflush((void *)¤t_thread_info() ->flags);

472 _monitor((void *)¤t_thread_info() ->

473 smp_mb();

474 if (!need_resched())

475 _sti_mwait(0, 0);

476 else

477 local_irq_enable();

478 trace_power_end(smp_processor_id());

479 trace_cpu_idle(PWR_EVENT_EXIT, smp_processor_id());

480 }

481 } else

482 local_irq_enable();

483 }

484 }

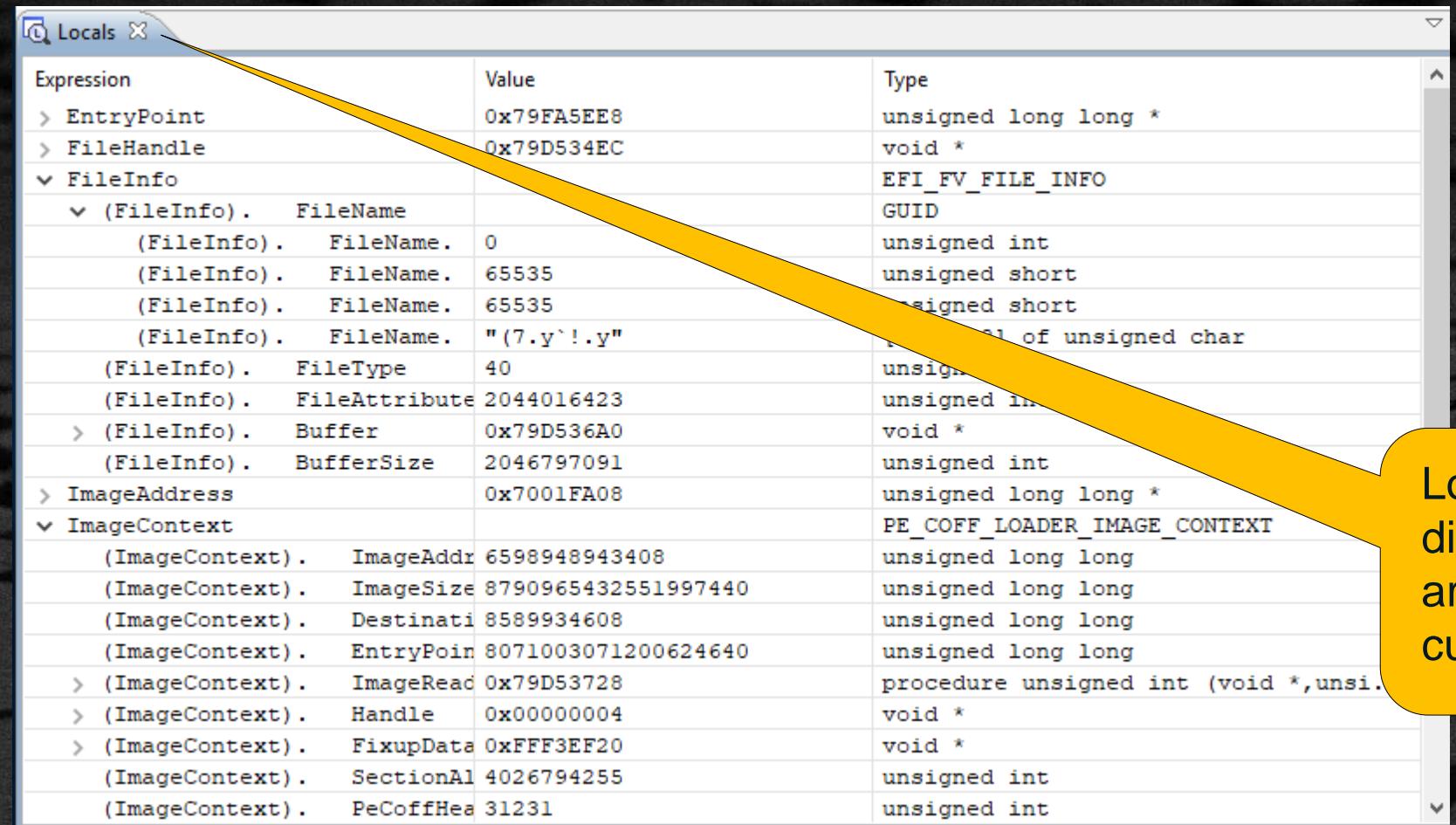
On Selection "current_thread_info ..."

- >Show Current Location
- Run Until Caller
- Add To Eval Window**
- Copy Ctrl+C
- Go to Line... Ctrl+G
- Input Methods

EFI

- Create Breakpoint
- Locate In Source
- Set Value...
- Add To Eval From Pointer
- Add To Eval From Address
- Add To Vector Eval Window
- Evaluate
- Evaluate From Pointer
- Evaluate Address

Local variables



The screenshot shows the Locals window of a debugger. It displays a list of variables with their values and types. A yellow arrow points from the text "In expression field indicate that this is a compound element and that it has been Expanded (> for not expanded)" to the first entry in the table, which is a compound element named 'FileInfo'.

Expression	Value	Type
> EntryPoint	0x79FA5EE8	unsigned long long *
> FileHandle	0x79D534EC	void *
FileInfo		EFI_FV_FILE_INFO
(FileInfo). FileName		GUID
(FileInfo). FileName. 0	0	unsigned int
(FileInfo). FileName. 65535	65535	unsigned short
(FileInfo). FileName. 65535	65535	signed short
(FileInfo). FileName. "(7.y`!.y"	"(7.y`!.y"	array of unsigned char
(FileInfo). FileType	40	unsigned int
(FileInfo). FileAttribute	2044016423	unsigned int
> (FileInfo). Buffer	0x79D536A0	void *
(FileInfo). BufferSize	2046797091	unsigned int
> ImageAddress	0x7001FA08	unsigned long long *
ImageContext		PE_COFF_LOADER_IMAGE_CONTEXT
(ImageContext). ImageAddr	6598948943408	unsigned long long
(ImageContext). ImageSize	8790965432551997440	unsigned long long
(ImageContext). Destination	8589934608	unsigned long long
(ImageContext). EntryPoint	8071003071200624640	unsigned long long
> (ImageContext). ImageRead	0x79D53728	procedure unsigned int (void *, unsi...
> (ImageContext). Handle	0x00000004	void *
> (ImageContext). FixupData	0xFFFF3EF20	void *
(ImageContext). SectionAl	4026794255	unsigned int
(ImageContext). PeCoffHea	31231	unsigned int



Local variable window will display all variables which are accessible from within current scope

The example above shows part of the 'FileInfo' structure

✓ In expression field indicate that this is a compound element and that it has been Expanded (> for not expanded)

How did I reach the current location?

Callstack



The screenshot shows a debugger interface with two main panes. The top pane is a 'Callstack' window with tabs for 'Source Files' and 'Callstack'. It displays a callstack with three entries:

Location	File	Module
PeCoffLoaderExtraActionCommon(class PE_COFF_LOADER, ...)	pecoffextraactionlib.c	PeiCore.efi
PeCoffLoaderRelocateImage(class PE_COFF_LOADER, ...)	basepecoff.c:1179	PeiCore.efi
LoadAndRelocatePeCoffImage(void*, void*, unsigned)	image.c:515	PeiCore.efi

A green arrow points from the text 'Source file: line number' to the third entry in the callstack. The bottom pane is an 'Assembler' window showing assembly code for file 'image.c'. The assembly code corresponds to the C code in the callstack. A yellow arrow points from the text 'Current file' to the tab for 'image.c' in the assembler window.

```
ASM Assembler: 0x79FFA503 to 0x79FFA636
iobmsc.c image.c pecoffextraactionlib.c image.c

509     return Status;
510 }
511 //
512 // Relocate the image in our new buffer
513 //
514 Status = PeCoffLoaderRelocateImage (&ImageContext);
515 if (EFI_ERROR (Status)) {
516     return Status;
517 }
518 //
519 // Flush the instruction cache so the image data is written before we execute it
520 //
521 if (ImageContext.ImageAddress != (EFI_PHYSICAL_ADDRESS) (UINTN) Pe32Data) {
522     InvalidateInstructionCacheRange ((VOID *) (UINTN) ImageContext.ImageAddress, (UINTN) ImageContext.ImageSize);
523 }
524 //
525 *ImageAddress = ImageContext.ImageAddress;
526 *ImageSize = ImageContext.ImageSize;
```

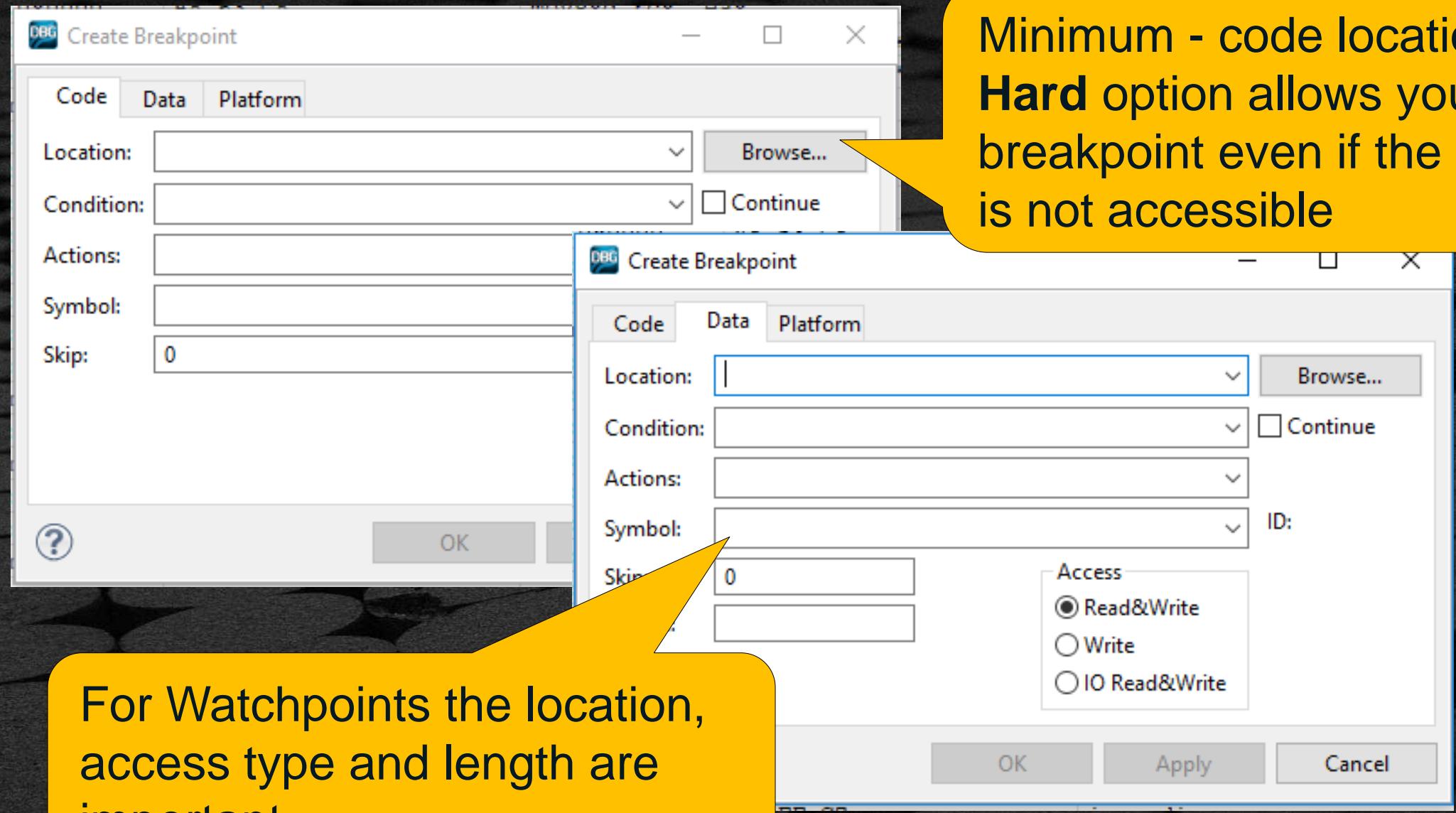


Source file:
line number

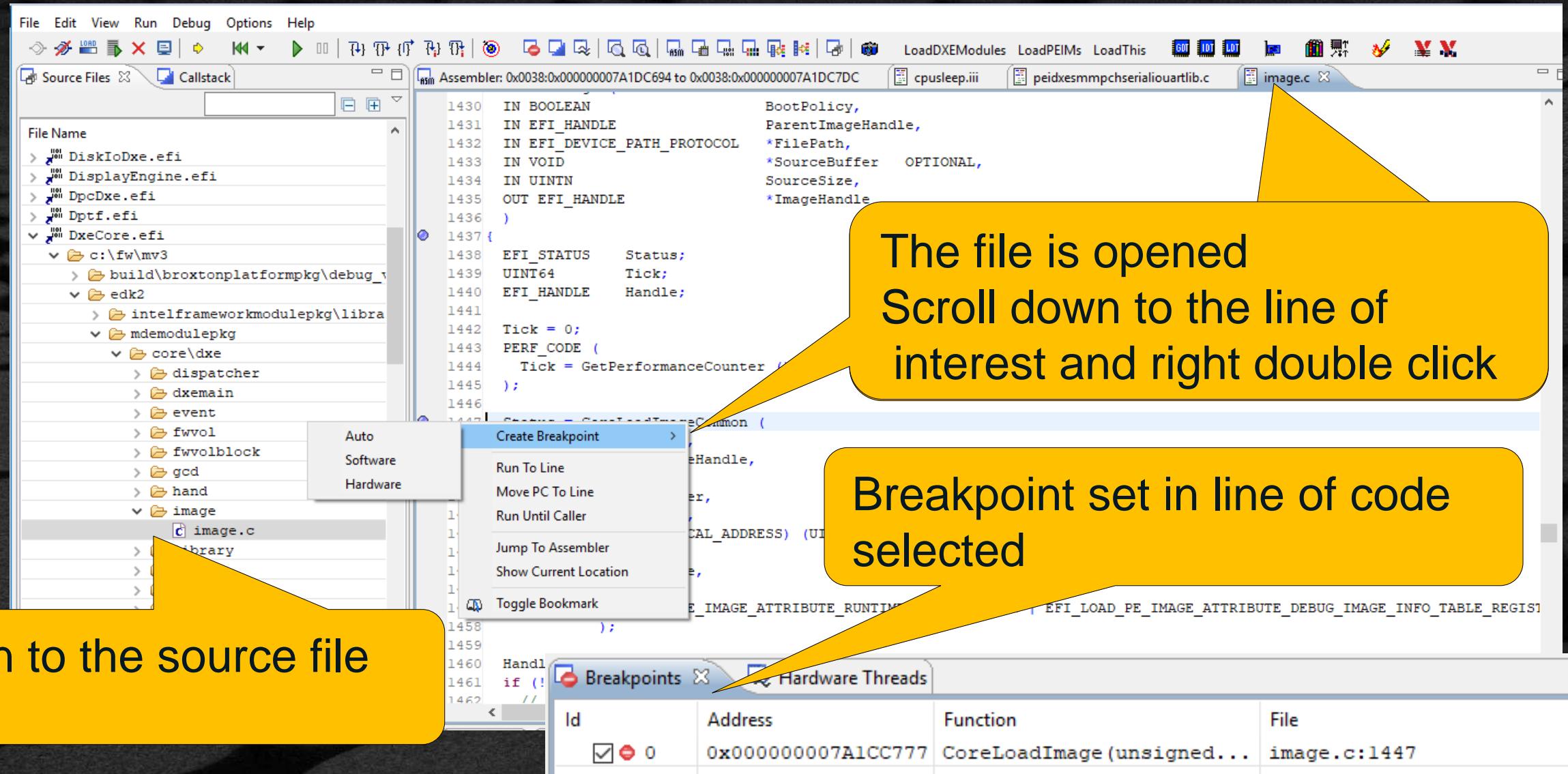
→ Current file

Create a breakpoint

Debug → create Breakpoint



Create a breakpoint in Source



What happened? Inspecting System State

Intel System Debugger presents state in a “Human Readable” format

- Relationships: “*show the nesting of structure members*”
- High-level decode: “*show the names of bits in a register*”
- Context: “*show the documentation for a register*”

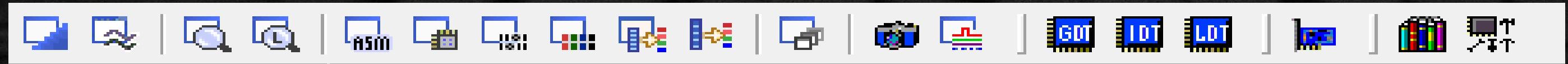
Program State:

- Memory contents -> “Memory Window”
- Program State -> “Locals Window / Eval Window”

CPU State:

- General and Extended Registers
- Page Tables
- IA CPU data structures (GDT, IDT, LDT)

Overview of State Windows:



Callstack & HW Threads

Program Variables

Memory & Registers

Page Tables

Linux Kernel State

PCI Topology & Devices

Processor Structures

Execution Trace

Program State:

The screenshot shows the TianoCore debugger interface with two main windows highlighted by yellow callouts.

Callstack Window

Show call stack for the currently selected hardware thread

Locals

Program state for the currently selected callstack frame

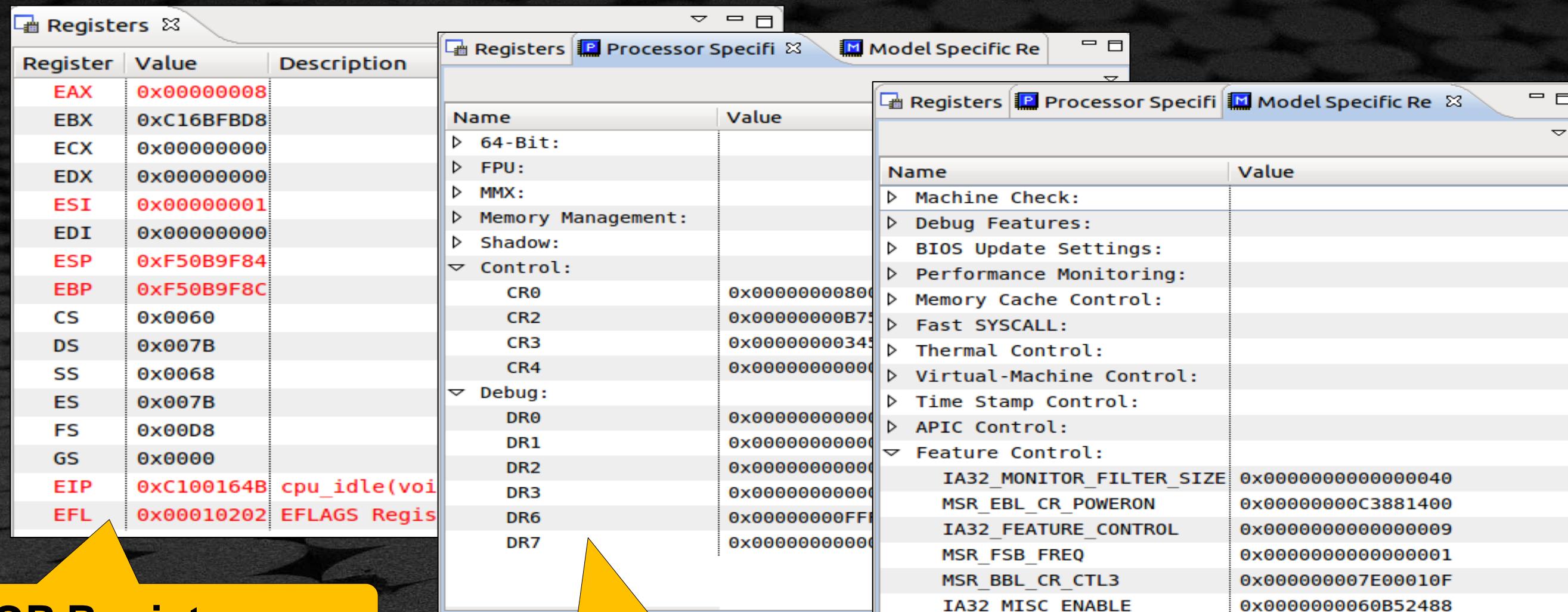
Callstack Window (Top Left): This window displays the call stack for the currently selected hardware thread. It has tabs for "Source Files" and "Callstack". The "Callstack" tab is active, showing a list of function calls with their file names and line numbers. A yellow callout points from the text "Show call stack for the currently selected hardware thread" to this window.

Location	File
InternalMemcpyMem(void)	copymem.iii:36
SecStartupPhase2(void*)	secmain.c:200
SecStartup(unsigned int,unsigned int,v	secmain.c:155
StartUpAp(void)	
lost frame-chain ...	

Locals (Bottom Right): This window displays the program state for the currently selected callstack frame. A yellow callout points from the text "Program state for the currently selected callstack frame" to this window.

Expression	Value	Type
AllSecPpiList	array [0 ... 5]	[array=6] of EFI_PEI_PPI_DESCRIPTOR
Context	0xFEFBFE94	void *
PeiCoreEntryPoint	0xFEFBFFC8	procedure void (_EFI_SEC_PROTOCOL * _PEI_CORE_ENTRYPOINT)
PpiList	0xFFFFFFF78	EFI_PEI_PPI_DESCRIPTOR *
*PpiList		EFI_PEI_PPI_DESCRIPTOR
Flags	1114084	unsigned int
Guid	0xFFFF8E00	GUID *
Ppi	0x6F8C2B35	void *

CPU State: Register Windows



The screenshot displays three windows showing CPU register states:

- GP Registers:** Shows standard general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, CS, DS, SS, ES, FS, GS, EIP, EFL) with their values.
- CPU Registers:** Shows various control and debug registers (CR0, CR2, CR3, CR4, DR0, DR1, DR2, DR3, DR6, DR7) with their values.
- Model-Specific Registers:** Shows model-specific registers (Machine Check, Debug Features, BIOS Update Settings, Performance Monitoring, Memory Cache Control, Fast SYSCALL, Thermal Control, Virtual-Machine Control, Time Stamp Control, APIC Control, Feature Control) with their values.

Register	Value	Description
EAX	0x00000008	
EBX	0xC16BFBD8	
ECX	0x00000000	
EDX	0x00000000	
ESI	0x00000001	
EDI	0x00000000	
ESP	0xF50B9F84	
EBP	0xF50B9F8C	
CS	0x0060	
DS	0x007B	
SS	0x0068	
ES	0x007B	
FS	0x00D8	
GS	0x0000	
EIP	0xC100164B	cpu_idle(voi
EFL	0x00010202	EFLAGS Regis

Name	Value
▷ 64-Bit:	
▷ FPU:	
▷ MMX:	
▷ Memory Management:	
▷ Shadow:	
▷ Control:	
CR0	0x0000000080000000
CR2	0x000000000B750000
CR3	0x0000000003450000
CR4	0x0000000000000000
▷ Debug:	
DR0	0x0000000000000000
DR1	0x0000000000000000
DR2	0x0000000000000000
DR3	0x0000000000000000
DR6	0x00000000FF000000
DR7	0x0000000000000000

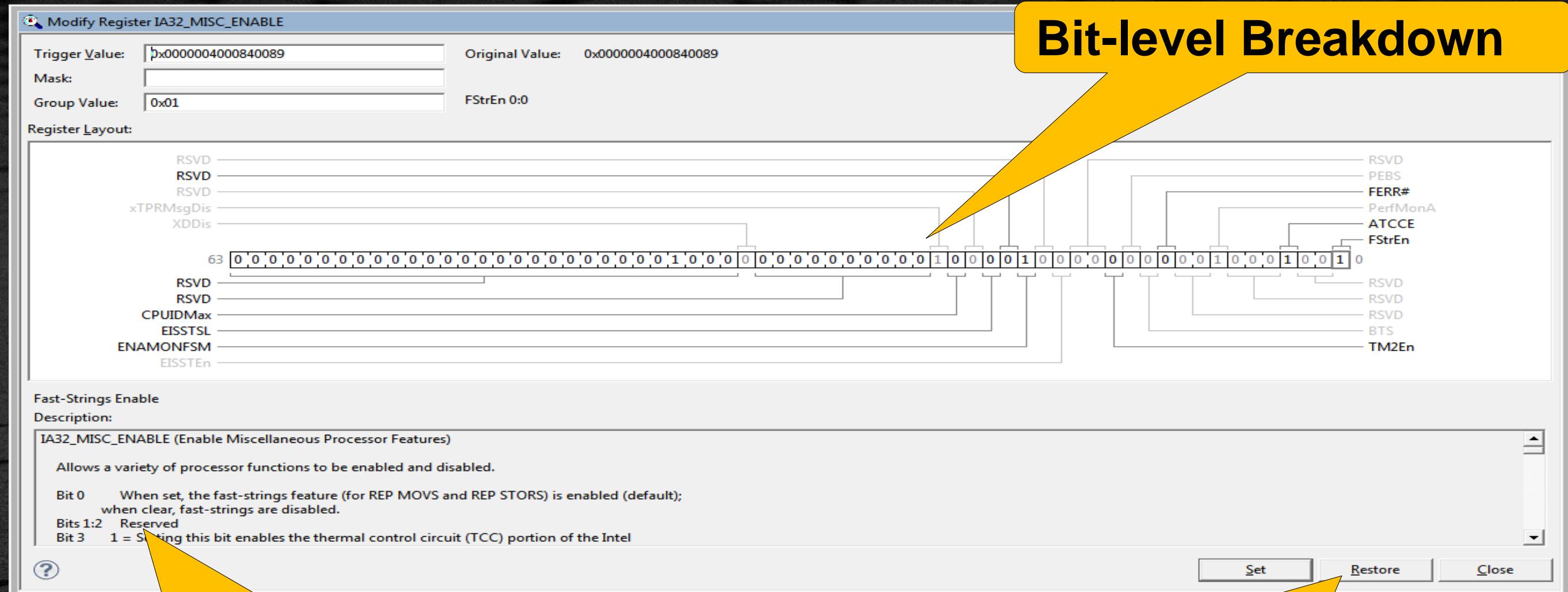
Name	Value
▷ Machine Check:	
▷ Debug Features:	
▷ BIOS Update Settings:	
▷ Performance Monitoring:	
▷ Memory Cache Control:	
▷ Fast SYSCALL:	
▷ Thermal Control:	
▷ Virtual-Machine Control:	
▷ Time Stamp Control:	
▷ APIC Control:	
▷ Feature Control:	
IA32_MONITOR_FILTER_SIZE	0x0000000000000040
MSR_EBL_CR_POWERON	0x00000000C3881400
IA32_FEATURE_CONTROL	0x0000000000000009
MSR_FSB_FREQ	0x0000000000000001
MSR_BBL_CR_CTL3	0x000000007E00010F
IA32_MISC_ENABLE	0x0000000060B52488

GP Registers

CPU Registers

Model-Specific Registers

CPU State: Detailed Register View



Bit-level Breakdown

Text Documentation

Set/Restore Values

What about paging?

Console View IDT: 0xc167a000 GDT: 0xf5c00000 Paging

Index	Virtual Memory Range	Physical Base Address	Description
PD [768]	0xC0000000 to 0xC03FFFFF	0x017B6000	P=1 R/W=1 U/S=1 PWT=0 PCD=0 A=1 D=1 PS=0 (4KB) PAT=0
PT [0]	0xC0000000 to 0xC0000FFF	0x00000000	P=1 R/W=1 U/S=0 PWT=0 PCD=0 A=1 D=1 PAT=0 G=1
PT [1]	0xC0001000 to 0xC0001FFF	0x00001000	P=1 R/W=1 U/S=0 PWT=0 PCD=0 A=1 D=1 PAT=0 G=1
PT [2]	0xC0002000 to 0xC0002FFF	0x00002000	P=1 R/W=1 U/S=0 PWT=0 PCD=0 A=1 D=1 PAT=0 G=1
PT [3]	0xC0003000 to 0xC0003FFF	0x00003000	P=1 R/W=1 U/S=0 PWT=0 PCD=0 A=1 D=1 PAT=0 G=1
PT [4]	0xC0004000 to 0xC0004FFF	0x00004000	P=1 R/W=1 U/S=0 PWT=0 PCD=0 A=1 D=1 PAT=0 G=1

Modify Page-Table Attributes

Register Value: 0x0163 Original Value: 0x0163
Group Value: 0x01 P 0:0

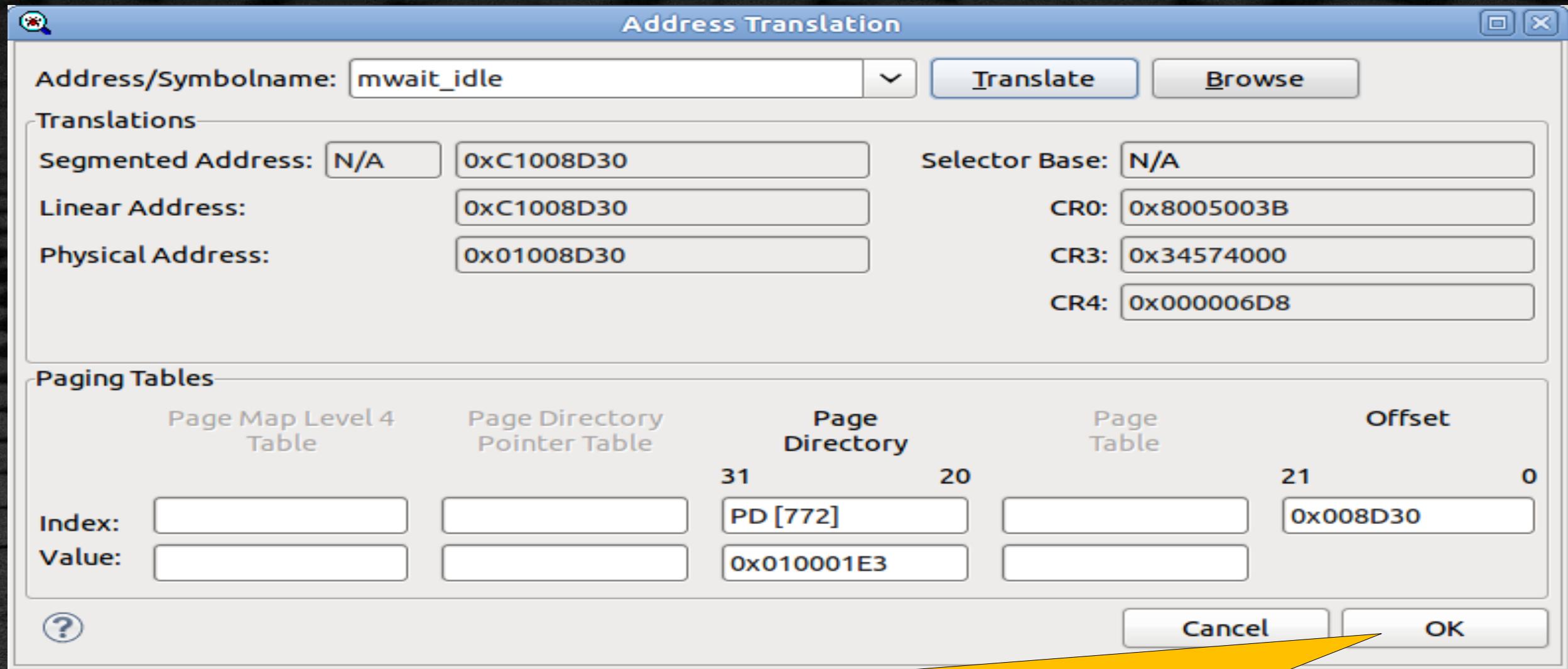
Register Layout:

Page is present in memory
Description:
Page-Table Attributes
Present Bit (P Bit)
The Present Bit indicates whether the page frame address in a page table entry maps to a page in physical memory. If the bit is set, the page is in memory.
Read/Write Bit (R/W Bit) and User/Supervisor Bit (U/S Bit)

Close Restore Set

Double click on a page table/directory entry and You will have the ability to modify the attribute bits with the bit field editor

Virtual -> physical address mapping



Select an address and press Translate to find the address mapping.
When you press OK the page directory/table involved will be shown in
the Paging window

Execution Trace

System Debugger supports execution trace via:

- Intel Processor Trace (Intel PT)
- Last-Branch Record (LBR)

Trace data is presented as C source code, view is integrated with other source debug features

User is not concerned with underlying mechanism, it “just works”



(R) Debugger

Trace Method: LBR

System Debugger Execution Trace Viewer

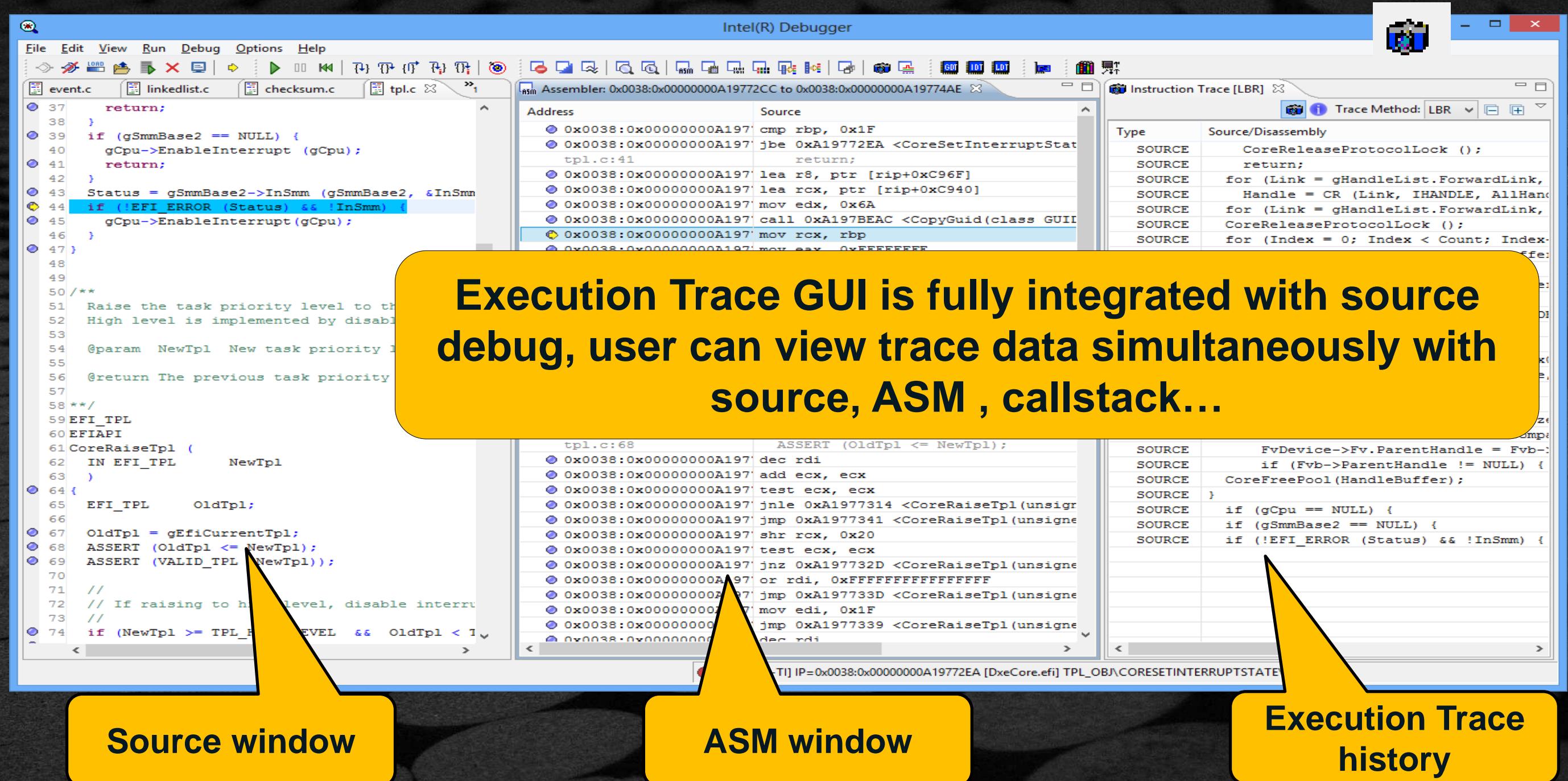
```

code           Source/Disassembly
CoreReleaseProtocolLock ();
return;
for (Link = gHandleList.ForwardLink, Count = 0; Link != &gHandleList;
Handle = CR (Link, IHANDLE, AllHandles, EFI_HANDLE_SIGNATURE);
for (Link = gHandleList.ForwardLink, Count = 0; Link != &gHandleList;
CoreReleaseProtocolLock ();
for (Index = 0; Index < Count; Index++) {
CoreConnectController (HandleBuffer[Index], NULL, NULL, TRUE);

CoreConnectController (HandleBuffer[Index], NULL, NULL, TRUE);
CoreFreePool (HandleBuffer[Index]);
if (FvDevice->Signature == FV2_DEVICE_SIGNATURE) {
    FvDevice->Fvb = Fvb;
} else {
    if ((Type & EVT_NOTIFY_SIGNAL) == EVT_NOTIFY_SIGNAL) {
        InsertHeadList (&gEventSignalQueue, &Event->SignalLink);
    CoreReleaseEventLock ();
} else {
    FvDevice = AllocateCopyPool (sizeof (FV_DEVICE), &mFvDevice);
    FvDevice->IsFfs3Fv = CompareGuid (&FwVolHeader->FileSignature,
    FvDevice->Fv.ParentHandle = Fvb->ParentHandle;
    if (Fvb->ParentHandle != NULL) {
        CoreFreePool (HandleBuffer);
    if (gCpu == NULL) {
    if (gSmmBase2 == NULL) {
        if (!EFI_ERROR (Status) && !InSmm) {
}
IP=0x0038:0x00000000A19772EA [DxeCore.efi] TPL_OBJ\CORESETINTERRUPTSTATE\@LINE 44

```

Execution Trace GUI Overview



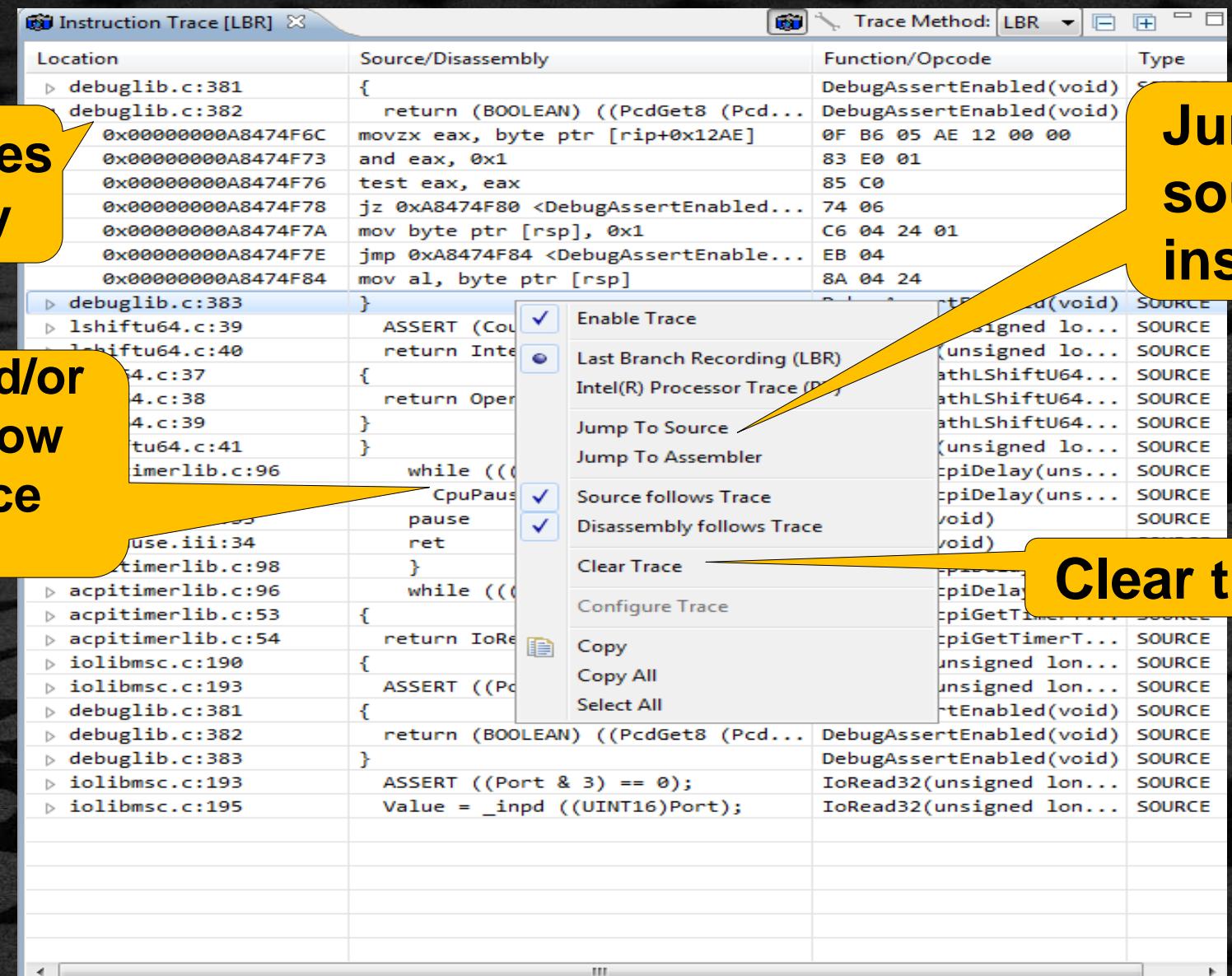
Execution Trace Features

Expand source lines
to reveal assembly

Keep source and/or
assembler window
in-sync with trace
window

Jump to this line's
source file or
instruction

Clear trace buffers



Execution Trace Features (continued)

Colors indicating how recently the instruction/line was executed

```
cpuio.c:340    mov rax, qword ptr [rsp...      48 8B 44 24 38
cpuio.c:341    cmp dword ptr [rsp+0x34...   83 7C 24 34 03
cpuio.c:342    jnz OxA7B91977                75 12
cpuio.c:343    MmioWrite64 ((UINTN...)        48 8B 44 24 38
cpuio.c:344    return EFI_SUCCESS;           E9 54 FF FF FF
cpuio.c:345    xor eax, eax                 48 83 C4 58
cpuio.c:346    C3
cpuio.c:347    CC
cpuio.c:348    /* Reads I/O registers.
cpuio.c:349
cpuio.c:350    The I/O operations are carried out exactly as requested. The
cpuio.c:351    for satisfying any alignment and I/O width restrictions that a
cpuio.c:352    platform might require. For example on some platforms, width is
cpuio.c:353    EfiCpuIoWidthUint64 do not work. Misaligned buffers, on the other
cpuio.c:354    be handled by the driver.
cpuio.c:355
cpuio.c:356    If Width is EfiCpuIoWidthUint8, EfiCpuIoWidthUint16, EfiCpuIo
cpuio.c:357    or EfiCpuIoWidthUint64, then both Address and Buffer are increased
cpuio.c:358    each of the Count operations that is performed.
cpuio.c:359
cpuio.c:360    If Width is EfiCpuIoWidthFifoUint8, EfiCpuIoWidthFifoUint16,
cpuio.c:361    EfiCpuIoWidthFifoUint32, or EfiCpuIoWidthFifoUint64, then only
cpuio.c:362    incremented for each of the Count operations that is performed
cpuio.c:363    write operation is performed Count times on the same Address.
cpuio.c:364
cpuio.c:365    If Width is EfiCpuIoWidthFillUint8, EfiCpuIoWidthFillUint16,
cpuio.c:366    EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64, then only
cpuio.c:367    incremented for each of the Count operations that is performed
cpuio.c:368    write operation is performed Count times from the first element.
```

```
329 // on the width of the transfer
330 InStride = mInStride[Width];
331 OutStride = mOutStride[Width];
332 OperationWidth = (EFI_CPU_IO_PROTOCOL_WIDTH) (Width & 0x03);
333 for (UInt8Buffer = Buffer; Count > 0; Address += InStride, Uint8Address += OutStride) {
334     if (OperationWidth == EfiCpuIoWidthUint8) {
335         MmioWrite8 ((UINTN)Address, *Uint8Buffer);
336     } else if (OperationWidth == EfiCpuIoWidthUint16) {
337         MmioWrite16 ((UINTN)Address, *((UINT16 *)Uint8Buffer));
338     } else if (OperationWidth == EfiCpuIoWidthUint32) {
339         MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffer));
340     } else if (OperationWidth == EfiCpuIoWidthUint64) {
341         MmioWrite64 ((UINTN)Address, *((UINT64 *)Uint8Buffer));
342     }
343 }
344 return EFI_SUCCESS;
345
346 /**
347 * Reads I/O registers.
348 */
349
350 The I/O operations are carried out exactly as requested. The
351 for satisfying any alignment and I/O width restrictions that a
352 platform might require. For example on some platforms, width is
353 EfiCpuIoWidthUint64 do not work. Misaligned buffers, on the other
354 be handled by the driver.
355
356 If Width is EfiCpuIoWidthUint8, EfiCpuIoWidthUint16, EfiCpuIo
357 or EfiCpuIoWidthUint64, then both Address and Buffer are increased
358 each of the Count operations that is performed.
359
360 If Width is EfiCpuIoWidthFifoUint8, EfiCpuIoWidthFifoUint16,
361 EfiCpuIoWidthFifoUint32, or EfiCpuIoWidthFifoUint64, then only
362 incremented for each of the Count operations that is performed
363 write operation is performed Count times on the same Address.
364
365 If Width is EfiCpuIoWidthFillUint8, EfiCpuIoWidthFillUint16,
366 EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64, then only
367 incremented for each of the Count operations that is performed
368 write operation is performed Count times from the first element.
```

```
329 // on the width of the transfer
330 InStride = mInStride[Width];
331 OutStride = mOutStride[Width];
332 OperationWidth = (EFI_CPU_IO_PROTOCOL_WIDTH) (Width & 0x03);
333 for (UInt8Buffer = Buffer; Count > 0; Address += InStride, Uint8Address += OutStride) {
334     if (OperationWidth == EfiCpuIoWidthUint8) {
335         MmioWrite8 ((UINTN)Address, *Uint8Buffer);
336     } else if (OperationWidth == EfiCpuIoWidthUint16) {
337         MmioWrite16 ((UINTN)Address, *((UINT16 *)Uint8Buffer));
338     } else if (OperationWidth == EfiCpuIoWidthUint32) {
339         MmioWrite32 ((UINTN)Address, *((UINT32 *)Uint8Buffer));
340     } else if (OperationWidth == EfiCpuIoWidthUint64) {
341         MmioWrite64 ((UINTN)Address, *((UINT64 *)Uint8Buffer));
342     }
343 }
344 return BOOLEAN ((PcdGet8 (PcdDebugEnabledPropertyMask) & DEB);
345
346 /**
347 * Reads I/O registers.
348 */
349
350 The I/O operations are carried out exactly as requested. The
351 for satisfying any alignment and I/O width restrictions that a
352 platform might require. For example on some platforms, width is
353 EfiCpuIoWidthUint64 do not work. Misaligned buffers, on the other
354 be handled by the driver.
355
356 If Width is EfiCpuIoWidthUint8, EfiCpuIoWidthUint16, EfiCpuIo
357 or EfiCpuIoWidthUint64, then both Address and Buffer are increased
358 each of the Count operations that is performed.
359
360 If Width is EfiCpuIoWidthFifoUint8, EfiCpuIoWidthFifoUint16,
361 EfiCpuIoWidthFifoUint32, or EfiCpuIoWidthFifoUint64, then only
362 incremented for each of the Count operations that is performed
363 write operation is performed Count times on the same Address.
364
365 If Width is EfiCpuIoWidthFillUint8, EfiCpuIoWidthFillUint16,
366 EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64, then only
367 incremented for each of the Count operations that is performed
368 write operation is performed Count times from the first element.
```

“+” indicates multiple hits

```
Console View > Debugger Commands
xdb> efi loadthis
INFO: Searching backwards from 0x00000000A7B9229D to 0x00000000A7A9229D for PE/COFF
header (semantics=MEM align=0x00001000 range=0x00100000)
INFO: Found PE/COFF module at 0x00000000A7B91000 - 0x00000000A7B94FA0 Entrypoint:
0x00000000A7B912FC (size: 16288 bytes)
INFO: Successfully loaded debug symbols found at offset 0x00000000A7B91000:
C:\Users\wsvregr\BDW070\Sym\Build\BroadwellPlatPkg\DEBUG_VS2012x86\X64\IntelFrameworkModulePkg\Universal\CpuIoDxe\CpuIoDxe\DEBUG\CPuIoDxe.efi
itrace: PT configuration: Trace enabled
WARNING: Multiple breaks, context is set to the most interesting.
execution stopped by "running"
xdb>
```

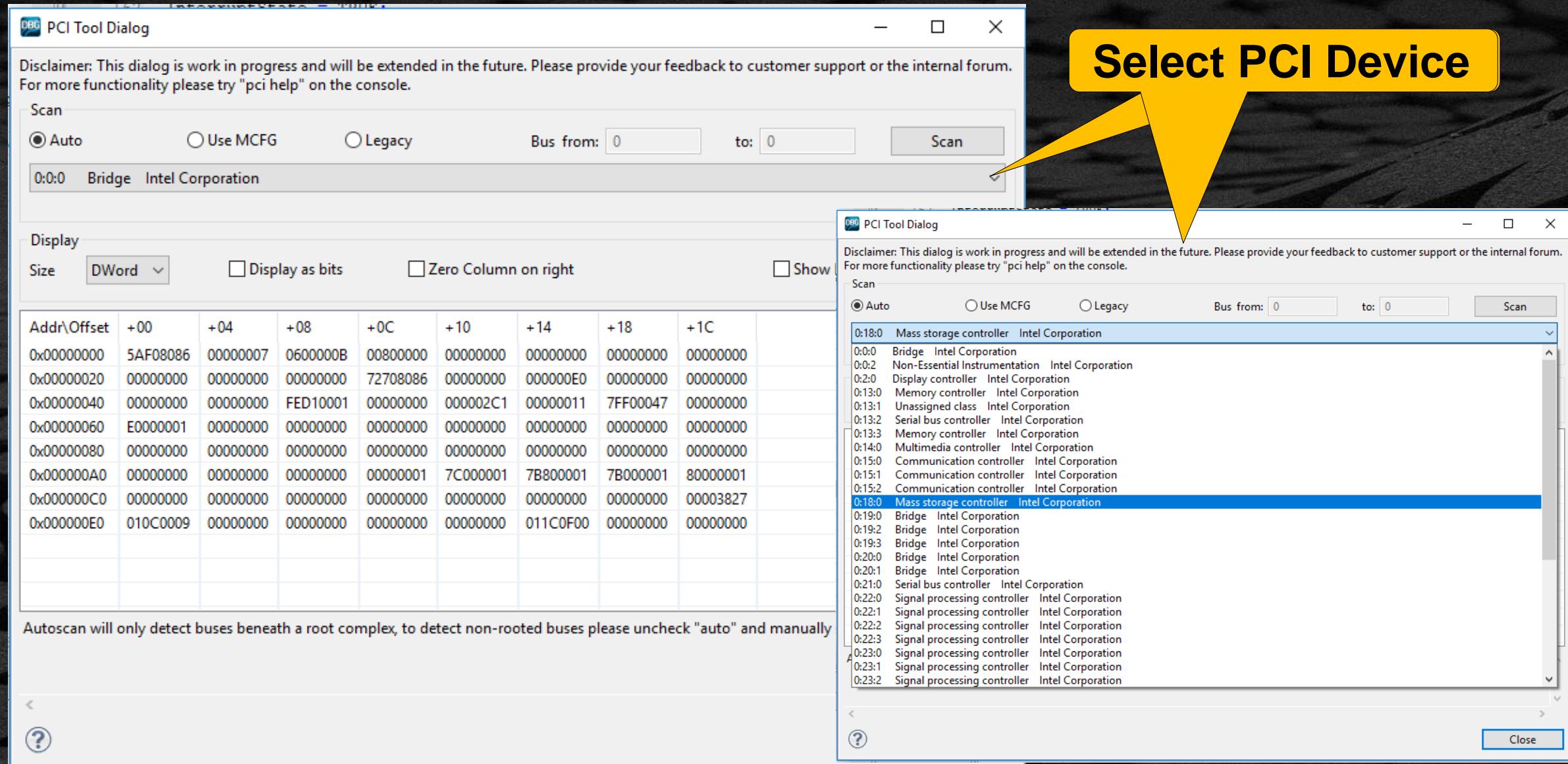
Register	Value	Description
EDI	0x97E88018	
ESP	0xA5B280E0	
EBP	0x0000000000000007	
CS	0x0038	
DS	0x0018	
SS	0x0018	
ES	0x0018	
FS	0x0018	
GS	0x0018	
EIP	0xA7B9229D	MmioWrite32(unsigned long lon...
EFL	0x00010246	EFLAGS Register

Intuitive following of execution flow

Coloring can be configured in Options -> GUI Preferences

PCI Scan Tool Technology Preview

Scan for devices, display device-specific registers:



[U]EFI Commands

Use “efi help” at the Console command line.

> efi help

EFI Debugger extension

Enter any of the following on the command line:

EFI "SHOWCONFIG"

Displays the debugger configuration.

EFI "SETSUFFIX %s"

Set the suffix for EFI debug information files. This setting controls the expected debug info format (Visual Studio/GCC). Accepted values are (efi|debug).

EFI "SHOWSUFFIX"

Show the active debug information suffix.

EFI "SETSWMODE %s"

Set the software execution mode of the debugger.
Accepted values are (auto|efi32|efi64).

EFI "SHOWSYSTAB [FORCE=%s(OFF|ON)]"

Print out information parsed from DXE system table.

If system table address is not set or the force flag is ON, memory is scanned for it. Command can be used in DXE boot phase.

EFI "SHOWDXEMODULES"

Print out the list of DXE EFI modules loaded to memory from the DXE system table. Command can be used in DXE boot phase.

EFI "LOADFROMMAP %s"

Load debug information for modules from the given EFI flash map file. Map file contains modules in flash (SEC and PEI phases).

-
-
-

Command “efi showsysstab”

> efi showsystab

INFO: Using cached EFI State Information
 EFI System table at 0x000000007FED018

Configuration Tables:

GUID:	Pointer:	Name:
GUID fc1bcdb0, 7d31, 49aa, {...}	0x77dadb98	
GUID ee4e5898, 3914, 4259, {...}	0x77dac718	
GUID 05ad34ba, 6f02, 4214, {...}	0x7a1e1fc0	DXE_SERVICES_TABLE
GUID 7739f24c, 93d7, 11d4, {...}	0x77d8c018	HOB_LIST
GUID 4c19049f, 4137, 4dd3, {...}	0x7a1e2830	
GUID 49152e77, 1ada, 4764, {...}	0x7a1e31e0	EFI_DEBUG_IMAGE_INFO_TABLE
GUID dba6a7e3, bb57, 4be7, {...}	0x77856998	
GUID 4e28ca50, d582, 44ac, {...}	0x79b5c018	

BootServices:

EFI_RAISE_TPL	0x000000007A1D2D4C
EFI_RESTORE_TPL	0x000000007A1D2DE8
EFI_ALLOCATE_PAGES	0x000000007A1CAD54
EFI_FREE_PAGES	0x000000007A1CAE08
EFI_GET_MEMORY_MAP	0x000000007A1CAF0B
EFI_ALLOCATE_POOL	0x000000007A1D14EC
EFI_FREE_POOL	0x000000007A1D19A8
EFI_CREATE_EVENT	0x000000007A1CD884
EFI_SET_TIMER	0x000000007A1D1218

EFI_WAIT_FOR_EVENT	0x000000007A1CDBFC
EFI_SIGNAL_EVENT	0x000000007A1CDCAC8
EFI_CLOSE_EVENT	0x000000007A1CDC9C
EFI_CHECK_EVENT	0x000000007A1CDB54
EFI_INSTALL_PROTOCOL_INTERFACE	0x000000007A1C7F88
EFI_REINSTALL_PROTOCOL_INTERFACE	0x000000007A1CD644
EFI_UNINSTALL_PROTOCOL_INTERFACE	0x000000007A1C8450
EFI_HANDLE_PROTOCOL	0x000000007A1C8680
EFI_REGISTER_PROTOCOL_NOTIFY	0x000000007A1CD568
EFI_LOCATE_HANDLE	0x000000007A1D009C
EFI_INSTALL_CONFIGURATION_TABLE	0x000000007A1D3E10
EFI_IMAGE_LOAD	0x000000007A1CC770
EFI_IMAGE_START	0x000000007A1CC80C
EFI_EXIT	0x000000007A1CCA8
EFI_IMAGE_UNLOAD	0x000000007A1CCBD8
EFI_EXIT_BOOT_SERVICES	0x000000007A1C5158
EFI_GET_NEXT_MONOTONIC_COUNT	0x000000007A1C5118
EFI_STALL	0x000000007A1D3378
EFI_SET_WATCHDOG_TIMER	0x000000007A1CED78
EFI_CONNECT_CONTROLLER	0x000000007A1C8E7C
EFI_DISCONNECT_CONTROLLER	0x000000007A1C9744
EFI_OPEN_PROTOCOL	0x000000007A1C86A4
EFI_CLOSE_PROTOCOL	0x000000007A1C89B4
EFI_OPEN_PROTOCOL_INFORMATION	0x000000007A1C8ADC
EFI_PROTOCOLS_PER_HANDLE	0x000000007A1C8C34
EFI_LOCATE_HANDLE_BUFFER	0x000000007A1D06F4
EFI_LOCATE_PROTOCOL	0x000000007A1D05F4
...	

UEFI DEBUG AGENT

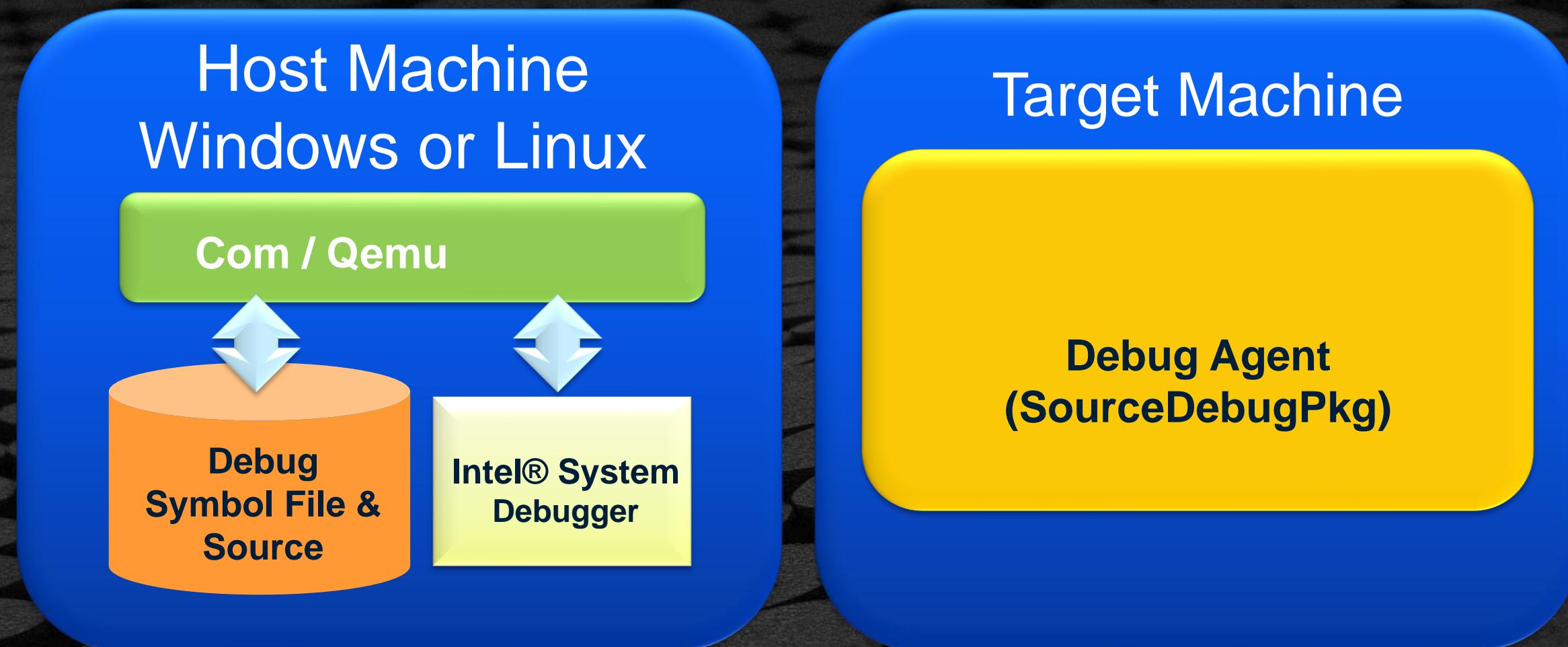
Configure Intel® System Studio Debugger

UEFI BIOS Support – Active Mode

Active Mode (agent-based):

- Receive notifications from agent as modules are loaded/unloaded
- Break at init of a named module, regardless of load position
- Requires debug agent “SourceLevelDebugPkg” in Github [EDK II](#)
- Uses Serial Port to connect to Target – no JTAG or DCI

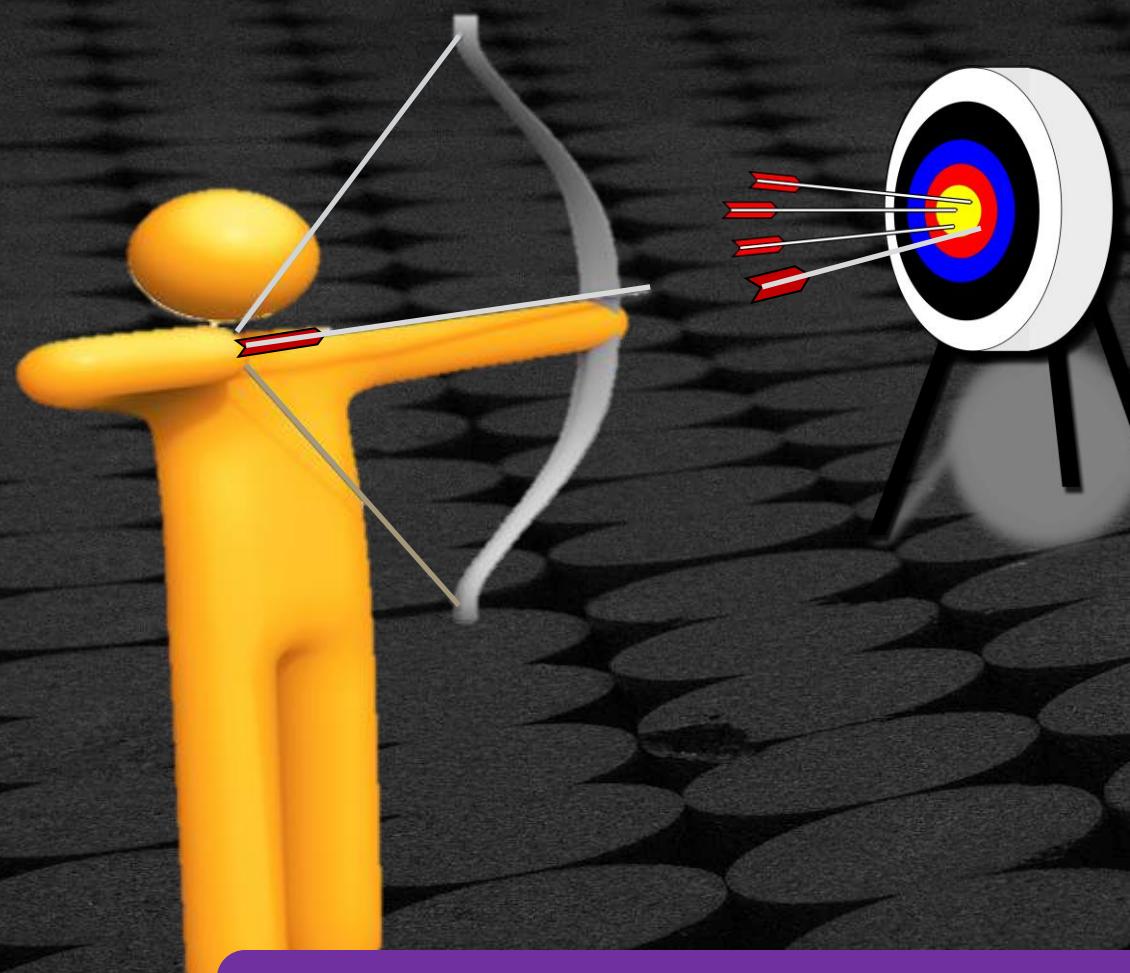
EDK II SourceLevelDebugPkg



Source Level Debugger for UEFI

Changes to Target Firmware

Set “`DEFINE SOURCE_DEBUG_ENABLE = TRUE`” in DSC file



Add call to new library class
(DebugAgentLib) In SEC, DXE Main,
and SMM CPU Mod.

Or if you don't want to add one
A **NULL** implementation of
DebugAgentLib is checked into
open source

Goal: Minimize changes to target firmware

Updates to DSC

Libraries

[LibraryClasses] **General**

PeCoffExtraActionLib

DebugCommunicationLib

[LibraryClasses.IA32] **SEC / PEI**

DebugAgentLib

[LibraryClasses.X64] **DXE**

DebugAgentLib

[LibraryClasses.X64.DXE_SMM_DRIVER] **SMM**

DebugAgentLib

SourceLevelDebugPkg Lib Instance

COM1
or
USB

- PeCoffExtraActionLibDebug.inf
- DebugCommunicationLibSerialPort.inf
- or
- DebugCommunicationLibUsb.inf

➤ SecPeiDebugAgentLib.inf

➤ DxeDebugAgentLib.inf

➤ SmmDebugAgentLib.inf

Updates to DSC for USB 3.0

Libraries

[LibraryClasses] **General**

PeCoffExtraActionLib

[LibraryClasses.IA32] **SEC/PEI**

DebugCommunicationLib

DebugAgentLib

[LibraryClasses.X64] **DXE**

DebugCommunicationLib

DebugAgentLib

[LibraryClasses.X64.DXE_SMM_DRIVER] **SMM**

DebugCommunicationLib

DebugAgentLib

SourceLevelDebugPkg Lib Instance

➤ PeCoffExtraActionLibDebug.inf

➤ DebugCommunicationLibUsb3Pei.inf

➤ SecPeiDebugAgentLib.inf

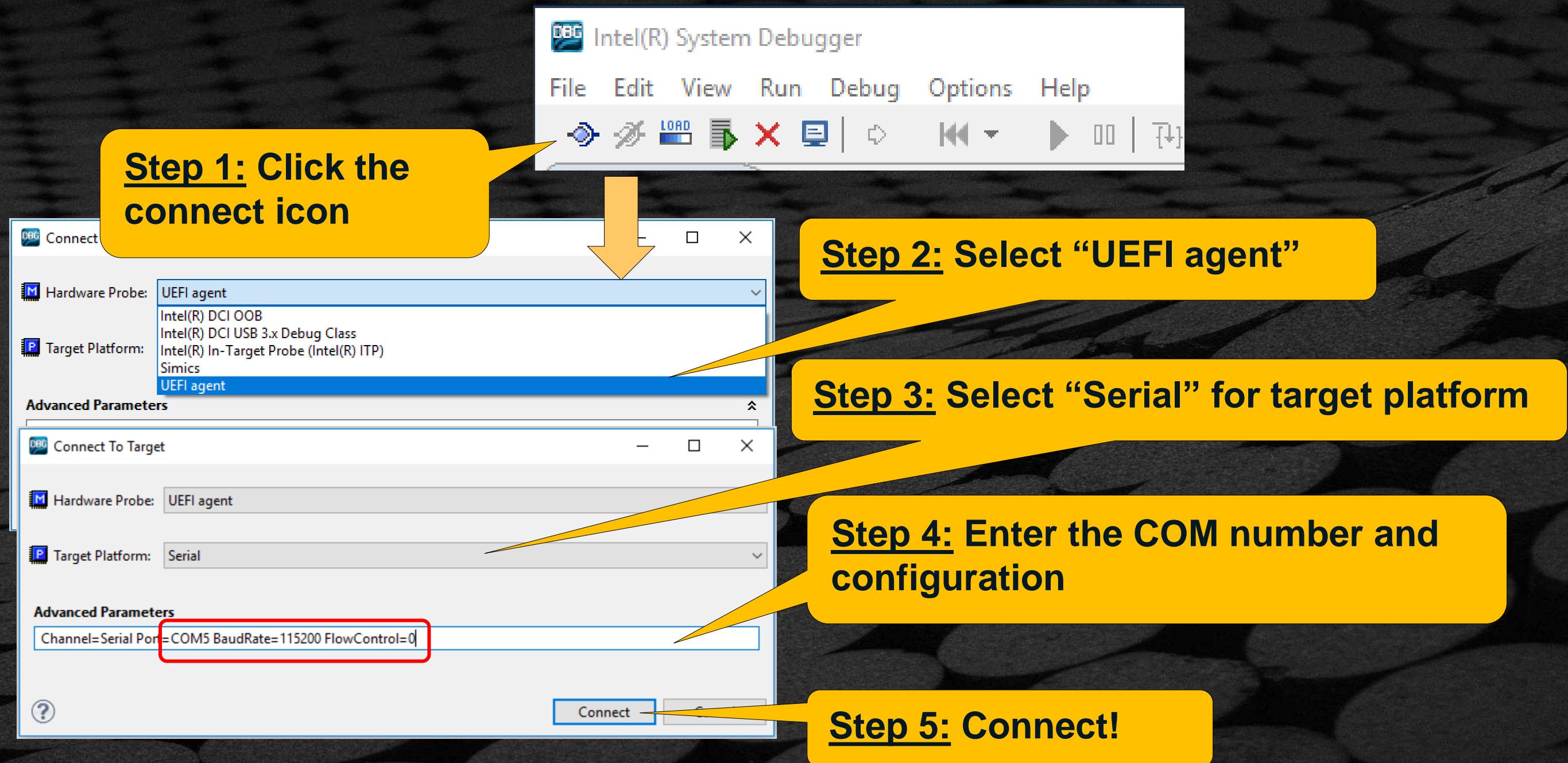
➤ DebugCommunicationLibUsb3Dxe.inf

➤ DxeDebugAgentLib.inf

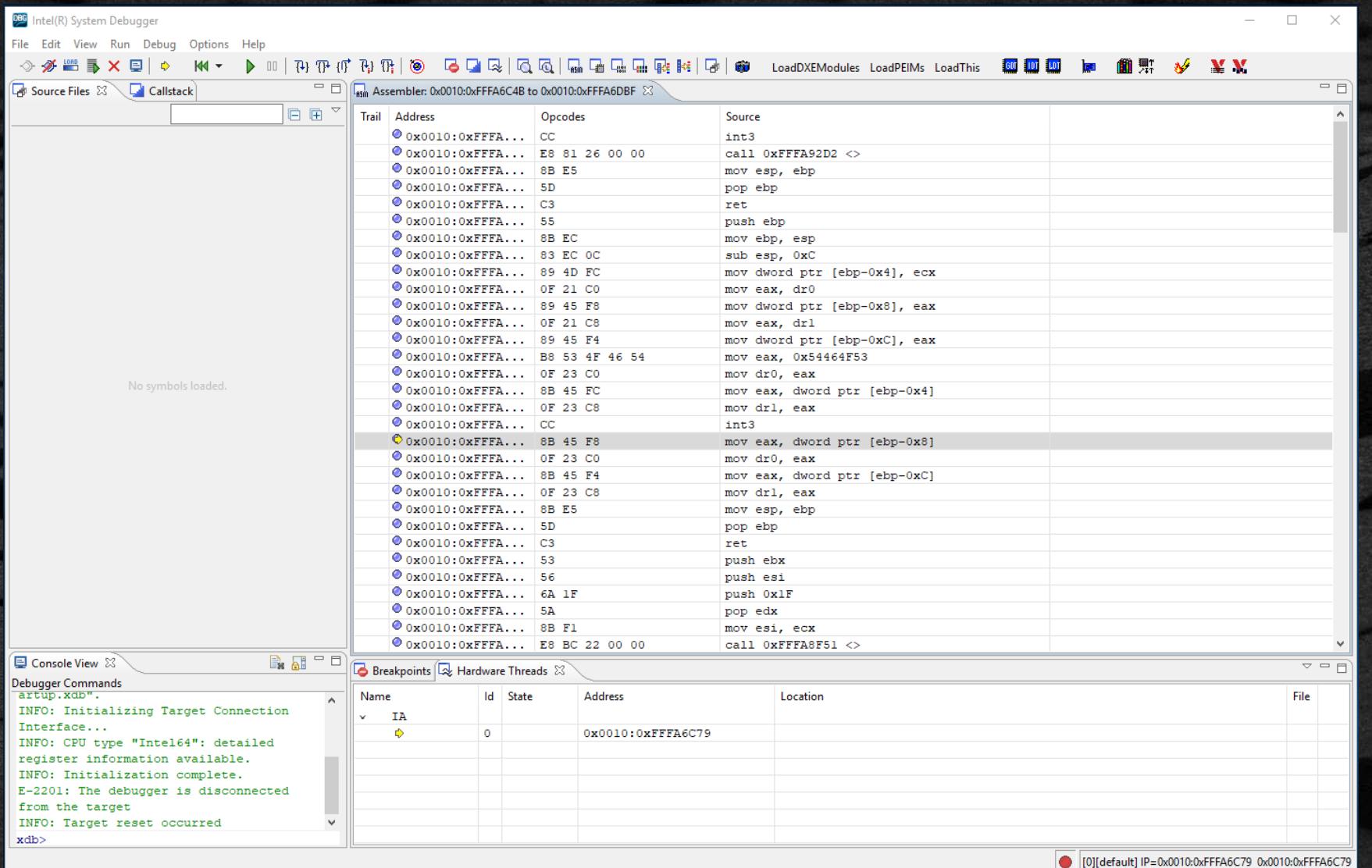
➤ DebugCommunicationLibUsb3Dxe.inf

➤ SmmDebugAgentLib.inf

Connecting To Targets

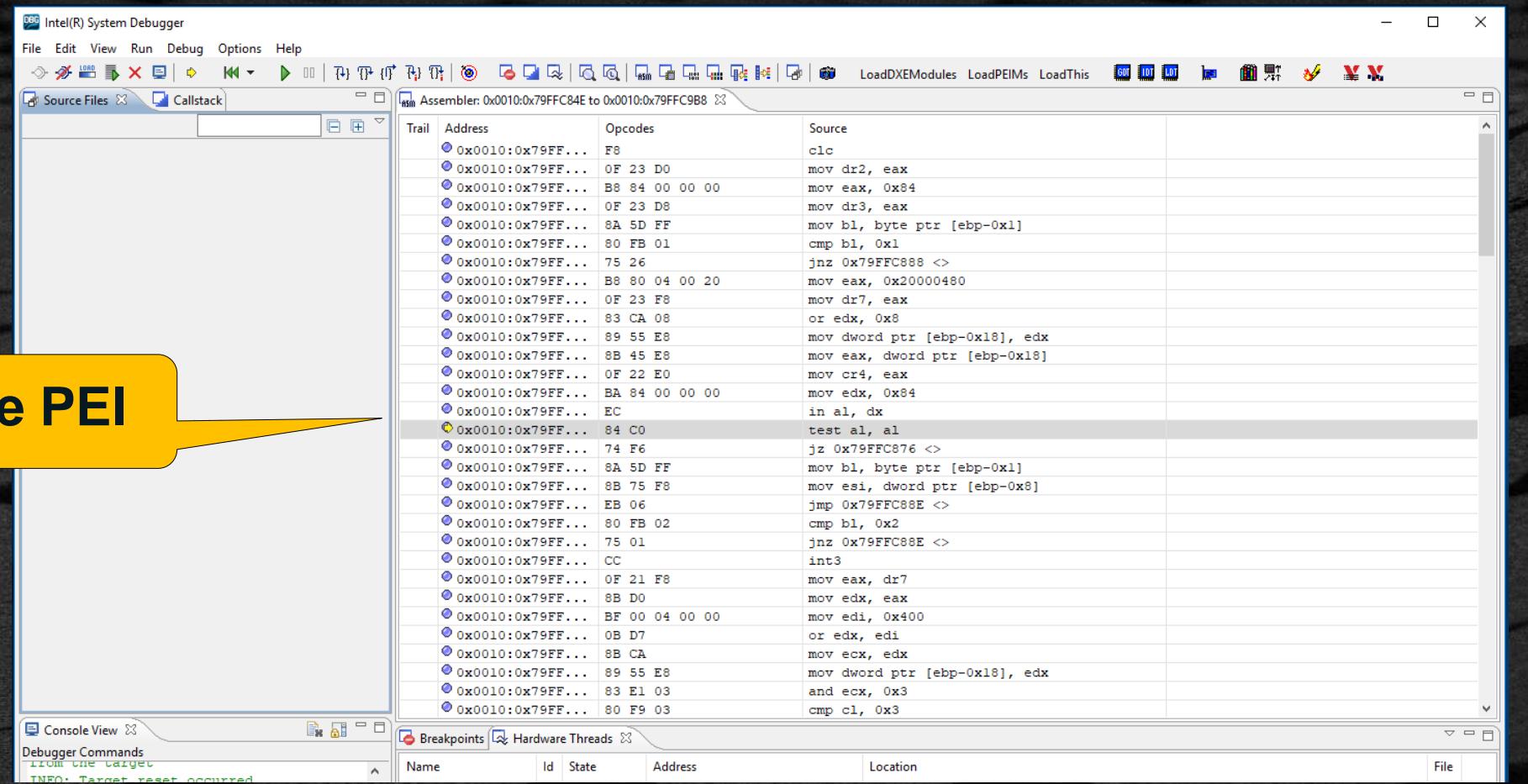


At Power on the target is in the Debug Agent code



Debugging UEFI

Select run “F5” ► then halt “pause” || quickly



Halt inside PEI

Intel(R) System Debugger

File Edit View Run Debug Options Help

Source Files Callstack

Assembler: 0x0010:0x79FFC84E to 0x0010:0x79FFC9B8

Trail	Address	Opcodes	Source
0x0010:0x79FF...	F8	clc	
0x0010:0x79FF...	0F 23 D0	mov dr2, eax	
0x0010:0x79FF...	B8 84 00 00 00	mov eax, 0x84	
0x0010:0x79FF...	0F 23 D8	mov dr3, eax	
0x0010:0x79FF...	8A 5D FF	mov bl, byte ptr [ebp-0x1]	
0x0010:0x79FF...	80 FB 01	cmp bl, 0x1	
0x0010:0x79FF...	75 26	jnz 0x79FFC888 <>	
0x0010:0x79FF...	B8 80 04 00 20	mov eax, 0x20000480	
0x0010:0x79FF...	0F 23 F8	mov dr7, eax	
0x0010:0x79FF...	83 CA 08	or edx, 0x8	
0x0010:0x79FF...	89 55 E8	mov dword ptr [ebp-0x18], edx	
0x0010:0x79FF...	8B 45 E8	mov eax, dword ptr [ebp-0x18]	
0x0010:0x79FF...	0F 22 E0	mov cr4, eax	
0x0010:0x79FF...	BA 84 00 00 00	mov edx, 0x84	
0x0010:0x79FF...	EC	in al, dx	
0x0010:0x79FF...	84 C0	test al, al	
0x0010:0x79FF...	74 F6	jz 0x79FFC876 <>	
0x0010:0x79FF...	8A 5D FF	mov bl, byte ptr [ebp-0x1]	
0x0010:0x79FF...	8B 75 F8	mov esi, dword ptr [ebp-0x8]	
0x0010:0x79FF...	EB 06	jmp 0x79FFC88E <>	
0x0010:0x79FF...	80 FB 02	cmp bl, 0x2	
0x0010:0x79FF...	75 01	jnz 0x79FFC88E <>	
0x0010:0x79FF...	CC	int3	
0x0010:0x79FF...	0F 21 F8	mov eax, dr7	
0x0010:0x79FF...	8B D0	mov edx, eax	
0x0010:0x79FF...	BF 00 04 00 00	mov edi, 0x400	
0x0010:0x79FF...	0B D7	or edx, edi	
0x0010:0x79FF...	8B CA	mov ecx, edx	
0x0010:0x79FF...	89 55 E8	mov dword ptr [ebp-0x18], edx	
0x0010:0x79FF...	83 E1 03	and ecx, 0x3	
0x0010:0x79FF...	80 F9 03	cmp cl, 0x3	

Console View

Debugger Commands

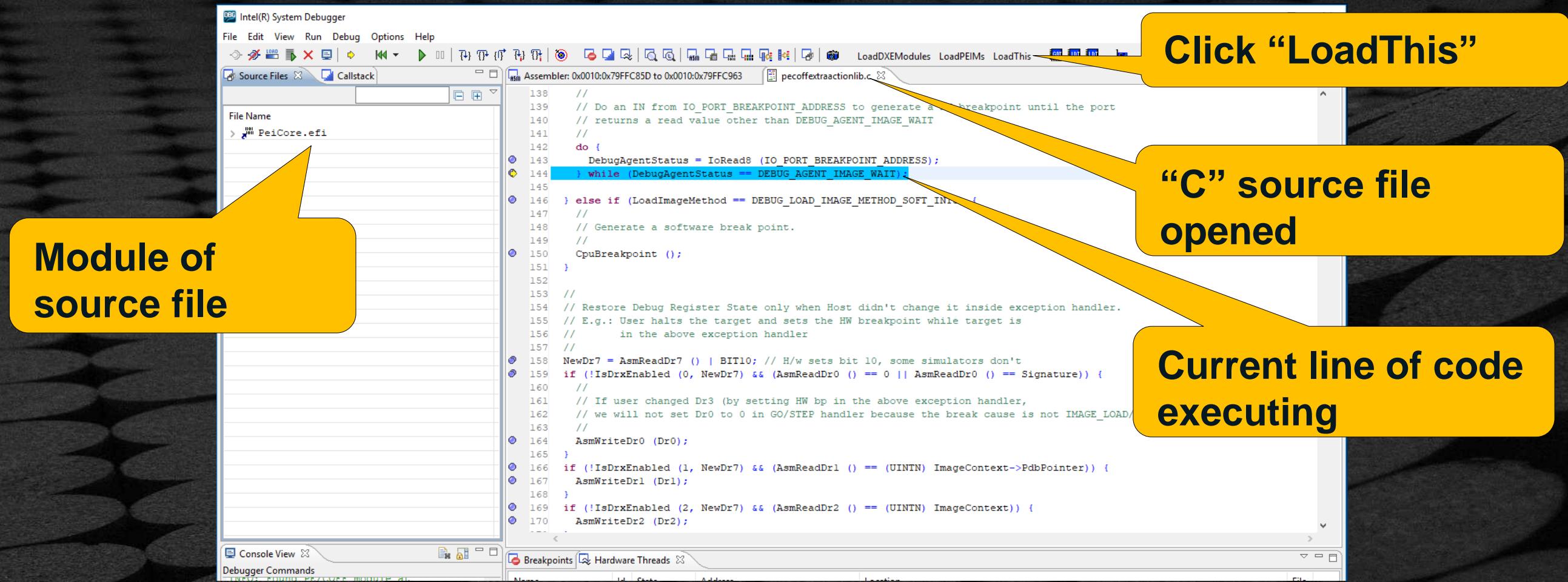
INFO: Target reset occurred

Breakpoints

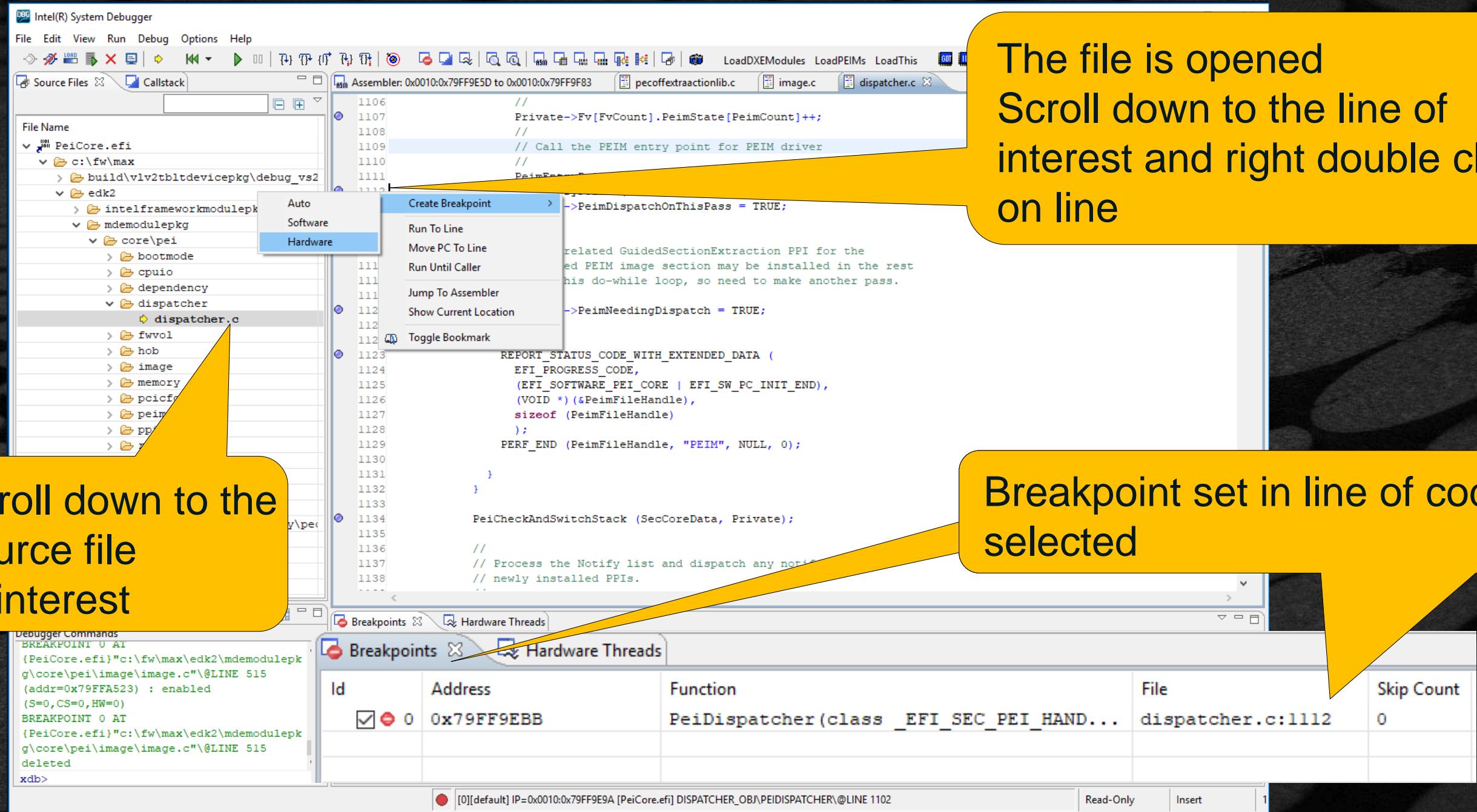
Hardware Threads

Debugging UEFI

Use System Debugger function “LoadThis” to load symbols



Breakpoint in PEI



The file is opened
Scroll down to the line of
interest and right double click
on line

Scroll down to the
source file
of interest

Breakpoint set in line of code
selected

BREAKPOINTS

Add Breakpoint to the Compiled BIOS / Firmware Source Code

EDK II Breakpoints w/ System Debugger

CpuBreakpoint

When using a UEFI agent connection:

- Debug agent – SourceLevelDebugPkg

CpuDeadLoop

When using a Hardware debugger:

- In-Target Probe (ITP)
- Intel® SVT DCI cable
- Intel® SVT Closed Chassis Adapter (CCA)
- other 3rd Party Hardware (i.e. Lauterbach w/ JTAG)

The functions `CpuBreakpoint()` and `CpuDeadLoop()` are part of the EDK II Base Libraries and can be compiled with any UEFI or PI Module at any phase of the boot flow (SEC, PEI, DXE, BDS, TSL)

Special DCI Breakpoint

CpuIceBreakpoint

The Intel Architecture has a special op-code for a breakpoint: int1
Better than a CpuDeadLoop() since it halts the processor. Better trace information

Downside, there is no “C” equivalent – needs to be assembly code

CpuIceBreakpoint.nasm



Needs to be in /ia32 & /X64 directory

```
DEFAULT REL
SECTION .text
global ASM_PFX(CpuIceBreakpoint)
ASM_PFX(CpuIceBreakpoint):
    int1
    ret
```

What is a ICEBP

Intel® Processor
PMCR

An undocumented
op code

Makes debugging
easier

- Interrupt Redirection is enabled in the Probe Mode Control Register **PMCR**
- not software accessible
- ICE must be connected to modify PMCR
- to figure out where your program was loaded in memory since it halts the processor

Compiling for CpuIceBreakpoint

“C” File

```
extern VOID CpuIceBreakpoint();  
  
VOID  
EFIAPI  
CpuIceBreakpoint();  
. . .  
  
UefiMain(  
{  
. . .  
CpuIceBreakpoint();  
. . .  
}
```

INF File

```
. . .  
[Sources]  
    SampleApp.c  
[Sources.Ia32]  
    Ia32/CpuIceBreakpoint.nasm  
[Sources.X64]  
    X64/CpuIceBreakpoint.nasm
```

See example: [link](#)

Enable INT1 Exception handler

How to enable INT1 Exception handling in Intel System Debugger to trap the CPU and get the control for source debugging. Target Communication Framework (TCF)

1. On "Ipccli"
2. In System Debugger TCF debugger's command line console

Enable INT1 Exception handler- “Ipcccli”

On "ipcccli", after halt the process then issue "itp.breaks.int1exception=1" which enable int1 trap now.

The screenshot shows an IPython terminal window titled "IPython: C:\tools\OpenIPC_1.1917.3733.200". The terminal displays the following session:

```
In [5]: itp.status()
Status for      : GLM_C0_T0
Processor      : Halted
Processor mode : Compatibility, Real
Status for      : GLM_C2_T0
Processor      : Halted
Processor mode : Compatibility, Real

In [6]: itp.breaks.int1exception
Out[6]: 0L

In [7]: itp.breaks.int1exception=1

In [8]: itp.go()

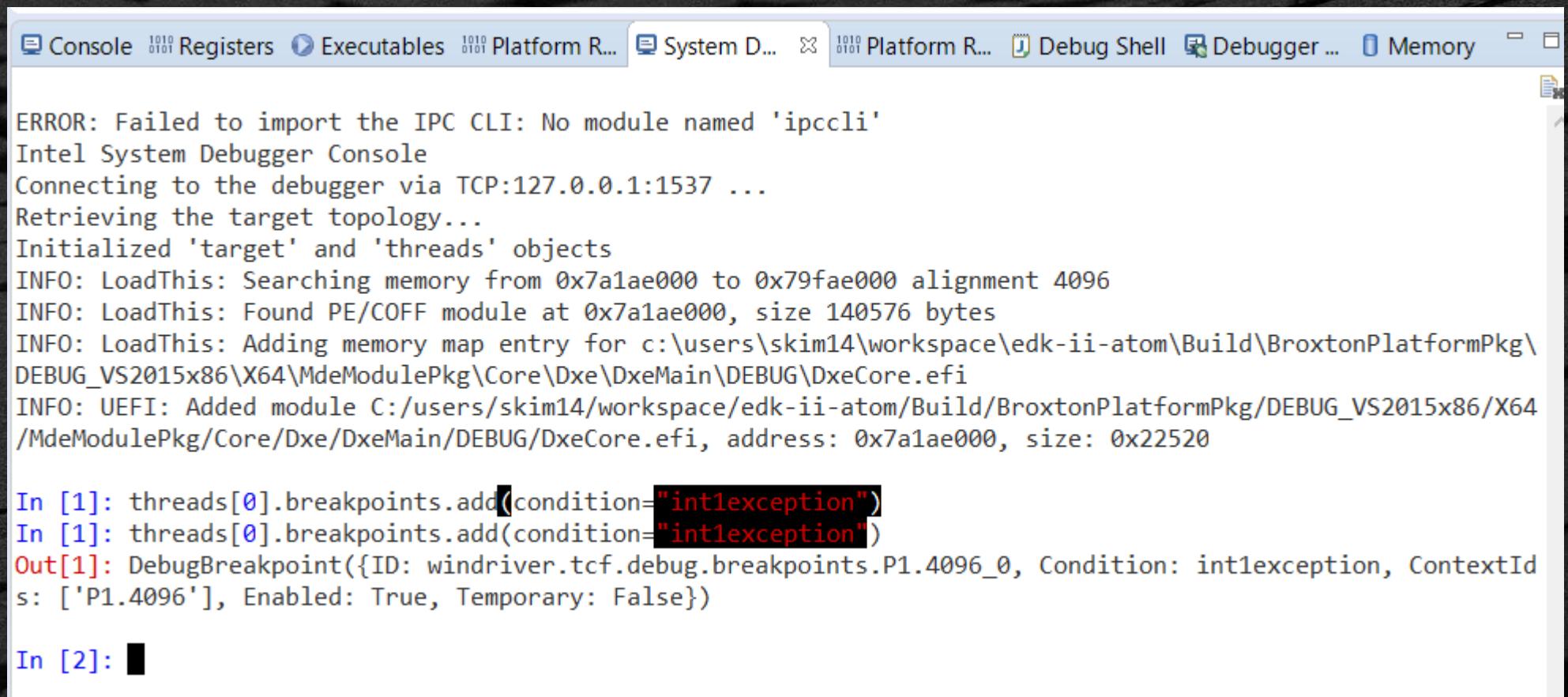
[GLM_C0_T0] Resuming -- 10:58:37.695000 2019-05-24
[GLM_C2_T0] Resuming -- 10:58:37.696000 2019-05-24

[GLM_C0_T0] Debug trap Break at [0x38:000000007a1ae2c1] -- 10:59:03.013000 2019-05-24
[GLM_C2_T0] Wait for SIPI loop Break at [0xf000:ffffffffff00010000] -- 10:59:03.015000 2019-05-24

In [9]:
```

Enable INT1 Exception handler- Console

On "System Debugger Console" of TCF debugger, issue the command - "threads[0].breakpoints.add(condition="int1exception") ", then it will enable INT1Exception trap from now



The screenshot shows the Intel System Debugger Console interface. The top navigation bar includes tabs for Console, Registers, Executables, Platform R..., System D..., Platform R..., Debug Shell, Debugger ..., and Memory. The main window displays a series of log messages and command history:

```
ERROR: Failed to import the IPC CLI: No module named 'ipccli'
Intel System Debugger Console
Connecting to the debugger via TCP:127.0.0.1:1537 ...
Retrieving the target topology...
Initialized 'target' and 'threads' objects
INFO: LoadThis: Searching memory from 0x7a1ae000 to 0x79fae000 alignment 4096
INFO: LoadThis: Found PE/COFF module at 0x7a1ae000, size 140576 bytes
INFO: LoadThis: Adding memory map entry for c:\users\skim14\workspace\edk-ii-atom\Build\BroxtonPlatformPkg\DEBUG_VS2015x86\X64\MdeModulePkg\Core\Dxe\DXeMain\DEBUG\DXeCore.efi
INFO: UEFI: Added module C:/users/skim14/workspace/edk-ii-atom/Build/BroxtonPlatformPkg/DEBUG_VS2015x86/X64/MdeModulePkg/Core/Dxe/DxeMain/DEBUG/DxeCore.efi, address: 0x7a1ae000, size: 0x22520

In [1]: threads[0].breakpoints.add(condition="int1exception")
In [1]: threads[0].breakpoints.add(condition="int1exception")
Out[1]: DebugBreakpoint({ID: windriver.tcf.debug.breakpoints.P1.4096_0, Condition: int1exception, ContextIds: ['P1.4096'], Enabled: True, Temporary: False})

In [2]:
```

UEFI Application w/ CpuIceBreakpoint

Example SampleAppSS.efi with int1

Stops at int1 ASM code

click "LoadThis"

Continue stepping into your code, "F7"

File Edit Setup Control Window Help
F50:\up2\Debug\> SampleAppSS.efi
 InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 76B09040
 Loading driver at 0x00026AE8000 EntryPoint=0x00026AE8394 SampleAppSS.efi
 InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 76B05018
 ProtectUefiImageCommon - 0x76B09040

workspace-new_nda-07 - Disassembly - Intel System Debugger 2019 NDA U1921

File Edit Source Refactor Navigate Project Run Tools Window Help

DBG Broxton P

Debug Project Explorer

New_configuration (Intel System Del)

BXTP_CLTAPCO

GLM_C0_T0 (Step Over)

0x000000076ae82c3: nop
int1
 ret

0x000000076ae82c4: int3
 0x000000076ae82c5: int3
 0x000000076ae82c6: int3
 0x000000076ae82c7: int3
 0x000000076ae82c8: int3
 0x000000076ae82c9: int3
 0x000000076ae82ca: int3
 0x000000076ae82cb: int3
 0x000000076ae82cc: int3
 0x000000076ae82cd: int3
 0x000000076ae82ce: int3
 0x000000076ae82cf: int3
 0x000000076ae82d0: push rdi
 0x000000076ae82d1: push rcx
 0x000000076ae82d2: xor rax, rax
 0x000000076ae82d5: mov rdi, rcx
 0x000000076ae82d8: mov rcx, rdx
 0x000000076ae82db: shr rcx, 0x3
 0x000000076ae82df: and rdx, 0x7
 0x000000076ae82e3: rep stosq qword
 0x000000076ae82e6: mov ecx, edx

sampleapp.c

```

DEBUG((EFI_D_INFO, "\r\n>>>>[UefiMain] End\r\n"));
DEBUG ((0xffffffff, "\n\nUEFI Base Training DEBUG DEMO\n"));
DEBUG ((0xffffffff, "0xffffffff USING DEBUG ALL Mask Bits Set\r\n"));
//ASSERT_EFI_ERROR(0x80000000);
//CpuBreakpoint();

CpuIceBreakpoint();

DEBUG((DEBUG_INIT, " 0x%08x USING DEBUG DEBUG_INIT\n", (UINTN)(DEBUG_INIT)));
DEBUG((DEBUG_FS, " 0x%08x USING DEBUG DEBUG_FS\n", (UINTN)(DEBUG_FS)));
DEBUG((DEBUG_POOL, " 0x%08x USING DEBUG DEBUG_POOL\n", (UINTN)(DEBUG_POOL)));
DEBUG((DEBUG_PAGE, " 0x%08x USING DEBUG DEBUG_PAGE\n", (UINTN)(DEBUG_PAGE)));
DEBUG((DEBUG_INFO, " 0x%08x USING DEBUG DEBUG_INFO\n", (UINTN)(DEBUG_INFO)));
DEBUG((DEBUG_DISPATCH, " 0x%08x USING DEBUG DEBUG_DISPATCH\n", (UINTN)(DEBUG_DISPATCH)));
DEBUG((DEBUG_VARIABLE, " 0x%08x USING DEBUG DEBUG_VARIABLE\n", (UINTN)(DEBUG_VARIABLE)));
DEBUG((DEBUG_BM, " 0x%08x USING DEBUG DEBUG_BM\n", (UINTN)(DEBUG_BM)));
DEBUG((DEBUG_BLKIO, " 0x%08x USING DEBUG DEBUG_BLKIO\n", (UINTN)(DEBUG_BLKIO)));
DEBUG((DEBUG_NET, " 0x%08x USING DEBUG DEBUG_NET\n", (UINTN)(DEBUG_NET)));
DEBUG((DEBUG_UNDI, " 0x%08x USING DEBUG DEBUG_UNDI\n", (UINTN)(DEBUG_UNDI)));
DEBUG((DEBUG_LOADFILE, " 0x%08x USING DEBUG DEBUG_LOADFILE\n", (UINTN)(DEBUG_LOADFILE)));
DEBUG((DEBUG_EVENT, " 0x%08x USING DEBUG DEBUG_EVENT\n", (UINTN)(DEBUG_EVENT)));
DEBUG((DEBUG_GCD, " 0x%08x USING DEBUG DEBUG_GCD\n", (UINTN)(DEBUG_GCD)));
DEBUG((DEBUG_CACHE, " 0x%08x USING DEBUG DEBUG_CACHE\n", (UINTN)(DEBUG_CACHE)));
DEBUG((DEBUG_VERBOSE, " 0x%08x USING DEBUG DEBUG_VERBOSE\n", (UINTN)(DEBUG_VERBOSE)));
DEBUG((DEBUG_ERROR, " 0x%08x USING DEBUG DEBUG_ERROR\n", (UINTN)(DEBUG_ERROR)));
  
```

UEFI Application w/ CpuIceBreakpoint

The screenshot shows the Intel System Debugger interface with a UEFI application named "sampleapp.c" open. The code window displays a loop where the application waits for a key press and then reads the key. A specific line of code, `CpuIceBreakpoint();`, is highlighted with a red rectangle. A yellow callout bubble points to this line with the text "Stepping through your code ‘F6’". The Variables window on the right shows local variables for the current module, including `ImageHandle` (type <Void> *, value 0x0000000076a...), `SystemTable` (type EFI_SYSTEM_TABLE *, value {Hdr={Signatu...}), `EventIndex` (type unsigned long long, value 0x000000000000...), `ExitLoop` (type N/A), and `Key` (type EFI_INPUT_KEY, value {ScanCode=0x0...}). A yellow callout bubble points to the `Variables` window with the text "Variables show Locals for current module".

```
//2 7.4
Print(L"\nPress any Key to continue : \n\n");
gBS->WaitForEvent (1, &gST->ConIn->WaitForKey, &EventIndex);

//3 7.5
Print(L"Enter text. Include a dot ('.') in a sentence then <Enter> to exit:\n\n");
ZeroMem (&Key, sizeof (EFI_INPUT_KEY));
gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);
ExitLoop = FALSE;
do {
    //CpuBreakpoint();
    CpuIceBreakpoint();

    gBS->WaitForEvent (1, &gST->ConIn->WaitForKey, &EventIndex);
    gST->ConIn->ReadKeyStroke (gST->ConIn, &Key);
    Print(L"%c", Key.UnicodeChar);
    if (Key.UnicodeChar == CHAR_DOT){
        ExitLoop = TRUE;
    }
} while (!(Key.UnicodeChar == CHAR_LINEFEED ||
           Key.UnicodeChar == CHAR_CARRIAGE_RETURN) ||
         !(ExitLoop) );

Print(L"\n");

return EFI_SUCCESS;
}
```

Stepping through your code “F6”

Variables show Locals for current module

Intel® System Debugger

- Intel® System Debugger software solution for
 - ✓ UEFI BIOS, firmware, bootloader, OS and device driver debug.
- Intel's system software trace solution for
 - ✓ complex hardware/software interaction visible
 - ✓ isolate tricky software bugs quicker
- It supports JTAG via ITP-XDP3 / DCI hosting BSSB / USB3
- It supports UEFI BIOS debug via EDKII debug agent and JTAG
- It is available as part of Intel System Studio

SUMMARY

- ★ Intel® System Studio Debugger Overview

- Identify the Intel® System Studio Debugger host and target

- ★ Getting started with Intel® System Studio Debugger

- Using IPCCLI

- TCF GUI

- XDB GUI

- ★ Configure when target has UEFI debug agent

- ★ Using the INT1 ICEBP instruction as a debug break point

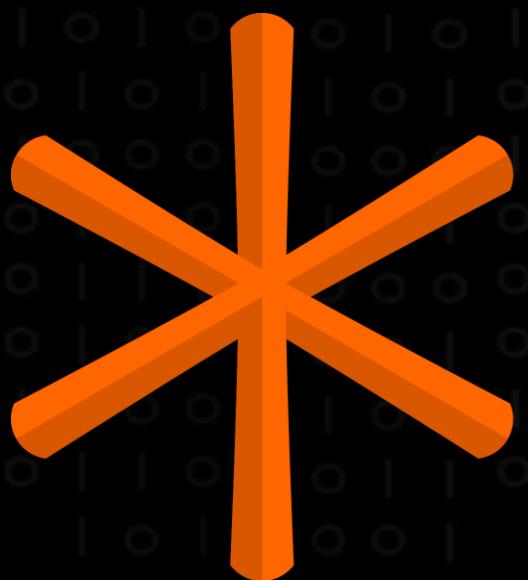
Questions?



Return to Main Training Page



Return to Training Table of contents for next presentation [link](#)



tianocore

