

Intel(R) System Debugger NDA



Intel® System Debugger is a modern, GUI-based architecture system software debugger. It provides a toolset to explore the state of the system, capture and analyze traces from hardware, firmware, and software sources, analyze system crash logs, and much more.

The documentation is available in the following formats:

- HTML bundle at `<install-dir>/documentation/index.html`
- Online book at [Resource & Documentation Center](#) (requires signing in)
- Eclipse* IDE Help. Go to **Help** (or press F1) and select **User Guide** (the top item).

For system requirements, new features, known issues, and a list of supported target components, see Release Notes. You can find them at the [Resource & Documentation Center](#) or the product download page at [Intel® Products](#).

Get Started Here

What Is New

Intel® System Debugger 2023 NDA Update 2350

- Platform Register context for Simics® simulator connections.
- Limited support for the extended PCI configuration space.
- New context called PCIExtCfg, accessible through the PCI Devices view.
- Hexdump export format for I/O Port, MSR and PCI raw views.
- Write access for PCI Devices view.
- Extended Load Current Module, LoadPEIM, BIOSROM and LoadDXE support for BIOS (EDK II) GCC builds.
- TraceCLI command line options for Catalog Service configuration.
- intel_ontarget_trace now supports detection of platform, BIOS, CSME and Audio FW version using TCA.

- Support for ARL-H and DNR-D in Trace.
- New dialog in the UI to show the latest version of Intel® System Debugger.
- Support in the TCA CLI for connecting to the local system, called On-Target connection.

Update Notice

Check the latest training materials covering key use cases:

- [How to Capture Trace Log on Simics® Simulator using Intel® System Debugger](#)
- [How to Capture Intel® Trace Hub \(Intel® TH\) Early Boot Trace over USB2 Debug Class \(DbC\) Cable](#)

Video trainings demonstrating this scenario:

- [Alder Lake target](#)
- [Raptor Lake target](#)
- [Tiger Lake target](#)

To access the materials, you will need to log into the [Resource & Documentation Center](#) (contact Customer Support for assistance).

! See also

[Trainings](#)

Get Started

Intel® System Debugger provides a wide set of tools to perform different debugging and tracing tasks like the ones below:

- Debug BIOS or an EFI application

To follow this scenario, ensure that the binaries you want to debug are compiled with debug information and without optimization (see the [recommended compiler flags](#)), [connect to the target](#), and proceed with [System Debug](#) instructions.

- Capture (or import) and analyze system trace

Depending on whether you want to capture trace of the connected target system or import pre-recorded trace file, you need to start using [System Trace](#) differently.

See [Use Case Overview](#) for System Trace.

- Extract and analyze system crash logs

[Connect to the target](#) and proceed with GUI-based or pure CLI interface of [Crash Log](#).

- Debug Windows* OS

Use [Intel® Debug Extensions for WinDbg](#).

The standard startup procedure looks as follows:

1. Set up your target.
2. Connect your target system to the host machine.
3. Launch Intel® System Debugger and configure target connection.

The detailed steps involved in the startup procedure are given below:

Set Up Target

Configure the target platform BIOS settings and FIT/mFIT settings described in the Platform Closed Chassis Debug User Guide. Sign in to [Resource & Documentation Center](#) and search by document number, [626623](#) for all platforms before Meteor Lake and [730814](#) for Meteor Lake. Refer to the [Troubleshooting](#) section for instructions on troubleshooting target connection issues. If further support is required, contact your BIOS vendor, or your Intel support to get further guidance.

Common Configuration

To enable a target for debugging and tracing, the associated BIOS configuration must be applied. The following instructions are generic and might not completely match your setup. If available, follow the target specific instructions first.

These are examples of options and possible configurations, not all of them can be applied to your target. If an option is available, choose the value that fits best.

1. Check if your target BIOS provides any of the following menus:

- Intel Advanced Menu > Debug Settings
- Platform Configuration > PCH-IO Configuration > Debug Settings

2. Enable Intel® Direct Connect Interface (Intel® DCI): search for a **Intel® DCI enable** or **Platform Debug Consent** option and configure it according to your connection method.

Intel® DCI is the communication layer for USB JTAG debugging required for the following connection methods:

- Intel® Silicon View Technology (Intel® SVT) DbC USB Debug Cable
- Intel® DCI OOB via Intel® SVT Closed Chassis Adapter (CCA)
- Intel® DCI OOB via Intel® SVT Closed Chassis Adapter (CCA) + 2-wire adapter

To use these methods, verify that the Intel® DCI driver is installed. It should happen automatically during Intel® System Debugger installation.

3. Enable debugging of your target: enable **CPU Run Control** options in your BIOS settings to use System Debug and WinDbg features.
4. Enable System Trace (if required and supported) to trace your target execution: search for **Trace Hub** or **North Peak**, or **NPK** options and enable them. This enables Intel® Trace Hub (Intel® TH) and firmware trace sources.

Connect your Target

Depending on your target type, follow instructions for connecting:

Physical Target

1. Check [Supported Platforms and Probes](#) to see what probe types are supported for your target.
2. Plug the debug probe into the target platform.

Refer to the detailed instructions on connecting primary probe types below.

For troubleshooting, use the [System Configuration Checker](#) to diagnose your target configuration.

Probe Setup

To use the Intel® System Debugger on a hardware board the host and target hardware must first be connected. The probe drivers are automatically installed by the installer.

Intel® In-Target Probe - Extended Debug Port (Intel® ITP-XDP)

1. Plug in the probe power supply. The probe fan must start spinning. Failing to connect the power supply when the probe is in use will damage the probe.
2. Plug the USB cable into the probe and into an USB2.0 (or greater) port of your host computer.
3. Identify the XDP port on your target board, usually named “CPU XDP”, and plug the small PCB with connector at the end of the ribbon cable to it.
4. Power on the target board.

Intel® Silicon View Technology (Intel® SVT) Closed Chassis Adapter

1. Enable Intel® Direct Connect Interface (Intel® DCI) debug from target board BIOS.
2. Identify the USB debug port on your target board.
3. Plug the USB cable to the probe and into an USB port of your host computer.
4. Plug the debug cable to the probe and to the USB debug port on the target board.
5. Power on the target board.

USB Native Debug Class (DbC) Cable

1. Enable Intel® Direct Connect Interface (Intel® DCI) debug from target board BIOS.
2. Identify the USB debug port on your target board.
3. Plug the debug cable to the host USB port and to the USB debug port on the target board.

4. Power on the target board.

Lauterbach* Probes

If you need to use Lauterbach* probes, refer to [additional connection instructions](#).

To identify the status of the connection, use the [Target Indicator](#).

! See also

Platform Closed Chassis Debug User Guide. Sign in to [Resource & Documentation Center](#) and search by document number [626623](#).

Virtual Target (Simics® Simulator)

Requirements

Base Package Version (#1000)

The following Base Package (#1000) versions are required:

- Simics® simulator 6: >= 6.0.24
- Simics® simulator 5: >=5.0.202 (deprecated, use Simics simulator 6 instead)

For tracing with Simics® simulator (using System Trace), [additional configuration](#) is required.

Launch the Simulator

You need to enable (remote) access to the Simics® simulator via Target Communication Framework (TCF).

! Warning

Start Simics® simulator from the CLI to be able to connect from the Intel® System Debugger.

If you start Simics simulator from Eclipse*, you are restricted to localhost connection.

1. Launch the target startup script `run-command-file.simics`.

2. In the simulator console, run the following command:

```
start-eclipse-backend
```

3. Execute the following command inside the Simics simulator CLI to expose the simulation to the Intel® System Debugger:

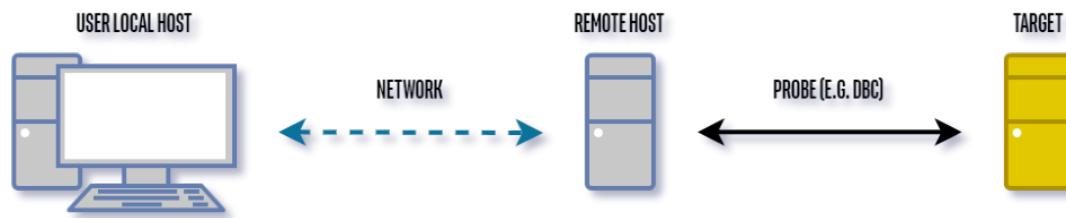
```
start-eclipse-backend TCP::<port> "<session name>"
```

where:

- *port* is any port that is not in use. If you pick a busy one and get an error, just try another one.
- *session_name* (optional) is your custom name for the session.

The console returns your IP and port combination in the following format: `TCP:<ip>:<port>`.

Remote Target



Debug on your local development host a target which is connected to a remote host. This is achieved by forwarding the OpenIPC session running on the remote host to your local development host.

Currently supported:

- Configuration and Detection of remote Target
- Debugging using Intel® System Debugger - System Debug
- Forwarding of OpenIPC session for generic usage (e.g. IPCCLI)

Prerequisites

Ensure your setup contains the following components:

- Local host system
- Remote host system (for example, in a lab)
- Target system to be debugged
- Debug probe

1. Install Intel® System Debugger on the local host system.
2. Install Intel® System Debugger on the remote host system.
3. Enable SSH on the remote host following the OS specific instructions:
 - For [Windows* OS host](#)
 - For [Linux* OS host](#)
4. (Optional) Setup authentication using a key file

After the first login, an SSH session is preserved for the current instance (runtime) of Intel® System Debugger. No user name or password is stored. To skip this initial login every time on first connect, Intel® System Debugger supports the usage of an SSH configuration file. The file path must be “`~/.ssh/config`”.

5. Connect the target under debug to the remote host using a USB probe (for example, [Intel® SVT DbC USB Debug Cable](#)).
6. Follow instructions given in [Remote Target](#) section on the Launch Intel® System Debugger page.

Note

This step is not required if you want to analyze previously captured trace (not to capture new data).

Launch Intel® System Debugger

1. Start Eclipse* IDE.
2. Minimize or close the User Guide view to launch the Target Connection Assistant wizard. If the wizard is not launched automatically, click **New Connection** in the main Eclipse toolbar.

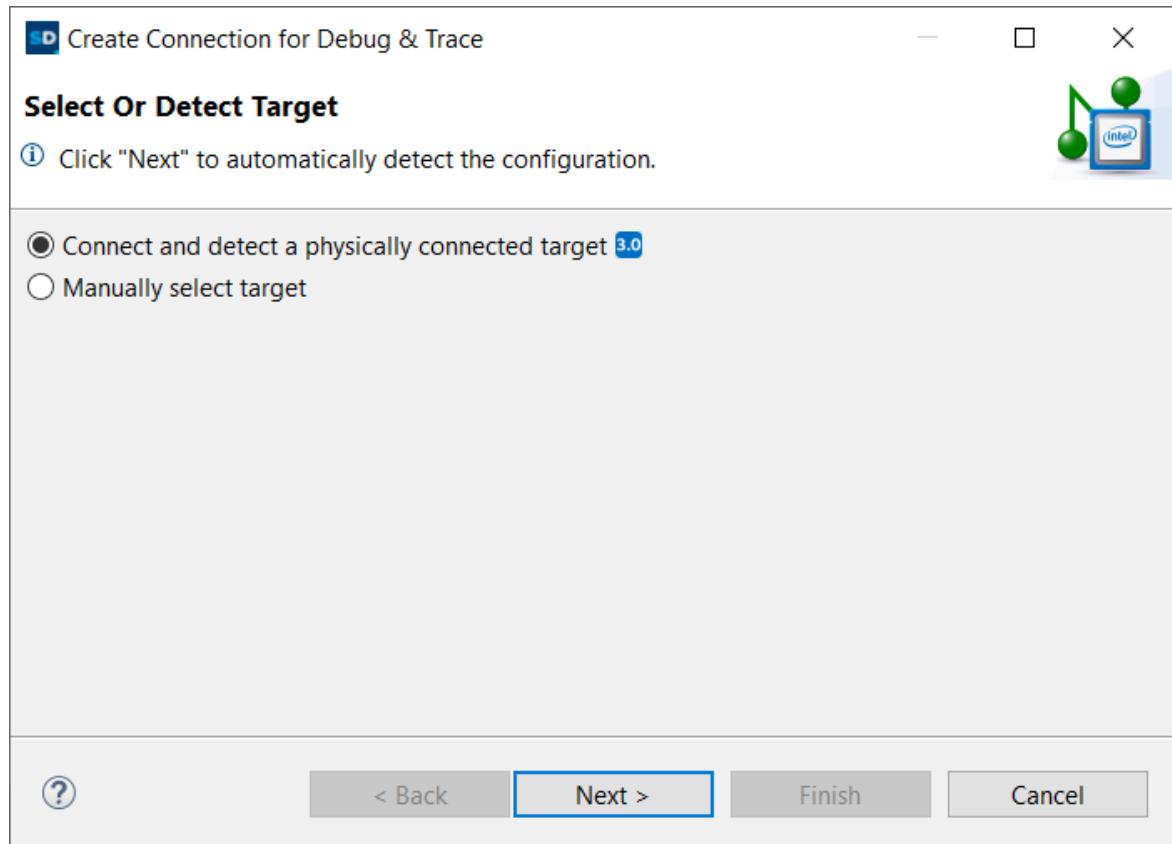


3. If the target is connected properly, the wizard detects it automatically. Choose the connection mode depending on your target:

- **Physical Target**

Select **Connect and detect a physically connected target** and click **Next**.

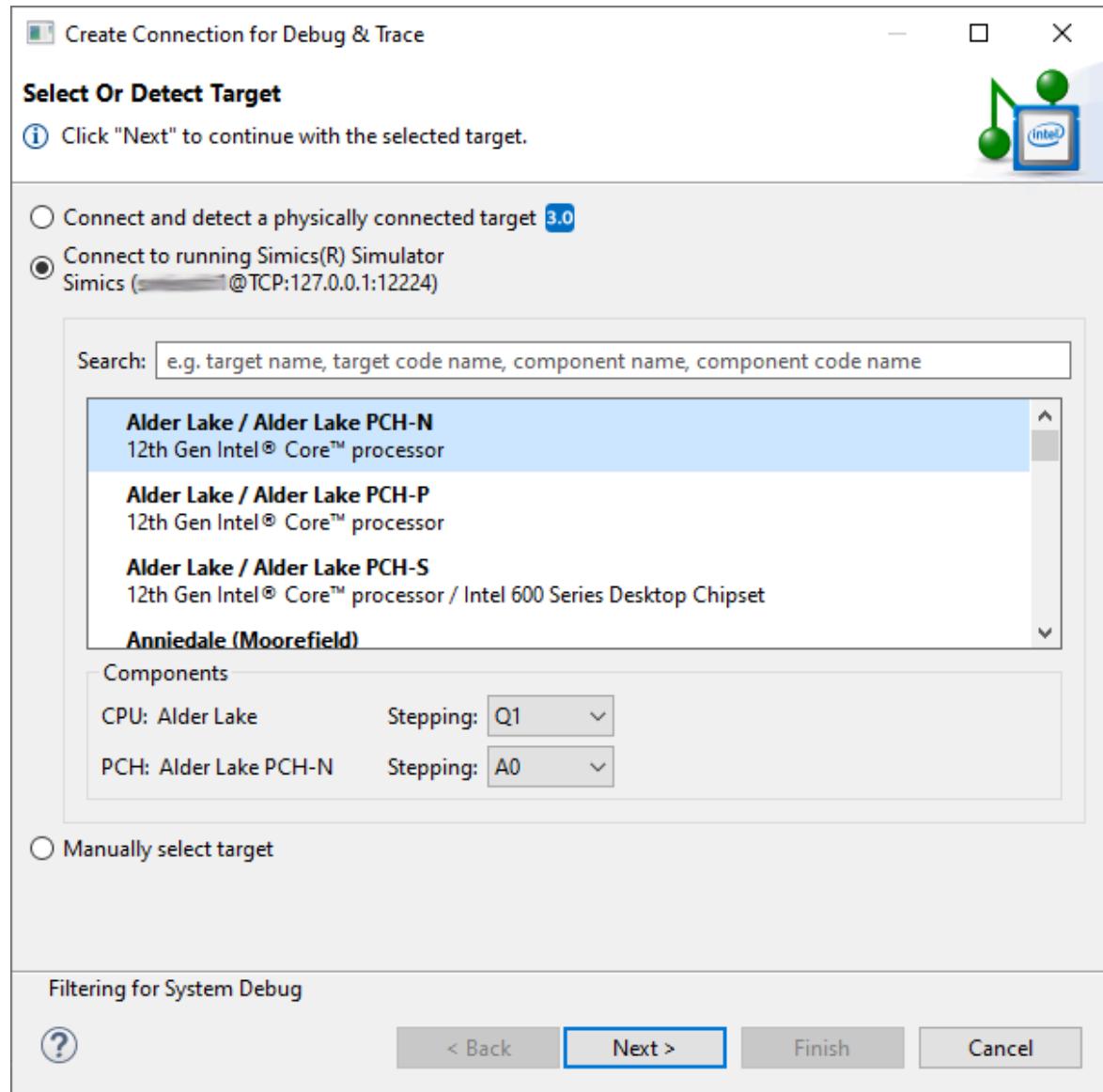
The wizard also displays an icon that identifies the detected probe. To learn more about the icons, see [Target Indicator States](#). To check additional information on the connection, use the [Target Indicator](#).



- **Virtual Target**

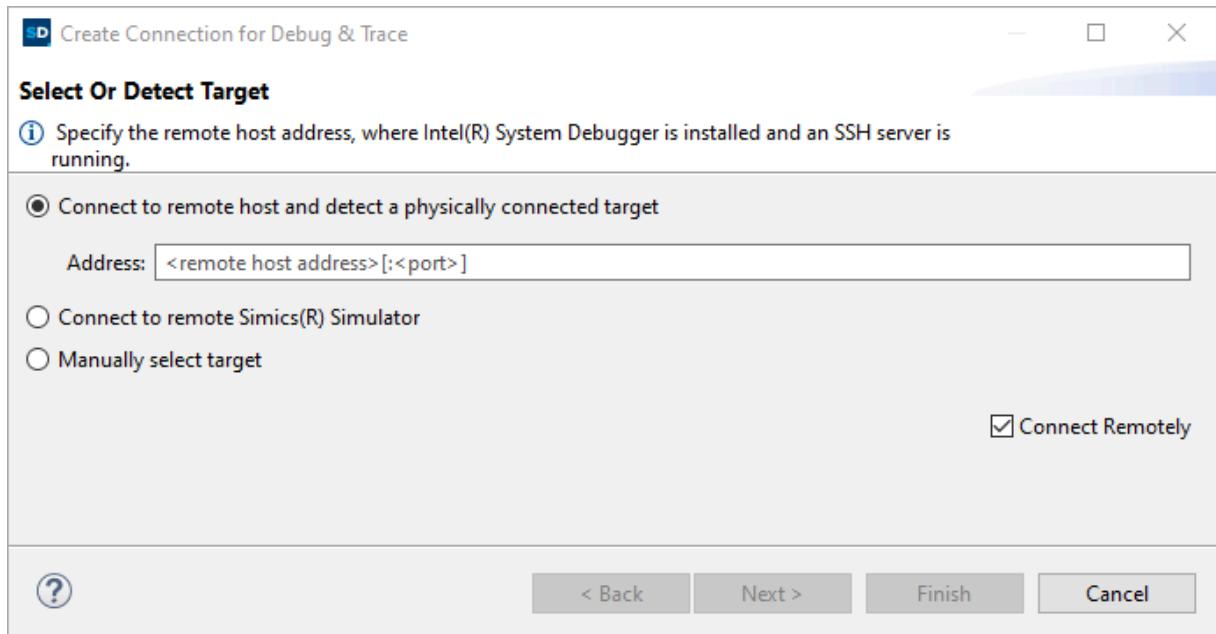
If the simulator is running locally, the wizard displays the detected IP and port.

Select **Connect to running Simics® Simulator**, select your target from the list, and click Next.



- **Remote Target**

Check **Connect Remotely** box at the bottom-right and specify remote host address of your target.



! Note

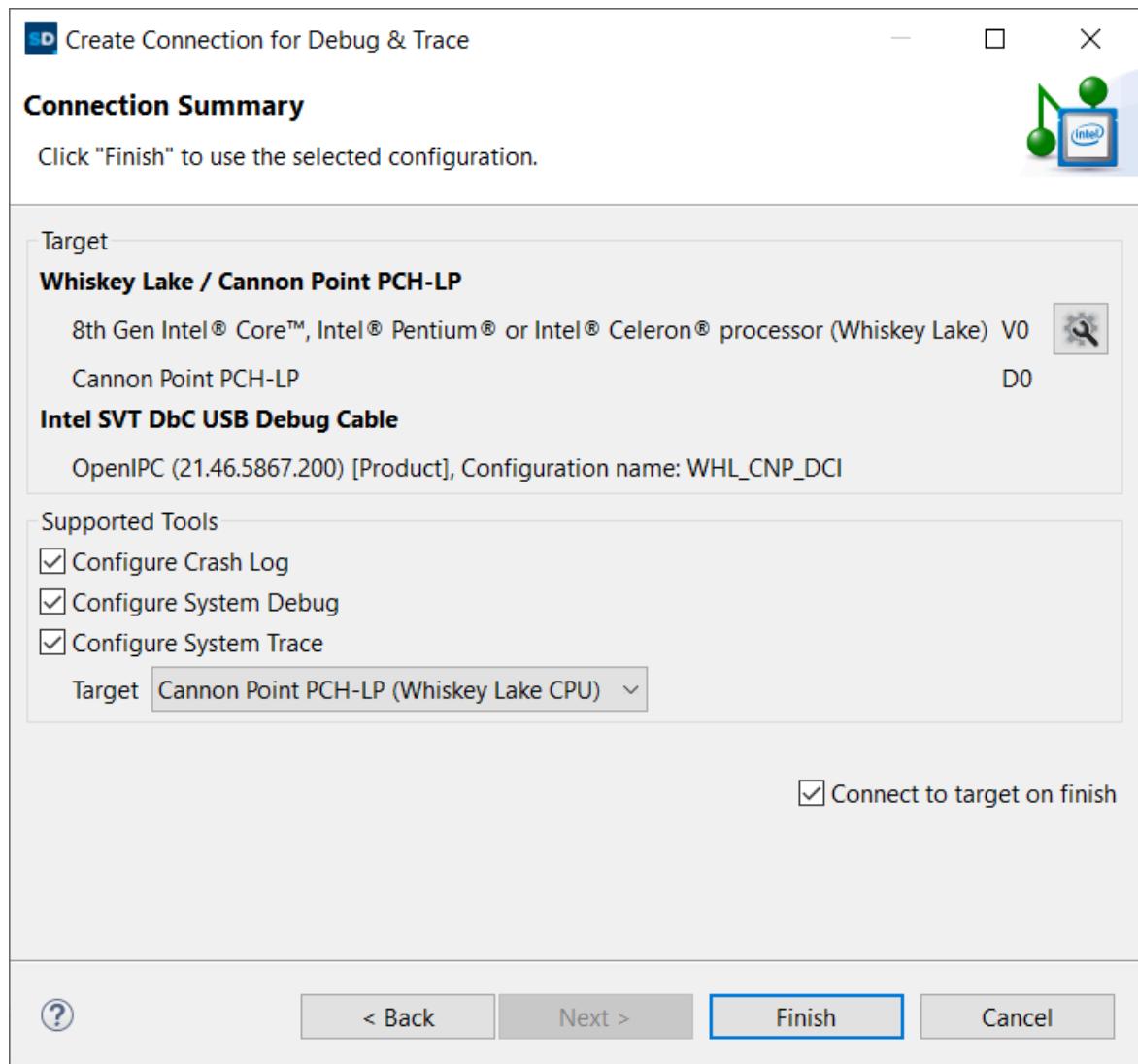
Use the manual mode to configure your connection in the following cases:

- You want to analyze pre-captured trace
- Your target is not detected properly for some reason ("Connect" options are disabled)

4. In the **Connection Summary** dialog, check the pre-configured settings. You can see the following data:

- Components of the detected target and their steppings (if available) in the **Components** pane.
- Probe selected for connection.
- Version of the running IPC provider.
- Intel® System Debugger components supported by the current combination of target and connection method.

See [primary use cases](#) that each component can be used for. If a particular component is not required for your goal, you can choose not to configure it (unchecked a corresponding box).



5. If any of the following is true for your debug session, check [additional configuration options](#):

- Your target platform runs an OS
- You plan to debug a platform across low power states
- You want to change IPC provider settings

6. If you want to establish the connection right now, check the **Connect to target on finish** box at the bottom-right.

7. Click **Finish**.

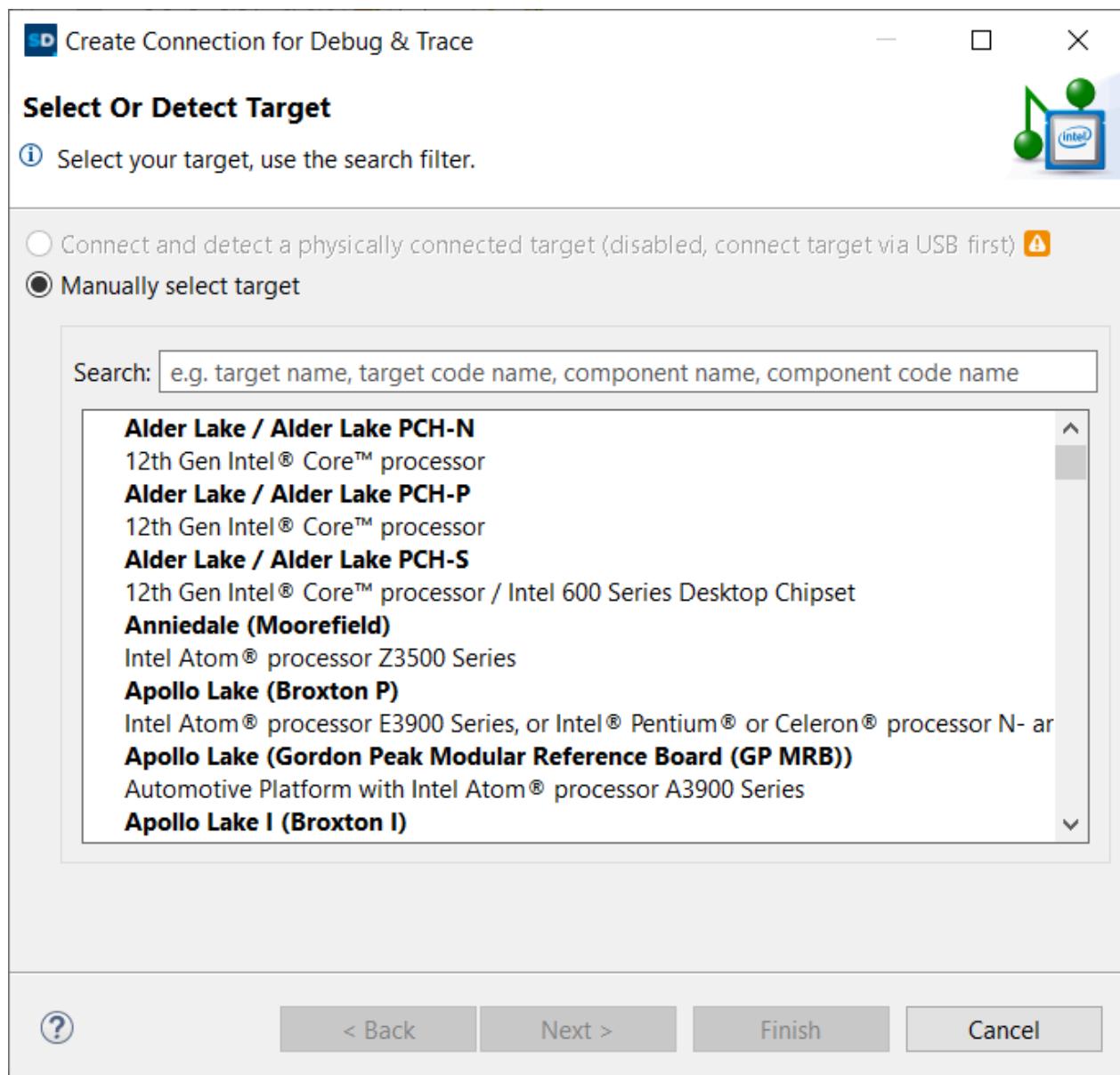
Wait until the debugger completes the session setup. Proceed with instructions for the particular component of Intel® System Debugger.

Manual Configuration

To configure the target connection manually, follow the steps below:

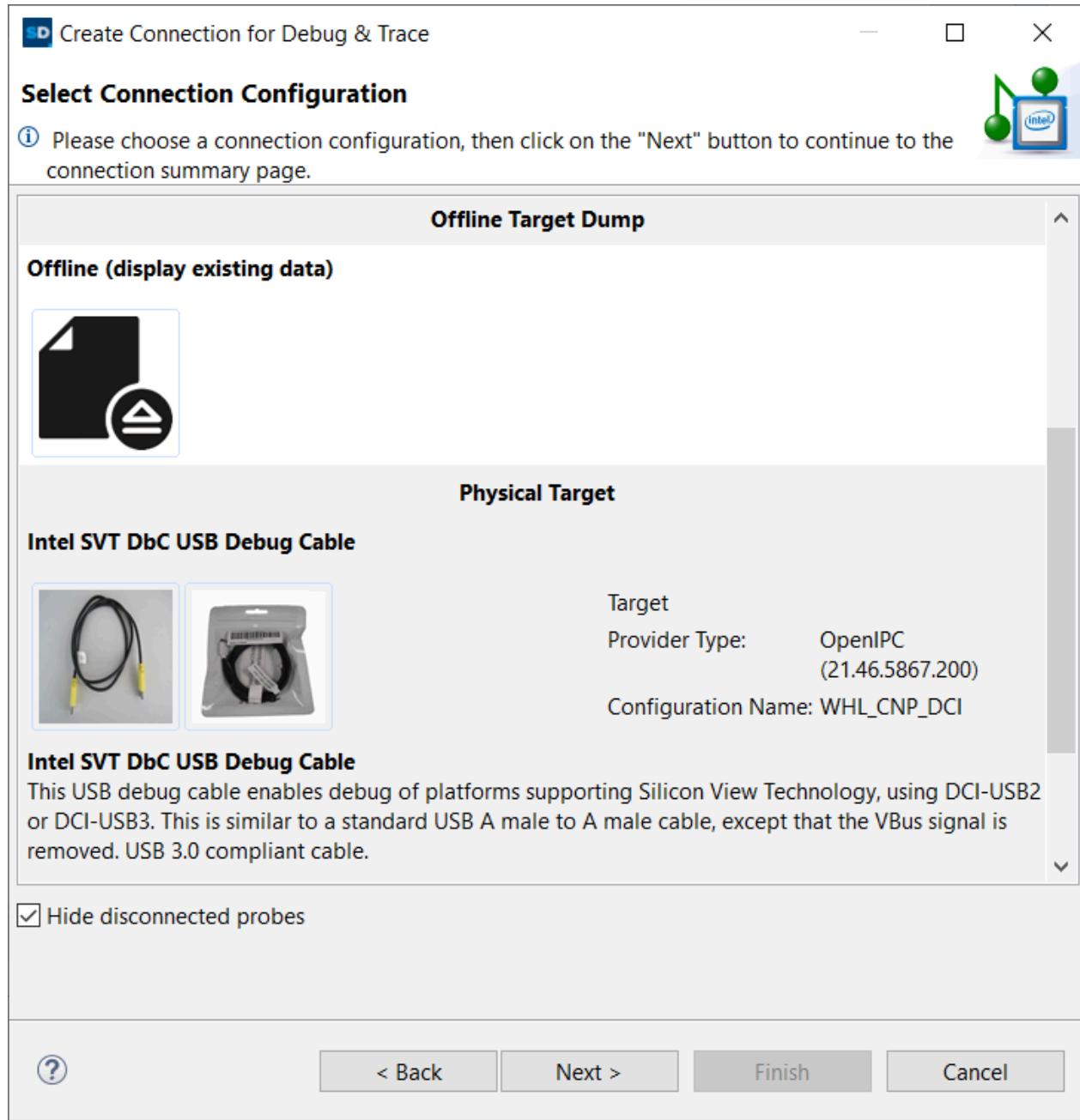
1. In the first step of Target Connection Assistant wizard, choose **Manually select target**, select your target from the list, and click Next.

If you have pre-captured trace to analyze, select the target that this trace has been captured on.



2. Select the required connection method. To display all probes supported for this target (but not currently detected), uncheck the **Hide disconnected probes** box.

For pre-captured trace analysis, select **Offline (display existing data)**.



Click **Next**.

3. Review the final settings in the **Connection Summary** dialog.
4. If you want to try to establish the connection right now, check the **Connect to target on finish** box at the bottom-right.
5. Click **Finish**.

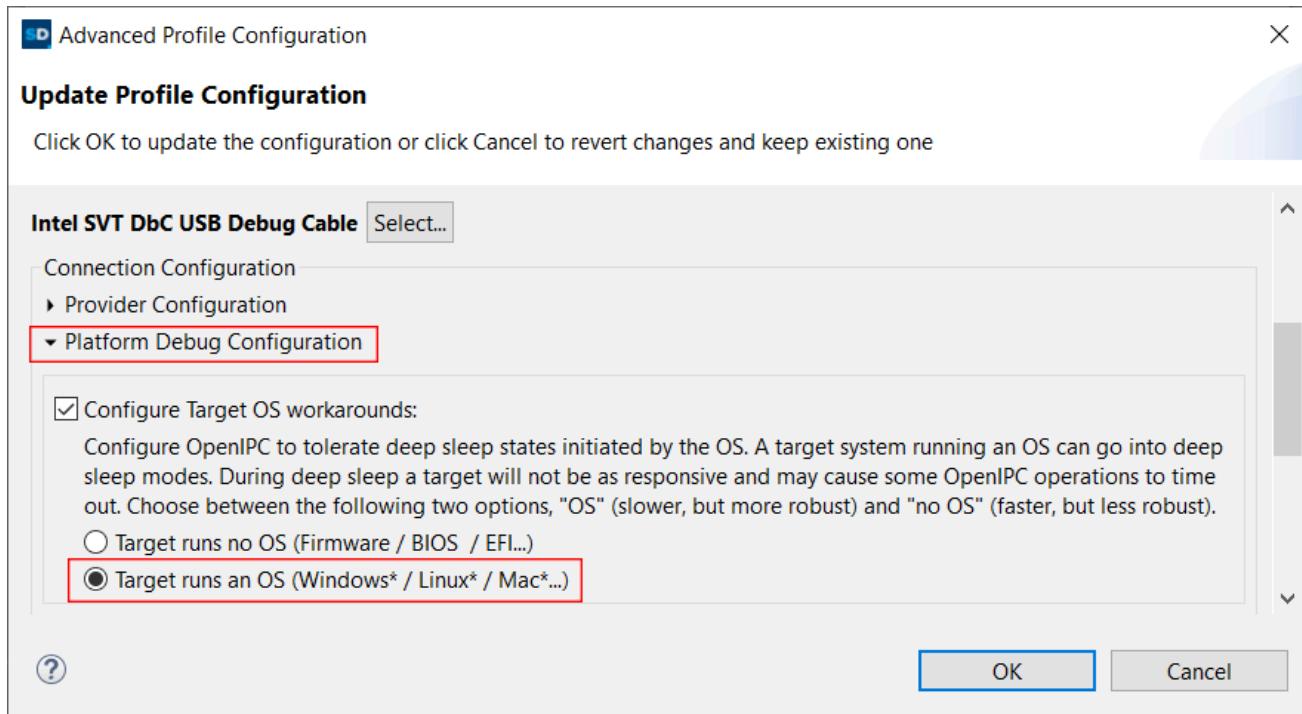
See also

[Troubleshooting](#)

Additional Configuration Options

Target platform runs an OS

In the final Connection Summary dialog, expand the Platform Debug Configuration pane, check the Configure Target OS workaround box, and select the corresponding radio button.



Debug platform across low-power states

In the final Connection Summary dialog, expand the Platform Debug Configuration pane and set the following:

- Check the **Configure Transport Low Power Preset** box and select *Low-Power*
- Check the **Configure High-Bandwidth Transports** box and select *USB2 Only*.

Update Profile Configuration

Click OK to update the configuration or click Cancel to revert changes and keep existing one

Intel SVT DbC USB Debug Cable

Connection Configuration

▶ Provider Configuration

▼ Platform Debug Configuration

 Configure Transport Low Power Preset:

Configures OpenIPC to survive a target's low power states. Normally debugging would be disabled when entering low power states. In order to guarantee this survivability, OpenIPC might have to override some power management flows on the platform, therefore modifying its behavior around entering/exiting low-power states (e.g. the platform might no longer power down a specific voltage rail upon entering a specific low-power state).

 Preserve

 Low-Power

 Configure High-Bandwidth Transports:

Configures OpenIPC to use the specified transportation mode. Using fast transportation modes (USB3) might provide faster communication (useful for DMA and tracing), but they are not always available, and they might incur side-effects on the target (e.g. prevent to enter low-power sleep states).

 USB2 Only

 Prefer USB3


OK

Cancel

Select alternative stepping**Warning**

Changing any default configurations can lead to non-functional connection. Only default settings are fully tested.

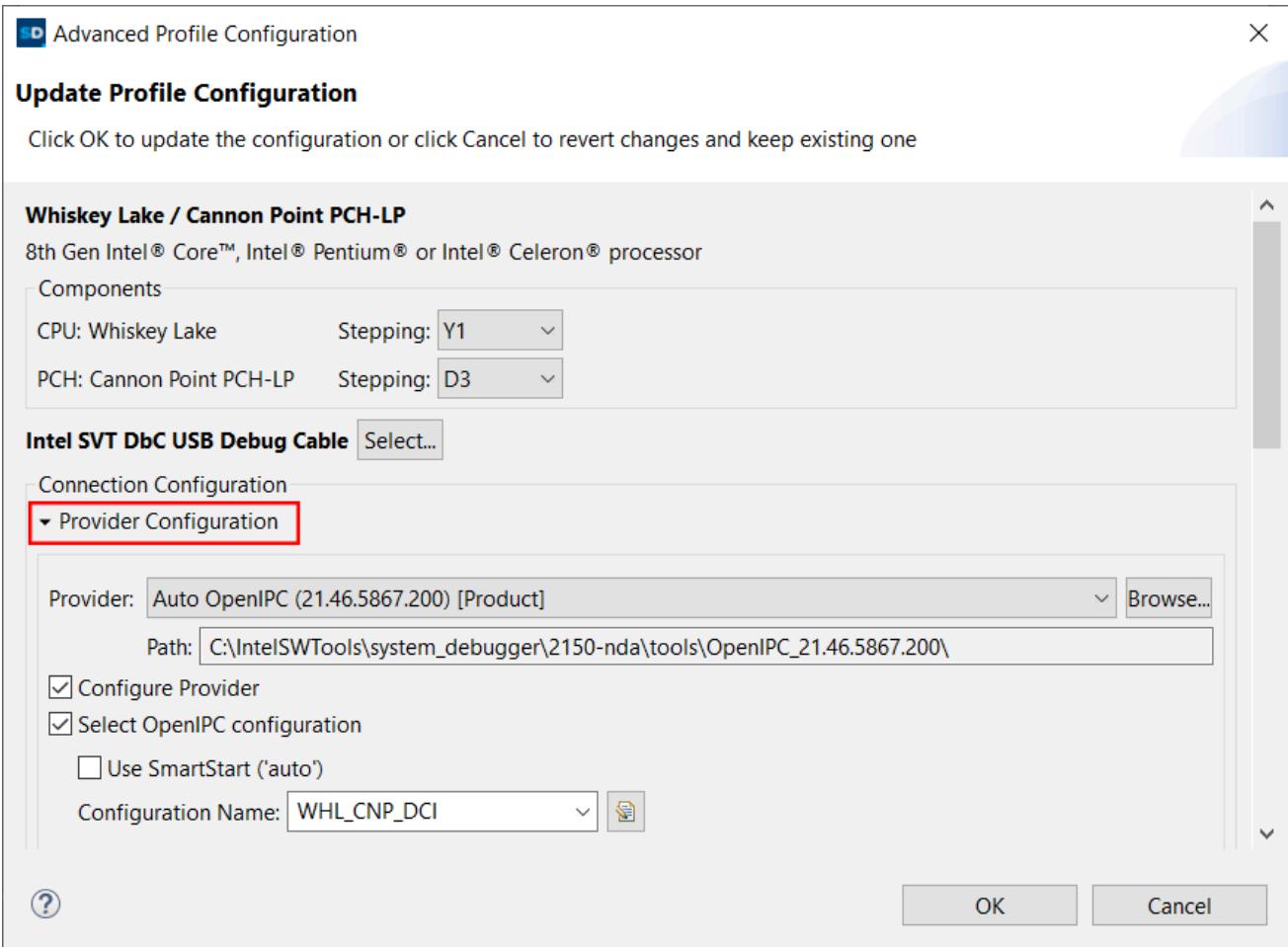
In the final **Connection Summary** dialog, click to open advanced configuration. Use **Stepping** dropdown lists in the **Components** panel.

Use custom provider**Warning**

Using any custom configurations can lead to non-functional connection. Only default settings are fully tested.

In the final **Connection Summary** dialog, click to open advanced configuration and expand the **Provider Configuration** pane.

You can add a custom IPC provider or change the name of the IPC provider configuration.



For assistance, contact [Customer Support](#).

Version History

This page contains update summary of previous releases. To check full Release Notes for previous product versions:

- Online: open the [online book](#) and expand the Version drop-down menu on the top-left.
- Offline: go to the [Intel® System Debugger download page](#) at Intel® Products and select a required version from the drop-down menu.

Intel® System Debugger 2023 NDA Update 2350

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Added a new dialog in the UI to show the latest version of Intel® System Debugger.
- Added support in the CLI for connecting to the local system, called On-Target connection.

Intel® System Debugger - System Debug

- Enabled Platform Register context for Simics® simulator connections. From Simics® simulator Base 173 a new context, called Platform Registers has been accessible. This context is now accessible through the

Platform Registers Dictionary view. R/W actions can be performed to the register that are shown on this context. Also they can be added to Platform Register Watch view and perform R/W operation on the bitfields through the Register Editor View. Due to the high number of registers that this context contains the search is also enabled.

- Added limited support for the extended PCI configuration space. Current implementation supports the extended PCI configuration space during early EFI boot stage only. Added a new context called PCIEExtCfg. This context is now accessible through the PCI Devices view, when the extended PCI configuration space is enabled with the new button.
- Added hexdump export format for I/O Port, MSR and PCI raw views.
- Added write access for PCI Devices view.
- Extended Load Current Module, LoadPEIM, BIOSROM and LoadDXE support for BIOS (EDK II) GCC builds.

Intel® System Debugger - System Trace

- Added TraceCLI command line options for Catalog Service configuration (check ‘intel_systrace catalog_remote –help’)
- intel_ontarget_trace now supports platform detection using TCA, as well as detection of BIOS, CSME and Audio FW version. This information will be used to select right catalog.
- Added support for ARL-H.
- Added support for GNR-D.

Intel® Debug Extensions for WinDbg

- Support for extraction of crashlog data from live target and memory dump using Intel® Debug Extensions.

Fixed Issues

Target Connection Assistant

- Fixed various issues with the UI terminal on recent versions of Windows 11.

Intel® System Debugger - System Debug

- Fixed kernel module loading with “Load Linux Kernel Modules from Kernel Image Folder” button.
- Fixed source loading in module view for elf symbols.
- Fixed PCI Devices raw view for devices with address 0.

Intel® System Debugger - System Trace

- Fixed ACE catalog matching in TraceCLI.
- Fixed memory extraction by using trace extraction service.

- Fixed pre-BIOS NPK tracing from Bootstall Throws Errors.
- Fixed early boot script fails during capture. Note: due to bad power supply might get Power On events during power Off, that would confuse early boot script.
- Run memory sanitizer and thread sanitizer on engine and tms services. Fix various number of issues that improves tool stability.

Intelligent Debug & Validation Tool (IDV Tool)

- Enabling analog capture has been blocked for signals in BMC configurations, BMC supports only digital signals.
- Fixed capture time value auto-multiplication if time unit was not entered.

Intel® System Debugger 2023 NDA Update 2346

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Added an option in the UI to decompose debug tokens getting back knobs which the token was created with.

Intel® System Debugger - System Debug

- Write access for I/O Port and MSR views.
- Automatic address alignment for I/O Port view.
- Binary export format for I/O Port, MSR and PCI raw views.

Intel® System Debugger - System Trace

- Reworked the command line interface for the OnTarget Agent.
 - Added dedicated options to start/stop trace collection from the command line.
 - Added new command line flag ‘fw-only’ to only capture FW logs (without ETW messages).
 - The ‘intrusive’ flag is now enabled by default, use ‘non-intrusive’ to disable it.

Intel® Debug Extensions for WinDbg

- Support for creating full memory dump in non-existing folder.

Fixed Issues

Target Connection Assistant

- Fixed an issue that would cause connecting to a target to freeze due to the Windows Terminal on Windows 11.

Intel® System Debugger - System Debug

- Fixed ITrace view action management on a running target.

Intel® System Debugger - System Trace

- Fixed an issue with the loading of the default Profile in the trace configuration editor when using the ITP-XDP3 probe.

Intel® Debug Extensions for WinDbg

- Fixed exception when breaking preserving state of core.

Intel® System Debugger 2023 NDA Update 2342

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Added the ability to select multiple pid.bin files when clicking the “Load from File” button.

Intel® System Debugger - System Debug

- Added customizable number of columns to the I/O Port view
- Added option for copying the address from the Interrupt Descriptor Table view

Intel® System Debugger - System Trace

- Added support for Linux kernel tracing, see “Linux kernel tracing” chapter in the System Trace user guide for details
- Added OOB MSM trace support to Granite Rapids platform
- Rename VPU trace name to NPU
- Added support for Lunar Lake platform for TraceCLI Earlyboot

Intel® Debug Extensions for WinDbg

- Added support for Intel Neural Processing Unit (NPU) driver in Intel® Debug Extensions for WinDbg over Simics® simulator.
- Added OpenIPC version detection.

Fixed Issues

Target Connection Assistant

- A styling issue was fixed which would make it difficult to see which entry in the connection configuration wizard was selected.

Intel® System Debugger - System Debug

- Fixed I/O Port view endianess display
- Fixed environment variable enablement for hardware breakpoints over reset

Intel® System Debugger - System Trace

- Fixed trace streaming using CCA 4-wire probe
- Fixed timestamp issues (showing only ??) for TraceCLI Earlyboot

Intel® Debug Extensions for WinDbg

- Fixed !ipi and !cpuinfo functions for LNL platforms.

Intelligent Debug & Validation Tool (IDV Tool)

- Removed misleading tooltip message over vertical white bar at the end of waveform which said that waveforms to the right existed.

Intel® System Debugger 2023 NDA Update 2338

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Add UI support for debug token injection and generation via Intel(R) Download and Execute (DnX). Add DnX probe detection to correctly show status in the IDE.
- Add DnX command line interface (CLI). DnX python module provides “dnx” extension point to run DnX commands directly from the command line.

Intel® System Debugger - System Debug

- Improved the Interrupt Descriptor Table view to better support other CPU architectures
- Refresh event added for Address, Size and Count fields in I/O Port and MSR view on “Enter” key
- Added support for finding the bootstrap processor (BSP) on the Intel architecture

Intel® System Debugger - System Trace

- Added support for tracing Intel® Trace Hub memory and extraction on Battlemage platform

Intel® Debug Extensions for WinDbg

- Introducing support for pci devices discovery. Adds WinDbg* command to dump information about pci devices in a manner similar to MSFT.

Fixed Issues

Target Connection Assistant

- Fixed mapping of reset type in the Intel(R) CSME reset logic.
- Fixed debug token creation UI on Linux. Properly display debug token creation status.

Intel® System Debugger - System Debug

- Fixed the overlapping lines and the bit selection display in the Register Editor view
- Fixed the bit group “Size” in Platform Register view
- “Copy Stack Details” enabled for all threads and stack frames

Intel® System Debugger 2023 NDA Update 2334

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Updated to Eclipse 2023-06 and latest JRE 17.
- Re-enable telemetry again after switching from Google Analytics* to a new Intel internal backend, users can use the consent dialog to opt-in and opt-out.
- Improved Intel® DnX CLI, implemented new error reporting providing user instructions how to improve the situation together with the existing error code. Implemented download of current firmware.

Intel® System Debugger - System Debug

- Column sorting added to Modules view
- Full compatibility with RISC-V for Interrupt Descriptor Table view

Intel® System Debugger - System Trace

- Added AET trace support for Lunar Lake platform
- Added trace support for Clearwater Forest Simics platform

Intelligent Debug & Validation Tool (IDV Tool)

- Added new Baseboard Management Controller (BMC) platform project type and support for the BMC-server-based Platform Signal Tracing in the IDV Tool

Fixed Issues

Target Connection Assistant

- Fixed Lunar Lake Hotham support to support debug token creation and injection.
- Fixes 2K debug token signing. Change has been verified with BXT-P platform.

Intel® System Debugger - System Debug

- Jump to Source option in Symbols view will take into consideration partial paths.

Intel® System Debugger - System Trace

- “Decode as” Dialog in Eclipse will now show all support target platforms.

Intel® System Debugger 2023 NDA Update 2330

Check the summary below or see the [online version](#).

New Features

Intel® System Debugger - System Trace

- Added new version of the ITH Windows Driver with new functionality for intrusiveness control. Its recommended to install this new ITH Driver 10.0.22000.686 by enabling the checkbox in the Installer menu, or installing it manually from BKC.
- Intrusive flag allows to prevent power managing of Intel® Trace Hub device by OS Windows. When intrusive flag is set, trace captures will be in lossy mode, but preventing low power states. Default mode is to allow low power states, but loss messages when memory is not available for use by Intel® Trace Hub.

Intelligent Debug & Validation Tool (IDV Tool)

- Implemented support for signal names in color mapping.

Fixed Issues

Target Connection Assistant

- Fixed stable Python API usage for Intel® Download and Execute (Intel® DnX) Python WHL file, included intel.dnx-0.0.9.200-cp36-abi3 supports Python 3.6 - 3.12.
- Fixed hang on reset complete on MTL-P platforms.
- Fixed usage of CSE boot stall while live target information updates are enabled. If an ongoing reset has been detected, Intel® System Debugger will delay the live target information update queries until the reset is complete.

Intel® System Debugger - System Debug

- Configure Trace dialog will not have any options selected if no rings are reported.
- Fix I/O Port view on adjusting the display to a larger number of requested bytes.
- **Update the following labels in I/O Port view:**
 - “Port count” is now “Size”
 - “Size” from the context menu is now “Cell Size”

Intel® System Debugger - System Trace

- Added the missing eclipsetcf Python WHL file when creating the Target Agent ZIP package from the Eclipse UI.

Intel® System Debugger 2023 NDA Update 2326

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- PoC of Debug Token injection and Firmware update using Intel® Download and Execute (Intel® DnX).
- Improved detection of new Arrow Lake and Meteor Lake platforms.
- Added / Improved debug token support for Apollo Lake, Arrow Lake and Comet Lake.

Intel® System Debugger - System Debug

- Added I/O Port View and documentation.
- Added option for re-enabling hardware breakpoints over reset.
- Improved modules filtering in Symbols view
- Added Raw MSR View and documentation.
- Added RISC-V PageTables View.
- Added RISC-V Interrupt Vector Table support.

Intel® System Debugger - System Trace

- Implemented a feature to automatically reconnect to the Target Agent feature in Eclipse.
- Added TraceCLI function to dump NPK registers (trace.debug.dump_tracehub_registers()).
- Updated PCH/SOC ESE Trace source naming convention (SOC ESE++ -> ESE_SOC, PCH_ESE -> ESE_PCH).

Intel® Debug Extensions for WinDbg

- Added support for Power Event Tracing for Processor Trace in WinDbg extension.
- Added support for remote hardware debugging.

Intelligent Debug & Validation Tool (IDV Tool)

- Added Drag and Drop to change signal order in signal tab and waveform viewer.
- Added Waveform Viewer signal order sorting if no lane order is specified.
- Added an example project with pre-built rules and captures to easily try out IDV Tool features.
- Updated to JRE 17 and updated to latest available Eclipse.

Fixed Issues

Intel® System Debugger - System Debug

- Jump to Source in Symbols view will take into consideration the defined path map.

Intel® System Debugger - System Trace

- Resolved MTL-S target hang issue when tracing across warm resets and Sx flows.
- Fixed issue where TraceCLI debug markers did not show up in the Eclipse Message view when starting the capture from Eclipse for MTL.

Intel® Debug Extensions for WinDbg

- Fixed mask values from !mtrr in WinDbg extension.

Intel® System Debugger 2023 NDA Update 2322

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Implemented hyper links in System Debug Log view. URLs are marked as hyperlinks and can be clicked to open their location in the associated default Eclipse* editor or with an external browser.

Intel® System Debugger - System Debug

- Arguments of the LoadThis command can now be customized by using the extension “Load Module (Custom Search)”.
- Added new PCI config space tab to the PCI devices view.
- Added Arrow Lake S platform registers to enable Platform Register Dictionary View.

Intel® System Debugger - System Trace

- Added BIOS support for Sierra Forest
- Added Secured Started Services Module (S3M) support for Sierra Forest.
- Added Simics® simulation support for Panther Lake.
- **Added a new feature to TraceCLI which allows to send service messages from the debug host to the Intel® Trace Hub on the target. This allows to mark point of interests directly in the trace session (enabled for ADL and MTL).**

Example usage in Python* shell: `trace.debug.start_debug_marker()`

Intelligent Debug & Validation Tool (IDV Tool)

- Added lane reordering in timeline viewer by drag and drop
- Added support for hidden signals in lane order mappings

Fixed Issues

Target Connection Assistant

- Fixed hang on auto detection. If TCA has started OpenIPC to do auto detection it will automatically terminate OpenIPC to decrease detection time and prevent hangs on OpenIPC initialization.
- Fixed update flow on reset to query less data and delay refresh to post IPC reset complete event when target is released from boot stall. This improves early boot tracing and overall reset performance.
- Fixed usage of remote hardware connection. Support finding of new OpenIPC locations.

Intel® System Debugger - System Debug

- PCI Full Scan option is now displaying all devices in the list.

Intel® System Debugger - System Trace

- TODO

Intel® Debug Extensions for WinDbg

- Improved multiple breakpoints logic on Simics® simulation debug sessions.
- Fixed connection establishment to Sapphire Rapids platforms.

Intelligent Debug & Validation Tool (IDV Tool)

- Added restoring serial port communication to IDV Tool HW after resuming from Windows sleep or hibernation.

Intel® System Debugger 2023 NDA Update 2318

Check the summary below or see the [online version](#).

New Features

Documentation

- Generate Intel® System Debugger Python API and include in Intel® System Debugger User Guide.
- Include IPCCLI documentation in Intel® System Debugger User Guide.

Target Connection Assistant

- Added auto complete of remote host addresses to new Connection dialog. Maintain list of previously used remote host addresses in user settings (can be cleared using TCA Preference Page). These URLs are shared in user folder and can be used across multiple Eclipse workspaces.
- Added Simics® simulation support for Battlemage and Panther Lake platforms.

Token Management

- Extended debug token creation to support platforms previously supported with Platform Flash Tool (PFT). Include configurations for: Cannon Lake, Catlow, Cedar Fork, Comet Lake, Eagle Stream, Kaby Lake, Lakefield, Lewisburg, Tatlow.
- Update Debug Token Library to support TCA Collateral and re-use debug token configurations of UI.

Intel® System Debugger - System Trace

- Added BIOS support for Granite Rapids.
- Added Secured Started Services Module (S3M) support for Granite Rapids.
- Added Simics® simulation support for GNR Workstation.
- Added new mechanism to handle decode catalog files to Trace CLI which allows to automatically select the correct catalog file on decode if previously installed.

Usage on command line: intel_systrace catalog –import command to import new files and directories Usage in Python* shell: trace.catalog API

Fixed Issues

Intel® System Debugger - System Debug

- Added missing text box for IO Break (C/C++ Event Breakpoint) address field.

Intel® System Debugger - System Trace

- Trace Earlyboot script supports re-run in same location and clean-up previous run.

Intelligent Debug & Validation Tool (IDV Tool)

- Improved serial connection error reporting after resuming from sleep or hibernate and provided steps how to restore connection.
- Fixed sending incorrect (too low) value of SGPIO reference voltage to IDV Tool hardware.

Intel® System Debugger 2023 NDA Update 2314

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Implemented auto detection of platforms connected via Closed Chassis Adapter (CCA) + 2-Wire adapter.
- Implemented auto detection of platforms connected via Closed Chassis Adapter (CCA) + I3C adapter.
- Implemented Status Report summary which is presented when the generation has been completed. The dialog presets the content of the “meta.json” which is included in the status report. The “meta.json” file contains information about the installation, what Intel® System Debugger version, what components are installed, and the overall result of the status report.

- Implemented PoC of vUART feature. The vUART allows to read the BIOS and EFI shell serial output through OpenIPC without connecting an additional physical serial adapter. This is currently only for internal usage and requires specifically enabled platforms. Work with Intel® System Debugger Team to receive access.

Intel® System Debugger - System Debug

- Enabled Trusted Domain Execution (TDX) debugging which now supports TDX related Platform Breaks including “SEAM root” mode and “TD guest” mode.
- Add processor trace events decoding. CPU events appear in the trace output together with instructions.
- Add guest physical memory access for Simics® simulator.
- Add support for operating modes for Simics® simulator.
- The Granite Rapids (GNR) register database was updated with the content of EDS vol2 1.1 version.
- A new CLI command `load_kernel_modules()` has been added.

Intel® System Debugger - System Trace

- Add option to remove marker In timeline viewer, the marker can now be deselected either via the context menu (Go To > Deselect) or double click while the Alt key is pressed.

Intel® Debug Extensions for WinDbg

- Introducing Platform based Register Viewer based on RegDB. Contains WinDbg* commands to monitor and modify values of architectural and platform registers specific to connected platform.

Intelligent Debug & Validation Tool (IDV Tool)

- Added Platform Connected support (Requires new Firmware)
- Added Rule Copy across projects
- Added Rule Creation Date column under Setup Rules

Fixed Issues

Target Connection Assistant

- Removed empty Part ID from globally valid debug token.
- Fixed detection of Granite Rapids (GNR) and Sapphire Rapids (SRF) platforms.
- Fixed reading of Target Information on Meteor Lake S B1 platforms.
- Fixed hang of reset caused by Hotham `request_link`.

Intel® System Debugger - System Debug

- Capability to read Memory Mapped Register (MMR) from the Platform Register Watcher was fixed.
- Search feature is fixed for Platform Register context.
- **Fixed the following issues in the Interrupt Descriptor table:**

- Selector and Offset columns are now editable.
- Attributes and Type are displayed correctly in the Register Editor.
- Write, Refresh and restore buttons are now constantly visible in the descriptor table views.
- Platform Register Watch view will keep its content after reconnecting to the debug session if a Simics® simulator target is used.

Intel® Debug Extensions for WinDbg

- Fixed cleanup of COM instance

Intelligent Debug & Validation Tool (IDV Tool)

- Better error messaging when setting up invalid rules
- Improved Linux support

Intel® System Debugger 2023 NDA Update 2310

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Documented Offline Debug Token Generation flow.

Intel® System Debugger - System Debug

- Added TDX registers on MSRs list in Platform Register View.

Fixed Issues

Target Connection Assistant

- Fixed detection of MTL-P 682 B1 Stepping.
- Fixed loading Debug Token Part ID from a path including spaces.
- Fixed Processor State in Connection View after resume.
- Disable Telemetry in Target Connection Assistant and Target Indicator, even the user has opted in to participate in the Intel® Software Improvement Program.

Intel® System Debugger - System Debug

- Processor trace instruction view will not reload the list after fetching additional data.

Intel® System Debugger 2023 NDA Update 2306

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Show Simics® simulator specific connection indicator in IDE.
- Extend debug token support for: DG1, Tiger Lake platforms and Elkhart Lake-based platforms.
- Implemented support for offline debug token generation. Extend debug token creation dialog to import existing Part ID binaries.

Intel® System Debugger - System Debug

- The hardware topology updates automatically in the “Debug View” on new device detection.

Intel® System Debugger - System Trace

- Switched all raw streaming use-cases to use the new IPC_Stream_Reader.
- Added support for periodic Intel® Trace Hub flush to TraceCLI EmergencyBoot.

Intel® Debug Extensions for WinDbg

- Improved error handling.

Fixed Issues

Target Connection Assistant

- Fixed possible segmentation fault on “tca.status()”.

Intel® System Debugger 2022 NDA Update 2250

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Implemented debug token custom BIOS payload fields.
- Update Meteor Lake OEM BIOS Payload to latest firmware.
- Implemented Eclipse* Help / Browser navigation actions (added home, backwards and forward).

Intel® System Debugger - System Debug

- Added support for Hypervisor-Managed Linear Address Translation (HLAT)
- Implemented Core Group Selection for JTAG Hardware Targets

Intel® System Debugger - System Trace

- Added On Target Trace feature. See full documentation of this new feature
- Added support for TraceCLI Earlyboot emergency mode for older (SKL and newer) platforms
- Preview: TraceCLI session can be viewed in the Eclipse GUI

Intel® Debug Extensions for WinDbg

- Support for source level debugging.
- Support for importing Architectural LBRs.
- Support for Simics® simulator reverse debugging.

Intelligent Debug & Validation Tool (IDV Tool)

- Added possibility to set “ignore” to all Analog Voltage Level rules parameters to allow checking static voltages.
- Added color templates for waveforms in Timeline Viewer.

Fixed Issues

Target Connection Assistant

- Auto connect Simics® simulator connections after new Connection Wizard.
- Fixed crash on Hotham get firmware information.
- Fixed Hotham support on MTL-S.

Intel® System Debugger - System Trace

- Removed warning messages from the console for invalid entries in catalogs/trace extensions
- Fixed ability to set timeline definition file name in the new timeline definition creation wizard.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed invalid pin voltage threshold settings possible in Setup Signals tab.
- Fixed truncating total capture time read from VCD file “Total Cycle” field if the value was greater than ~43s.
- Fixed incorrect error message while attempting to create a FPGA backup file in read-only folder.

Intel® System Debugger 2022 NDA Update 2246

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Implemented generate report functionality. The status report contains all available information including log files and dumps.
- Extended live target firmware information to include further Intel® Converged Security and Management Engine (Intel® CSME) information.
- Implemented preview of virtual Python* Shell project, to integrate the shell workspaces into Eclipse Project View. Option can be enabled in TCA / Python* Shell preferences.
- Implemented UI option to externally sign a generated debug token. This allows customers to use their own signing infrastructure.

Intel® System Debugger - System Debug

- Improved launch configuration UI to show log file path permanently.
- Enable log file generation by default to simplify error reporting.

Intel® System Debugger - System Trace

- Improved MIPI SyS-T parser, added support for SBD packet type and subtype

MIPI System Software Trace (SyS-T) Decoder is now supporting a subset of the SyS-T Structured Binary Data (SBD) specification. Support is implemented by processing JSON SBD data and generating custom metadata PacketTypes. SBD allows more flexible log decoding under SyS-T format, including parameter compression, custom formatting, reordering, enum descriptions, and more. For more information refer to MIPI SyS-T Specification v1.1.

Intel® Debug Extensions for WinDbg

- Add debug support for Simics® simulator instances.
- Updated BSOD AR generator functionality to version 1.7.2.

Fixed Issues

Target Connection Assistant

- Intel® DFx Abstraction Layer (DAL) is not included starting from this release. For more information, contact [Customer Support](#) or send a message to bringupkitnda@intel.com.
- Filter USB probes which are disabled in Windows* OS device manager. Support auto detection use cases with connected but disabled Lauterbach* probes.
- Fix validation of Simics® simulator URL for detected local sessions.
- Fixed default enablement of Target Information and Token Management for supported platforms.
- Improved Simics® simulator Log View, fixed right click menu, performance and selection issues.
- Improved JTAG Hardware performance, use separate worker thread to process events received by IPC API.

Intel® System Debugger - System Debug

- Fix “Trace halts when AET is enable after AET configuration”
- Fix “View memory context menu must be available only in BAR element types on PCI device view”
- Fix missing warning message for out-of-date debugging information, whenever a path map is used.
- Fix wrong image size for TE images in the modules view.

Intel® System Debugger - System Trace

- Add decoder(s) path information to the console window

MIPI System Software Trace (SyS-T) Decoder will log all collateral that is loaded for trace processing. One SyS-T client will produce one line, describing these SyS-T client name (for example, BIOS, PMC) and path to the loaded XML file containing the SyS-T catalog.

Intel® Debug Extensions for WinDbg

- Fixed exception on command `forensic.image_scan()` while the target is in hypervisor context.
- Intel® Debug Extensions for WinDbg command `breakin(VMMEXIT)` is not breaking in VM exit.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed total capture time value reported in VCD field ‘\$comment Total Cycles’, incorrect total capture time value could cause the waveform presentation issues.
- Error messages reported during FPGA backup and update have been changed from generic OS errors to more meaningful ones. Now they are also suggesting possible solutions for the problems.
- Fixed UI issues with inconsistent context menu in Setup Signals, sometimes the list of menu items contained duplicated entries.
- Added reporting an error message in UI when user enters invalid SGPIO clock frequency.
- Added *Project Config* identification to platform.json file to inform user the about the file's purpose.

Intel® System Debugger 2022 NDA Update 2238

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Integrate Token Management into Intel® System Debugger

Depending on the hardware and firmware settings, Intel® hardware requires debugging and tracing features to be enabled before they can be used. The overall process of enabling a platform to debug is called platform provisioning.

Beginning with the Intel® Core™ processor code-named Alder Lake, Intel® System Debugger supports platform provisioning for client computing platforms using tokens. A token is a file which contains a configuration for the Intel® Converged Security and Management Engine (Intel® CSME) firmware running on the platform. If a token is valid and then accepted by the Intel® CSME firmware, its configuration defines which debug features will be enabled.

- New live target information view to show Intel® CSME firmware information

Add Target Information view to TCA's Connection Editor. This new view provides information about the Unlock state, the token acceptance and details like the firmware version and trace catalog version.

Intel® System Debugger - System Debug

- Support for platforms which have enabled and configured Intel Total Memory Encryption (TME) or Multi-Key Total Memory Encryption (MKTME)
- Support for Flexible Return and Event Delivery (FRED)

Intel® System Debugger - System Trace

- Preview of new Trace Message Viewer
- Support System Trace Project creation without connection configuration
- Improved Timeline Viewer annotation precision by supporting picosecond resolution

Intel® Debug Extensions for WinDbg

- Added support for Architectural LBRs in WinDbg over Intel® DCI Extensions.

Fixed Issues

Intel® System Debugger - System Trace

- Fixed Timeline Viewer zooming behavior

In some cases zooming was not possible because of incorrect mouse cursor focus. To fix this the mouse focus priority list was updated: Static marker, data point, cursor)

Intel® Debug Extensions for WinDbg

- Fixed issue with command !scanPreNT in WinDbg over Intel® DCI Extensions. WinDbg over Intel® DCI Extensions command !scanPreNT has been switched to !scanprent for compatibility with Windows* SDK versions older than 10.0.22621.0.

Intel® System Debugger 2022 NDA Update 2234

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Updated ISD to new Eclipse 2022-06 base
- Introduced new Desktop Shortcuts for Target Indicator and Intel® Debug Extensions for WinDbg NDA
- Provided semantic versioning for TCA LIB and TCA CLI, this allows users to clearly tell the version they are using and should simplify support.

Intel® System Debugger - System Debug

- Emit warnings when the user-provided source code and debug information does not match that associated with the executable, provided that the user's compiler and build tools support storing the necessary metadata used to detect such inconsistency.
- Simics® disassembly and source-level debugging support for VPU firmware, including SHAVE cores.

Intel® System Debugger - System Trace

- TraceCLI: Added usage of the TCA profile embedded into a trace capture file

Fixed Issues

Target Connection Assistant

- Fixed connect button logic. Use the connect toolbar button to show the current status of the selected connection.
- Removed second connect button from connection view. To not confuse users with the same button in multiple places the connect button has been removed from the connection view.
- Fix detection of MTL A2 package stepping.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed SGPIO signals did not show up in digital rule cells for and new rules can't be created for them
- Fixed GUI performance issue that caused delays when changing editor tabs
- Fixed GUI performance issue that caused disabling/enabling multiple rules to take very long

Intel® System Debugger 2022 NDA Update 2230

Check the summary below or see the [online version](#).

New Features

Intel® System Debugger - System Debug

- Last Branch Record (LBR) execution trace for Lakemont / Minuteman cores in Simics® simulation.

Intel® System Debugger - System Trace

- Enabled Ponte Vecchio tracing support for Intel® Graphics System Controller (Intel® GSC) and Intel® Chassis System Controller (Intel® CSC).
- Earlyboot: Show current trace cycle in Python* status bar.
- TraceCLI: Added target_status command to provide information about the currently configured target.

Intelligent Debug & Validation Tool (IDV Tool)

- Added warning message when using latest software, but FPGA programming is outdated.

Fixed Issues

Target Connection Assistant

- Show progress information while ongoing connection establishment.
- Fixed reset toolbar commands for Simics® simulation connection.
- Fixed possible hang if there is no probe connected while ongoing connection establishment.

Intel® System Debugger - System Debug

- Fixed conditions of IO access breakpoints. This fix enabled breaking on BIOS POST code changes.
- Proper display of power loss in the Debug view.
- Fixed some filtering issues in the Symbol Browser.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed incorrect app_sgpio_dir_ctrl (0x46) FPGA register programming.
- Fixed misleading error message verbiage “Error in serial communication while writing: Access is denied”.
- Fixed GUI sorting by multiple columns lasts performance issue.
- Fixed incorrect sending trigger config for disabled channels.
- Fixed IDV Tool hang when platform.json was changed and saved during capture.
- Fixed issue with rules order was always re-sorted by content in the first column after a save.

Intel® System Debugger 2022 NDA Update 2226

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Added an aligned Reset control which can be used from all tools within Intel® System Debugger without an active debug session.
- Implemented a new connect and disconnect button which toggles the state and uses more intuitive icons.
- Added filter for connected probes to manual connection configuration selection dialog.
- Increased Eclipse* Python* Shell command line buffer to minimum 5000 lines.

Intel® System Debugger - System Debug

- New System Debug CLI entry point “intel_sysdbg” is available. New System Debug CLI command “sysdbg.launch_and_connect()” is available to launch a System Debug instance and connect the CLI to it automatically from the Intel® System Debugger CLI.

Intel® System Debugger - System Trace

- Integrated the memory management library that would let the users limit the amount of memory decoding process uses.

Intel® Debug Extensions for WinDbg

- Added debug support for pre-NT environment.

Intelligent Debug & Validation Tool (IDV Tool)

- Added undo and redo capabilities to rule edit view.
- Added support for multiple rule notes.

Fixed Issues

Intel® System Debugger - System Debug

- Fixed issue with bitfield values of registers not showing correctly when the System Debug is connected to a Simics® simulator platform.
- Fixed issue with Memory View not being able to successfully write a value into physical memory.

Intel® System Debugger - System Trace

- Fixed reload button in Trace Capture view which reloads the target settings for System Trace UI.

- Fixed an issue where Message view was not showing filtered results when applied from Rules view.
- Fixed sporadic null pointer exception dialog during start trace.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed an issue where sending a manual start and stop triggers could break serial communication with FPGA.
- Fixed copy, paste, drag and drop issues of rules within groups. Only apply a change for the selected rules, not for the parental groups.

Intel® System Debugger 2022 NDA Update 2222

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Updated Intel® System Debugger to use latest Eclipse* IDE 2022-03 base.
- Improve documentation, use user guide in Intel® System Debugger Eclipse*

Intel® System Debugger - System Debug

- New Symbol Browser View.
- Add debug and breakpoint controls to Debug & Trace perspective

Intel® System Debugger - System Trace

- Capture OpenIPC logs with TraceCLI early boot script

Intelligent Debug & Validation Tool (IDV Tool)

- Added signal pin remapping dialog window.

Debug GdbServer* proxy

- Added support for Zephyr 2.7 based Intel® Programmable Services Engine (Intel® PSE) SDK.

Fixed Issues

Target Connection Assistant

- Fixed out of sync ipccli in ISD shell after ipc.forcereconfig() command
- Fixed stack trace caused by file not found on startup of Eclipse
- Fixed shutdown process of ISD CLI in eclipse, should not set error code anymore
- Fixed OpenIPC selection on re-usage of existing workspace, prevent accidental storing of default IPC provider path

Intel® System Debugger - System Trace

- Fixed trace support on Elkhart Lake
- Reduced package loss issues related to high bandwidth tracing
- Fixed capture view button behavior in case of missing project

Intel® Debug Extensions for WinDbg

- Removed deprecated command `forensic.get_kernel_base()`.

Intelligent Debug & Validation Tool (IDV Tool)

- Use unique GPIO names (CLK, LOAD_N, DOUT, DIN) for pins 37 to 40.
- Fixed GPIO signals in “Setup Signals” tab can not be enabled/disabled as a group.
- Fixed sporadic hang of IDV Tool when no trigger conditions are defined and capturing is stopped manually.

Intel® System Debugger 2022 NDA Update 2214

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Added remote debug support for hardware platforms. The connection flow in GUI and CLI is updated to allow you debug a hardware platform connected to a remote host. Refer to the Get Started chapter in documentation for step-by-step instructions.

Intelligent Debug & Validation Tool (IDV Tool)

- Introduced advanced rule creation functionality. Now all rules are aggregated in a single Rule tab (instead of multiple tabs). Refer to Intel® System Debugger documentation for updated instructions on creating rules.

Fixed Issues

Intel® System Debugger - System Debug

- Fixed issues that caused crashes if Intel® System Debugger was disconnected right after connecting to a Simics® simulator.

Intel® Debug Extensions for WinDbg

- Fixed loading full kernel dump for Elkhart Lake platforms.
- Fixed the `forensic.get_kernel_base()` command performance.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed duplicated rule IDs

- Enabled creating alias names for rules.

Intel® System Debugger 2022 NDA Update 2210

Check the summary below or see the [online version](#).

New Features

Intel® System Debugger - System Debug

- Enabled tracing of `ptwrite` instruction and power events via Instruction Trace view. See documentation for details.

Intel® Debug Extensions for WinDbg

- Integrated Target Connection Assistant into `ExdiServer` console.

Fixed Issues

Intel® System Debugger - System Debug

- Fixed experience of adding symbols for debugging target simulation. Previously, adding symbol files right after Simics® simulator launch could lead to Intel® System Debugger crash.

Intel® System Debugger - System Trace

- Fixed drag and drop of rules in the Rules vies. Previously, it could throw an exception and cause rules to disappear.

Intel® Debug Extensions for WinDbg

- Fixed performance of `!ipi` and `!cpuinfo` commands, which could not work on specific platforms with WinDbg over Intel® Direct Connect Interface (Intel® DCI)
- Fixed issues with launching WinDbg from Eclipse* IDE Tools menu.

Intelligent Debug & Validation Tool (IDV Tool)

- Fixed expanded console summary to show correct SGPIO total samples captured.
- Fixed file saving performance. Previously, Intelligent Debug & Validation Tool (IDV Tool) could crash during this operation.
- Fixed serialization of platform configuration. Previously, IDV Tool could accidentally overwrite platform configuration.
- Fixed information at IDV Tool splash screen.

Intel® System Debugger 2022 NDA Update 2206

Check the summary below or see the [online version](#).

New Features

Target Connection Assistant

- Updated the list of stepping for several platforms to include non-released stepping.
- Added new environment and launcher scripts. Old scripts will be removed with Intel System Debugger® 2210 release. Replace your script references as following:
 - `iss_env` to `isd_env`
 - `iss_ide_eclipse-launcher` to `isd_ide_launcher`
- Improved connection state, target state, and processor state handling for IPC connections. Added target condition information to the Connection view and `tca.status()` command result.
- Updated Intel® System Debugger to use latest Eclipse* IDE 2021-12 base. Updated `io.grpc(1.43.1)` and `io.netty(4.1.72)` third-party libraries, and removed log4j dependency from the Intel® System Debugger.

Intel® System Debugger - System Debug

- Enabled Simics® simulation support for Grand Ridge CPU / Emmitsburg PCH and Lunar Lake / Lunar Lake SoC M targets.

Intel® Debug Extensions for WinDbg

- Upgraded the BSOD bug check database to version v1.6.4

Intelligent Debug & Validation Tool (IDV Tool)

- Enabled toggling of the rule-checking for historical (pre- and posttrigger) captures.

Fixed Issues

Target Connection Assistant

- Fixed issues with the Simics® Simulator Log view, which previously could return `Failed to parse event, invalid arguments` error messages.
- Fixed workspace path selection issues on Windows*.
- Fixed IPC warnings related to `Failed to get device configuration value` messages to only request minimum information required.

Intel® System Debugger - System Debug

- Fixed UEFI Tracker toggle for Simics® simulation targets.

Upgrade Simics® base package to version 6.0.115 and higher for better experience.

Intel® System Debugger - System Trace

- Fixed version mismatch issue between Trace Python* module and target support packs, which was reported in the console but did not affect debugging.

- Fixed `NullPointerException` errors reported in the console.
- Fixed issues with the Console view showing only one `[INFO]` message when launching Eclipse* from a workspace that already contained a System Trace project.

Intelligent Debug & Validation Tool (IDV Tool)

- Improved project wizard and fixed problem with copy and paste projects.

Known Issues

Cannot launch Target Indicator, getting error about missing VCRUNTIME140_1.dll / Cannot do any tracing with “Failed to load platform” error (Windows* host only)

Issue: This should not be an issue, but in case it appears - it means Microsoft* C++ Redistributables package is broken.

Workaround:

1. Go to “Add/Remove Programs”.
2. Uninstall all “Microsoft C++ Redistributables”.
3. Uninstall Intel® System Debugger.
4. Install Intel® System Debugger again.

Trace and debug operations on Alder Lake S require changes in platforms OS workaround settings

Issue: Power state transition (S5, S4, WR) with probe mode detection method TapPolling causes debug tool hang. Meanwhile, TapPolling is required for run control on targets with OS running. Starting with U2122 release, Intel® System Debugger NDA enables you to manage probe mode detection logic based on use cases.

Workaround: Follow instructions at `<install_dir>/documentation/tca/usage/connecting_gui.html`.

Depending on your goals, complete step 7 as follows:

- For BIOS debugging or FW/SW tracing, select “Target runs no OS” (Default). Debugging and tracing on platform that does power transitions requires this option selected.
- For working with Architectural Event Traces (AET) or WinDbg over Intel® Direct Connect Interface (Intel® DCI), select “Target runs an OS”.

Disconnecting and reconnecting Intel System Debugger to Alder Lake S target with survivability flag enabled causes NorthPeak (NPK) hardware hang

Issue: During NPK tool disconnect and reconnect, NPK IP is power gated and then power un-gated and the default register value causes timeout and hang.

Workaround: Disable survivability mode if debugging across low power states is not required.

Low power debug is not supported for Alder Lake P due to missing Low Voltage Margining (LVM) overrides

Issue: Alder Lake P Best Known Configuration (BKC) does not support low power.

Workaround: N/A

Unicode error returned during target selection via console

Issue: When selecting a target with a command like `trace.interactive_setup()`, an error like `UnicodeEncodeError: 'latin-1' codec can't encode character 'u2122' in position 24: ordinal not in range(256)` is displayed.

Workaround: Before starting the console session, execute the following command:

- Windows: `set PYTHONIOENCODING=UTF8`
- Linux: `export PYTHONIOENCODING=UTF8`

Cannot connect Intel® System Debugger to Alder Lake P target with USB3 Debug Class (DbC)

Issue: Debug connection is not working with USB3 DbC connected to Alder Lake P. Reset break is also not functional.

Workaround: There are two ways to resolve connectivity issues with Intel System Debugger if USB3 DbC is detected:

- In the final step of establishing target connection (“Connection Summary” dialog box), expand the “Platform Debug Configuration” tab and uncheck the “Configure High Bandwidth Transport” box.
- Modify IFWI via Intel® Flash Image Tool (Intel® FIT) to enable USB2 port and avoid USB3 DbC detection.

Cannot establish USB DbC connection to debug the 10th generation Intel® NUC with Alder Lake M or Alder Lake P

Issue: When attempting to connect to the target using USB DbC, both Target Indicator and Windows* Device Manager report errors on the USB device (USB DbC probe).

Workaround: N/A

Target Indicator is not visible when being started on Linux* hosts with GNOME 3.26 and above (Linux* host only)

Issue: On Linux* hosts with GNOME 3.26 and above, the Target Indicator is not visible when being started because GNOME 3.26 removes the system tray that the Target Indicator heavily relies upon.

Workaround: You can start the Target Indicator executable with the `--window-only` (or short `-w`) flag, which makes the Target Indicator avoid using the system tray.

Cannot start the Target Indicator from a file explorer with older package versions (Linux* host only)

Issue: Starting the Target Indicator executable from a file explorer (for example, Nautilus) fails, since it is not recognized as an executable file.

Workaround: Upgrade the “file” package to a version higher than 5.36. Start the Target Indicator from the command line or via the provided application link (the package upgrade does not affect it). See the original [bug report](#).

Reset causes `libusb` error and loss of run control on Linux* host

Issue : Warm or cold reset of the target connected to Comet Lake CPU/Comet Lake PCH-H (CML-S) in user interface or command line (ipcccli) causes `libusb` error and loss of run control. Cold reset is necessary to connect to the target again.

Workaround: Use the Intel® DCI USB DbC2 instead of USB DbC3 on the port used for the debug connection. You can configure this in the Intel® Flash Image Tool (Intel® FIT)

1. Launch the Intel® FIT (obtained from Intel® Validation Internet Portal) and change the following settings within your current configuration:

- Debug > Direct Connect Interface Configuration > Direct Connect Interface (DCI) Enabled > Yes
- Debug > Early USB DBC over Type-A Configuration > USB2 DbC port enable > USB2 Port <X>

where X identifies the port number. Depending on the target, set the following value:

- 1 - to enable USB2 DbC on the vertical USB type-A port on CML-U DDR4 boards
- 3 or 4 - to enable USB2 DbC on either of the two stacked USB type-A ports respectively on CML-U LP3 boards
- 8 - to enable USB2 DbC on the vertical USB type-A port on CML-S boards with CMP-H or CNP-H chipset.

2. Build the image and flash the target with the new image.

Disconnecting a probe crashes the target (Windows* host only)

Issue: Intel® Silicon View Technology (Intel® SVT) Closed Chassis Adapter (CCA) and Intel® In-Target Probe (Intel® ITP) - XDP3 probes are supporting hot plug/unplug from a target, whereas for Intel® DCI DbC the connection is bidirectional. If a probe connection is lost, the debugger reports a Power Loss event.

Workaround: If the target was running, reconnect the probe (the disconnection has no effect on the running target). If the target was halted, reboot the target and restart the debugging session (the debug context is lost).

Platform security policy might inhibit debugger operation

Issue: In some platforms, the security policy might disable JTAG access to the CPU. This is intended to prevent reverse-engineering. In this case, the Intel® System Debugger will successfully connect to the target, but will not be able to discover any CPUs on the JTAG bus and will report that no CPU is available.

Workaround: Ensure that that platform firmware has enabled access to the CPUs via JTAG. You can do it by flashing a special “debug” firmware into the target. In some cases the CPU or the CPU module might physically disable JTAG access, especially in production or near-production versions. In this case, please work with the platform business unit to obtain a JTAG-enabled hardware.

Issue: When you attempting to archive Intel® DAL logs using the commands

`itp.daldebugloggerlevel("default")` followed by `itp.loggerarchive()`, a “Permission Denied” error is printed.

Workaround: Before every execution of the commands above, navigate to `<install-dir>\tools\DAL_<version>` and modify the `PublicLogging.log4net` file to replace “DALBase.log” with a location where you have write permissions (for example, `C:\Temp\DALBase.log`).

Cannot connect Intel® System Debugger to Raptor Lake CPU / Alder Lake PCH-S target with Intel® In-Target Probe (Intel® ITP) - XDP3

Issue: Debug connection is not working with Intel® ITP XPD3 connected to Raptor Lake CPU / Alder Lake PCH-S.

Workaround: Use a different debug probe (see Supported Platforms and Probes for available connection options).

Reset Break issues on Granite Rapids and Meteor Lake

Issue: Reset break does not work on properly Granite Rapids and Meteor Lake platforms.

Planting and hitting reset break does not work reliable. Once it may work, other times it may fail sporadically to halt all cores.

Workaround:

- Meteor Lake users can try to use Intel® System Debugger 2246.
- For Granite Rapids there is currently no workaround.

Resume or step from Software Breakpoint issue on Granite Rapids D

Issue: Resume or step from Software Breakpoint issue on Granite Rapids D does not work properly. An attempt to resume execution or perform a step on a target halted in a Software Breaks does not work properly an can lead to a MCA exception.

Workaround: Use should use Hardware Breakpoints instead of Software Breakpoint.

Target Connection Assistant

Sapphire Rapids 4-socket modular system is not detected automatically

Issue: Sapphire Rapids 4-socket system is not automatically detected by the Target Connection Assistant.

Workaround:

1. Launch Intel® System Debugger, start establishing target connection, and choose “Manually select target” (for instructions, see the Intel System Debugger documentation).
2. In the final step of the connection wizard, click the gear icon and update the new OpenIPC /Provider Configuration / Configuration Name to “SPR_EBG_XDPA_Modular_4Mhz”

Intel® System Debugger - System Debug

Intel® System Debugger: Debugging in SEAM root mode is known to currently be unstable

Issue: Debugging in SEAM root mode is known to currently be unstable

Workaround: N/A

Intel® System Debugger: TD guest debugging might require the target to be Red unlocked

Issue: TD guest debugging might require the target to be Red unlocked

Workaround: N/A

Intel® System Debugger - System Trace

Rocky Linux 8 Support

Issue: Trace does not support Rocky Linux 8 because of GLIBC version mismatch (/lib/x86_64-linux-gnu/libm.so.6: version `GLIBC_2.29 not found).

Workaround: Update to Rocky Linux 9

CentOS* 7 and OpenSuse* 15 host OS are not supported (Linux* host only)

Issue: CentOS* 7 and OpenSuse* 15 host OS are not supported for System Trace use cases.

Workaround: N/A

Various error messages referring to Interview_Decoder.dll or visa64.dll are displayed (Windows* host only)

Issue: If the host system has IVI tools installed, the tool erroneously tries to load visa64.dll from C:\Windows\System32 or Interview_Decoder.dll from <install-dir>\system_debugger_2020_ndasystem_tracebin.

Workaround: Do any of the following:

- Temporarily remove `C:\Windows\System32\visa64.dll` or uninstall the IVI tools.
- Check the information on the [Dynamic-Link Library Search Order](#) but be aware of the potential security implications.

On Discrete Graphic platforms (DG1/DG2), errors occur during connection attempt of trace capture start

Issue: Due to security policies in DG1 and DG2, access to North Peak Test Access Port over Lauterbach Processor Trace Interface connection is blocked. An error message “Failed Intel® Trace Hub hardware detection check!” appears.

Workaround: Currently, you can configure DG1/DG2 tracing only through firmware running on the target. To enable this tracing capability, launch Intel® Flash Image Tool (Intel® FIT), open Debug Settings, and configure the “Emergency mode”. To start tracing, configure the firmware as described above, connect the debug probe, deselect “Configure target during capture start and stop”, and press the “Start Capture” button in the Trace Tool. The error message “Error in connecting to the target. Target does not have power.” will appear but you can safely ignore it.

Issue: Using TraceCLI to capture traces fails with “Unknown mode specified, falling back to port list” error message (<https://jira.devtools.intel.com/browse/DEBGR-11176>).

Workaround: Use Eclipse-based System Trace instead of TraceCLI.

Issues when using CCA 2wire for trace streaming

Issue: Connection get lost or invalid/no trace packets are received when using CCA 2wire while trace streaming is active.

Workaround: Use a different probe for trace streaming or use MTB as trace destination

Intel® Debug Extensions for WinDbg

Compatibility issues of Intel® Debug Extensions for WinDbg for Windows* 10 SDK

Issue: Windbg over Intel® DCI plugin is not compatible for Windows* SDK versions 1809 (10.0.17763) and 1903 (10.0.18362.1).

Workaround: Install the latest Windows* SDK version 22000.

Latest Windows* 11 SDK is not able to download symbols.

Issue: Windbg over Intel® DCI plugin is not able to download symbols with Windows* 11 SDK versions 22H2 (10.0.22621.0).

Workaround: Install previous Windows* SDK version 22000 (10.0.22000.194), or latest version (build 25197 or newer) from Windows Insider Preview Builds.

Error collecting full memory dump in SPR targets

Issue: Windbg over Intel® DCI plugin is not able to collect full memory dump in SPR.

Intelligent Debug & Validation Tool (IDV Tool)

Captures are reported as starting later than the actual trigger

Issue: Sporadically captures are reported as starting later than the actual trigger.

Workaround: Trigger the same capture again.

Time difference between Analog and Digital samples

Issue: When capturing any channel as digital and analog a time difference between the two can be observed.

Workaround: None

Explanation: The analog values get sampled at a much lower speed and have to be transferred from the ADC to the IDV tool serially. Therefore an analog sample may have to wait n cycles, with n being the number of active analog channels up to a maximum of 16.

Stop trigger does not work when manual start trigger is used.

Issue: When capturing signals the stop trigger condition will not work if a manual start trigger was used.

Workaround: Configure a start trigger condition.

Supported Platforms and Probes

Each Intel® System Debugger tool supports its own set of platforms and probes. The following probes are commonly used for connection:

- Intel® In-Target Probe (Intel® ITP) - XDP3
 - ITP-XDP3* in the table.
- Intel® Direct Connect Interface (Intel® DCI) Out of Band (OOB) via Intel® Silicon View Technology (Intel® SVT) Closed Chassis Adapter (CCA).
 - CCA (DCI OOB)* in the table
- Intel® DCI 2-wire via Intel® SVT Closed Chassis Adapter (CCA).
 - CCA (DCI 2-wire)* in the table.
- Intel® SVT Debug Class (DbC) USB Debug Cable.
 - DbC* in the table.
- Intel® DCI OOB via Intel® SVT Closed Chassis Adapter (CCA) + Intel® DCI USB Debug Class (DbC). This method suggests simultaneous use of a DbC cable and a CCA adapter.
 - CCA + DbC* in the table.
- Simics® simulator – virtual target simulation.

! See also

Debug and Trace

The table below provides an overview of target/probe combinations supported by System Debug and System Trace components of the Intel® System Debugger NDA in the present release.

Note

The core count of the target platform component does not affect connection and debugging experience. If you face any issues, refer to the [Known Issues section](#) of Release Notes or contact Customer Support.

Target Components	Connection Methods: System Debug
Ash Creek Falls CPU / Lewisburg PCH	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Alder Lake CPU / Alder Lake PCH-N	ITP-XDP3, CCA (DCI 2-wire), DbC, Simics® simulator
Alder Lake CPU / Alder Lake PCH-P	ITP-XDP3, CCA (DCI OOB), CCA (DCI 2-wire), DbC, Simics® simulator
Alder Lake CPU / Alder Lake PCH-S	ITP-XDP3, CCA (DCI 2-wire), DbC, Simics® simulator
Arctic Sound GPU	Simics® simulator
Arrow Lake CPU / Meteor Lake PCH-S	ITP-XDP3, CCA (DCI 2-wire), DbC, Simics® simulator
Bay Trail SOC (MinnowBoard MAX*, Valleyview2)	ITP-XDP3, Simics® simulator
Broxton I SOC (Apollo Lake)	DbC, Simics® simulator
Broxton P SOC (Apollo Lake)	DbC, Simics® simulator
Broxton P SOC: Gordon Peak Modular Reference Board (GP MRB)	DbC, Simics® simulator
Cannon Lake CPU / Cannon Point PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Cascade Lake CPU / Kaby Lake PCH-H (Glacier Falls)	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Cascade Lake CPU / Lewisburg PCH (Purley Refresh)	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Coffee Lake CPU / Cannon Point PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Coffee Lake CPU / Kaby Lake PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Coffee Lake CPU / Cannon Point PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Comet Lake CPU / Comet Lake PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Comet Lake CPU / Cannon Point PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Simics® simulator
Comet Lake CPU / Comet Lake PCH-V	CCA (DCI OOB), DbC, Simics® simulator
Comet Lake CPU / Tiger Lake PCH-H (Rocket Lake S)	ITP-XDP3, CCA (DCI OOB), CCA (DCI 2-wire), DbC, Simics® simulator

Target Components	Connection Methods: System Debug
Comet Lake CPU / Comet Lake PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Cooper Lake CPU / Lewisburg PCH (Cedar Island)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Denverton SOC (Car Branch 1, Car Branch 2, Denverton)	DbC, Simics® simulator
DG1 PCI Express Card GPU	Simics® simulator
DG2 PCI Express Card GPU	Simics® simulator
Diamond Rapids CPU	Simics® simulator
Elkhart Lake CPU / Mule Creek Canyon PCH	CCA (DCI OOB), CCA (DCI 2-wire), D
Emerald Rapids CPU / Emmitsburg PCH (Eagle Stream Refresh)	ITP-XDP3, CCA (DCI OOB), CCA (DC
Gemini Lake SOC	DbC, Simics® simulator
Gemini Lake Refresh SOC (Gemini Lake +)	DbC, Simics® simulator
Grand Ridge CPU / Emmitsburg PCH	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Granite Rapids CPU	ITP-XDP3, Simics® simulator
Granite Rapids D (Kaseyville) CPU	ITP-XDP3, Simics® simulator
Ice Lake CPU / Cedar Fork PCH	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Ice Lake CPU / Lewisburg PCH (Whitley)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Ice Lake CPU / Ice Lake PCH-LP	CCA (DCI OOB), CCA (DCI 2-wire), D
Ice Lake CPU / Ice Lake PCH-N	CCA (DCI OOB), CCA (DCI 2-wire), D
Jasper Lake CPU / Jasper Lake PCH-N (Jasper Lake +)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Kaby Lake CPU / Kaby Lake PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Kaby Lake CPU / Sunrise Point PCH-H	ITP-XDP3, CCA (DCI OOB), Simics®
Kaby Lake CPU / Sunrise Point PCH-LP	CCA (DCI OOB), DbC, Simics® simul.
Lakefield SOC	CCA (DCI OOB), CCA (DCI 2-wire), D
Lunar Lake CPU / Lunar Lake SoC M	ITP-XDP3, CCA (DCI 2-wire), DbC, Si
Meteor Lake CPU / Meteor Lake PCH-S	ITP-XDP3, CCA (DCI 2-wire), DbC, Si
Raptor Lake CPU / Alder Lake PCH-S	ITP-XDP3, CCA (DCI OOB), CCA (DC
Rocket Lake CPU / Comet Lake PCH-H	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Rocket Lake CPU / Tiger Lake PCH-H (Tatlow)	ITP-XDP3, CCA (DCI OOB), CCA (DC
Sapphire Rapids CPU / Alder Lake PCH-S (Fishhawk Falls)	ITP-XDP3, CCA (DCI OOB), CCA (DC
Sapphire Rapids CPU / Emmitsburg PCH (Eagle Stream)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Sierra Forest CPU	ITP-XDP3, Simics® simulator

Target Components	Connection Methods: System Debug
Sierra Forest CPU / Emmitsburg PCH	ITP-XDP3, Simics® simulator
Skylake CPU / Sunrise Point PCH-H	ITP-XDP3, CCA (DCI OOB), Simics®
Skylake Server CPU / Kaby Lake PCH-H (Basin Falls)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Skylake Server CPU / Lewisburg PCH (Anderson Creek, Purley)	ITP-XDP3, CCA (DCI OOB), CCA+Db
Skylake CPU / Comet Lake PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Skylake CPU / Sunrise Point PCH-LP	ITP-XDP3, CCA (DCI OOB), Simics®
Snow Ridge CPU / Cedar Fork PCH (Jacobsville)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Spring Hill CPU / Ice Lake PCH-LP	CCA (DCI OOB), CCA (DCI 2-wire), D
Tiger Lake CPU / Tiger Lake PCH-H	CCA (DCI OOB), CCA (DCI 2-wire), D
Tiger Lake CPU / Tiger Lake PCH-LP	CCA (DCI OOB), CCA (DCI 2-wire), D
Tunnel Creek SOC (Lincroft)	ITP-XDP3, Simics® simulator
Whiskey Lake CPU / Comet Lake PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Whiskey Lake CPU / Cannon Point PCH-LP	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Whiskey Lake CPU / Sunrise Point PCH-LP (Amber Lake Y 4+2)	ITP-XDP3, CCA (DCI OOB), DbC, Sir
Ponte Vecchio (PVC) PCI Express Card GPU	
Battlemage	
Panther Lake	Simics® simulator
Clearwater Forest	Simics® simulator
Granite Rapids Workstation (Meteor Lake PCH)	Simics® simulator

Crash Log

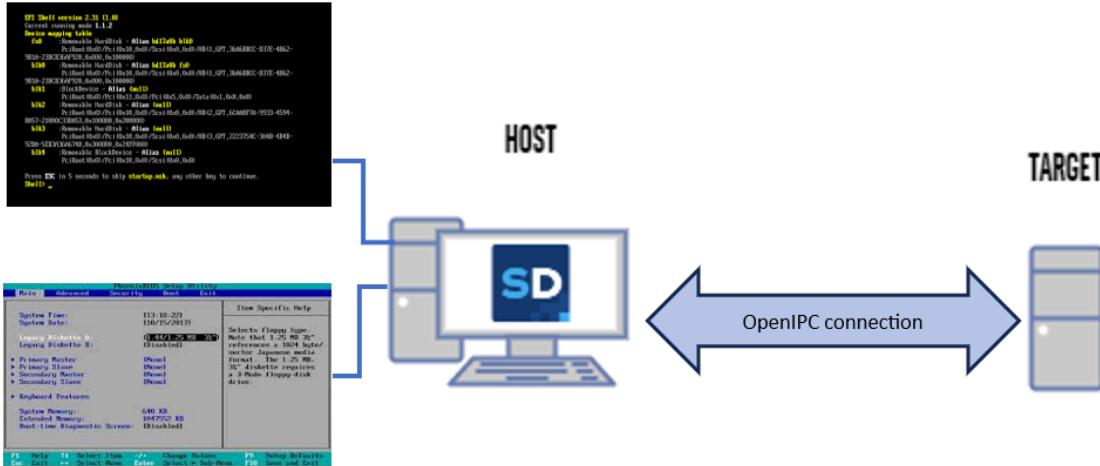
Intel® System Debugger - Crash Log supports the following target components in the present release:

- Alder Lake CPU (variants: M, N, P, S)
- Alder Lake PCH (variants: N, P, S)
- Broxton I, Broxton P
- Comet Lake PCH (variants: H, LP)
- Cannon Point PCH (variants: H, LP)
- Emmitsburg PCH
- Ice Lake CPU (variants: H, I, S3, S4, UN/YN)
- Ice Lake PCH (variants: H, LP, N)
- Jasper Lake CPU
- Jasper Lake PCH-N

- Lake Field CPU (base die and compute die)
- Meteor Lake CPU / Meteor Lake PCH-S
- Mule Creek Canyon PCH
- Raptor Lake CPU
- Rocket Lake (variants: S, U)
- Sapphire Rapids SP CPU
- Tiger Lake CPU (variants: H, R, S, UN/YN)
- Tiger Lake PCH (variants, H, K, LP)

Virtual UART over Intel® DCI

A typical debug connection of ISD is established using the OpenIPC over Intel® DCI (Direct Connection Interface)-based debug cable. Not only OpenIPC connections but many BIOS developers are also using UART connections to run UEFI shell or change BIOS settings with the BIOS setup menu. The virtual UART feature emulates this UART connection over an OpenIPC connection.



Prerequisites:

- The BIOS has DCI-IO driver for redirecting UART messages over Intel® DCI(Direct Connection Interface) connection.

Note

The DCI driver is included in Intel® reference BIOS of Panther Lake platform.

- Terminal emulator which opens the TCP port to communicate with ISD and renders UART message with “Telnet” protocol.

Virtual UART Usage

1. Launch ISD tool with an Intel® DCI debug connection
2. Initiate virtual UART on the **Console** with the following commands:

```
cons = tca.TcpConsole(8,4, "localhost", "8282")
cons.start_async()
```

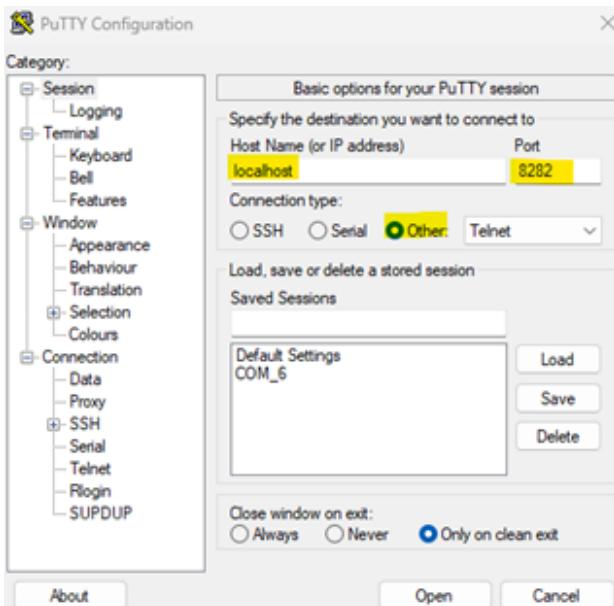
! Note

Parameters of “tca.TcpConsole”:

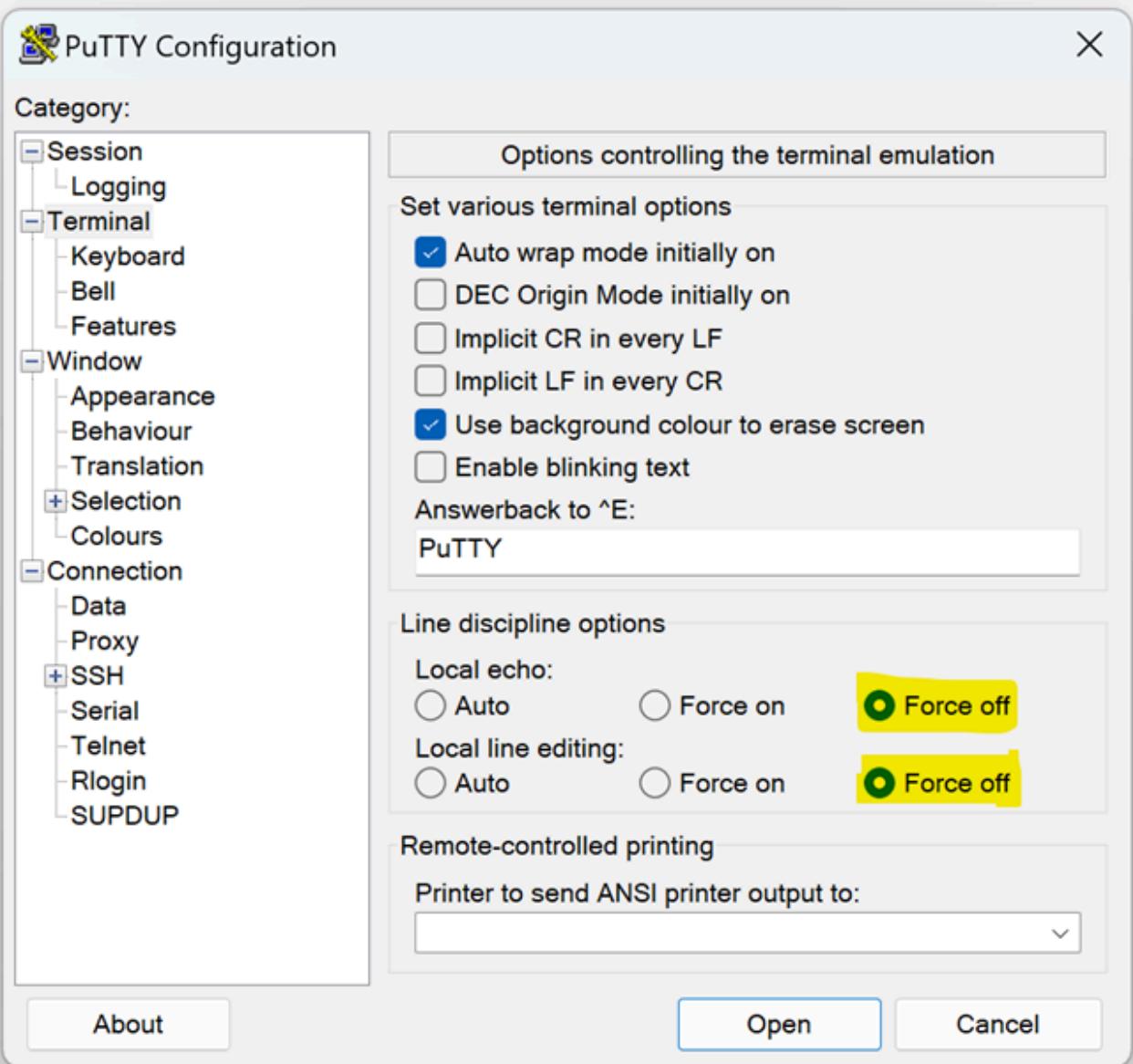
- 1st and 2nd parameter: read & write mailbox of the OpenIPC
- 3rd : IP address of debug host or use “localhost”
- 4th: TCP port number

3. Run the terminal emulator (e.g. Putty)

- Give the **localhost** in “Host Name” and **8282** as the port number
- Select a **Telnet** as the “Connection Type”

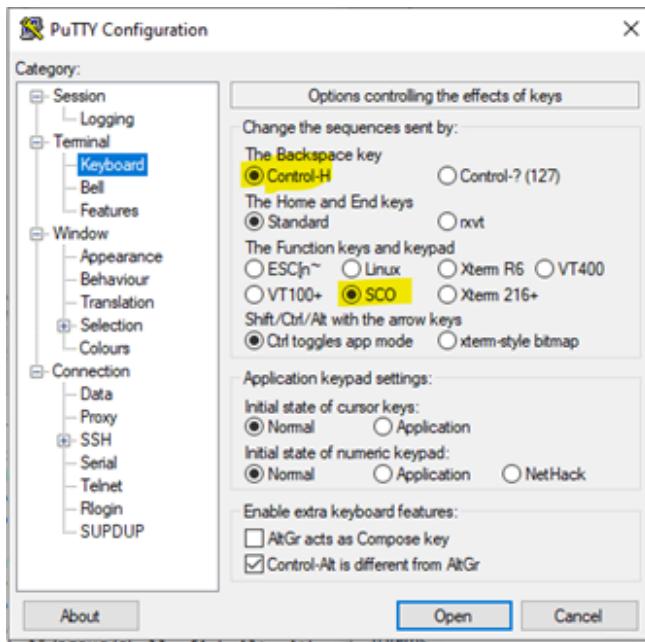


4. Open the “Options controlling the terminal emulation” under the “Terminal” category and select the **Force off** for the “Line discipline” of both local echo & line editing



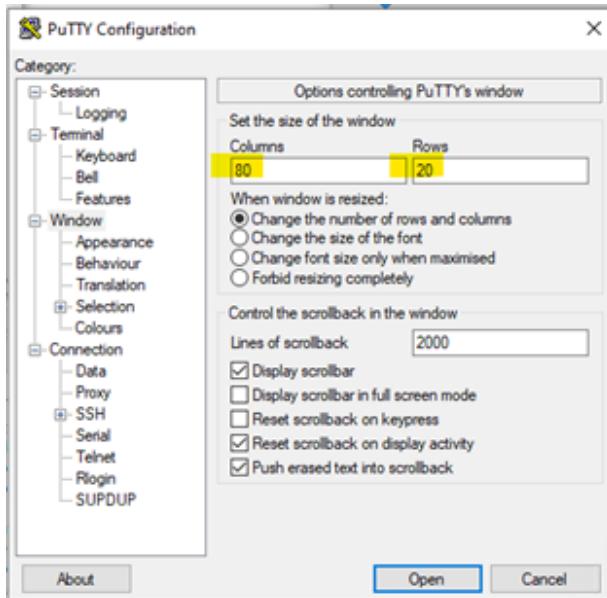
5. To use special characters on the terminal emulator:

- Open the “Keyboard” option under the “Terminal” group
- Select **Control-H** for the “Backspace key”
- Select **sco** for the “Function keys and keypad”



6. Open the “Window” to adjust the column of the emulator

- Change the “Columns” field to **80** and “Rows” to **20** (It is default for BIOS setup in a screen and please adjust those values according to your BIOS screen resolution)



7. Select “Open” at the bottom of the configuration and use virtual UART with Putty terminal

- The following screenshot is a typical view of “EFI Shell” over virtual UART

```

Shell>
Shell>
Shell>
Shell> ver
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (Intel, 0x00010000)
Shell> time
01:32:52 (LOCAL)
Shell> 

```

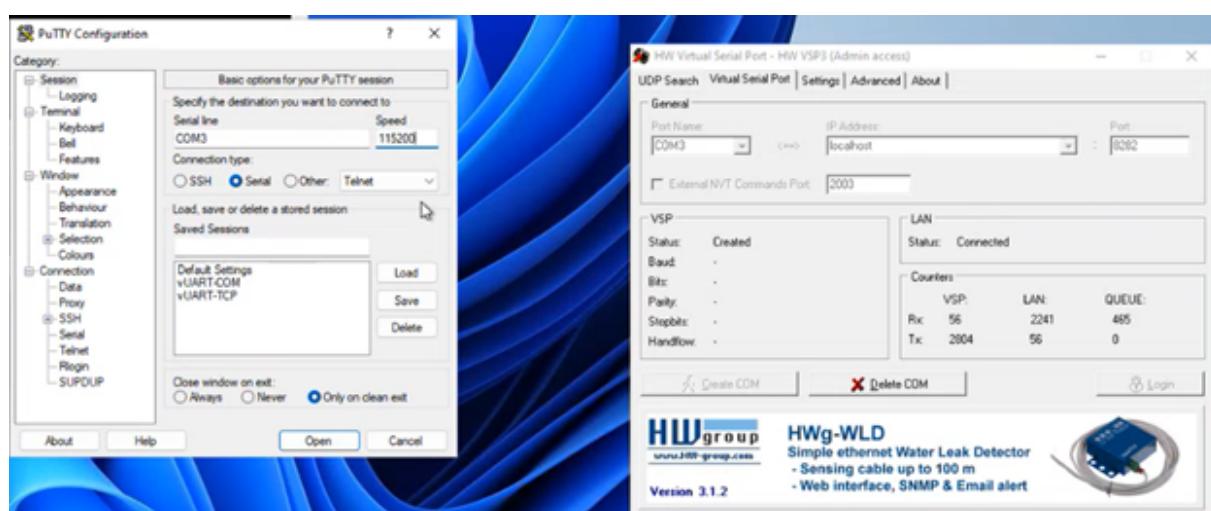
Note

User can use another terminal emulator that would open the specific TCP port. Please check if the BIOS has a DCI-IO driver which conveys UART message over Intel® DCI.

COM Port Interface for Virtual UART

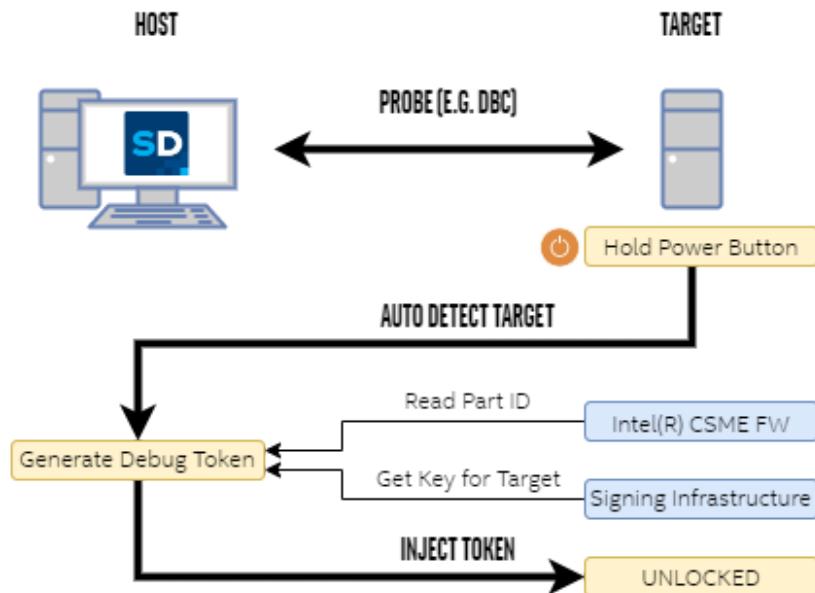
Most users are using a COM port to see BIOS setup menu or EFI shell. A user may want to use the same way to communicate with the target, then a virtual COM application will be another selection to utilize virtual UART over an OpenIPC connection such as **HW Virtual Serial Port**, **com0com** or **socat**.

The following example demonstrates how to configure a virtual COM port to bridge TCP connection to a regular COM port. This kind of 3rd party tool enables a user to use a serial port to communicate with target in the same way:



Platform Provisioning

Depending on the hardware and firmware settings, Intel® hardware requires debugging and tracing features to be enabled before they can be used. The process of enabling a platform for debugging is called platform provisioning.



Beginning with the Intel® Core™ processor code-named Alder Lake, Intel® System Debugger supports platform provisioning for client computing platforms using tokens. A token is a file which contains a configuration for the Intel® Converged Security and Management Engine (Intel® CSME) firmware running on the platform. If a token is valid and then accepted by the Intel® CSME firmware, its configuration defines which debug features will be enabled.

Not all platform and use cases require a token. The following documentation applies if your platform specific debug guide asks you to generate and provide a token.

Prerequisites:

- The platform has to be enabled for Intel® Direct Connect Interface (Intel® DCI) and connected to the host.

Currently Intel® System Debugger only supports token provisioning using Intel® DCI. If a Intel® DCI connection is not possible, check the *Tokens Guide* provided by the Intel® CSME firmware kit for other methods of token injection. These other methods will be enabled in future releases of Intel® System Debugger.

- An authorized private key or a customer specific signing infrastructure to sign the token.

Check the *Signing and Manifesting User Guide* and the *Tokens Guide* for more information about how to generate an own private key and include it into the firmware. These documents are available in the Intel® CSME firmware kit.

! Note

Intel® System Debugger needs to communicate with the Intel® CSME firmware to read the Part ID and inject tokens. These features are only available if the firmware is up and running, which means the provisioning feature over Hotham is practically available after the CPU gets out of reset.

Debug Token Configuration

Based on the platform and token type, different configuration options are available.

Flags

Flags define the acceptance behavior of a token.

- Globally valid

Disables the validation of the token's part ID(s). The token can be applied to any platform.

! Note

Globally valid feature is only applicable to OEM tokens.

- No anti-replay

Disables the validation of the part ID's 'nonce' field. Every time the firmware generates a new part ID, the stored nonce field is updated. If no anti-replay has been set, a token is still valid even the nonce field on the platform has changed.

! Note

Anti-replay with Expiration is the most secure setting as it prohibits the same token from being re-used. A token with no anti-replay set is not invalidated by requesting a new Part ID.

- No expiration

The token does not expire.

! Note

Non expiration requires no anti-replay to be set as well.

Expiration

The time after a token expires in seconds beginning from the time the Part ID has been generated.

Part ID (a.k.a. Part Data or PID)

The Part ID is a function of a Platform identifier, nonce and time base. It defines where the token can be applied. Unless configured differently using flags, the Part ID is verified by the Intel® CSME firmware to ensure the token is (still) valid and has been created for the associated platform instance.

A Part ID is generated by the Intel® CSME Firmware on request. Intel® System Debugger supports to request the Part ID from the running platform's firmware via Intel® DCI. In future releases Intel® System Debugger plans to support HECL and Intel® Download and Execute (Intel® DnX) for this purposes as well. It is not supported to manually create Part IDs, but if available an existing Part ID can be reused and provided to the tool. Check the *Token's Guide* in Intel® CSME firmware kit on how to get Part ID, if not using the Intel® System debugger.

Unlock State of the System

The unlock state maps the hardware stored system debug state to a human-readable format. The detailed status can be obtained while being connected using `hotham.loader.dscp.get_dfx_state()`.

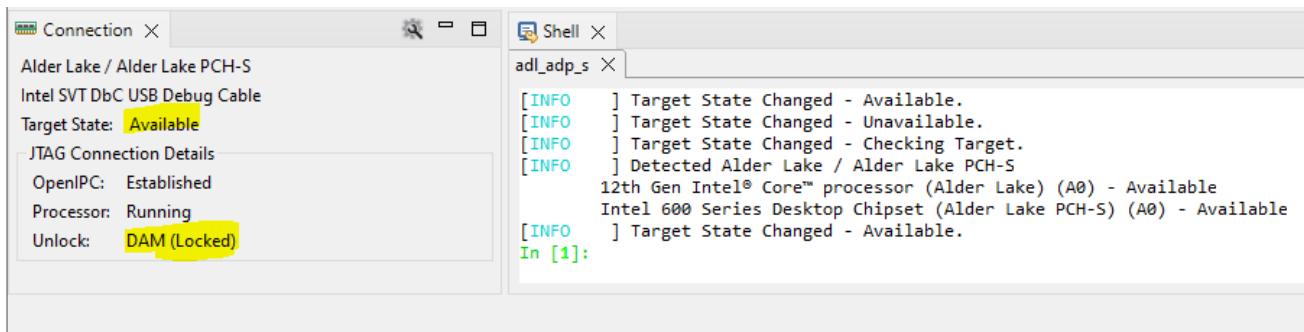
Unlock State Table

Name	Description
Security Locked	Security Locked, explicit consent may or may not be provided.
Functionality Locked	Exception/error mode, system is not supposed to boot.
Security Unlocked	Intel user in Manufacturing DLCS/Phase, requires Intel Red Unlocked password.
DAM (Locked)	Delayed Authentication Mode (DAM), allows unlocking at any time until next power cycle.
DAM (Unlocked)	Delayed Authentication Mode (DAM), currently unlocked.
Intel Unlock	Intel user in production DLCS/phase, requires Intel Red Unlock password. Requires Intel Red Unlocked password.
OEM Unlock	Debug policy for Intel OxMs. OEM password/debug token required.
EnDebug	Remote debug for Intel.

Debug Token usage in UI

Provisioning a target requires a functional connection. First setup Intel® System Debugger and connect to your platform using Intel® Silicon View Technology (Intel® SVT), DbC USB Debug Cable or Intel® DCI OOB via Intel® SVT Closed Chassis Adapter (CCA).

Before starting, check if your Target State shows 'Available' in the 'Connection' view. The 'Unlock' shows the current unlock state:



Connection X Shell X

Alder Lake / Alder Lake PCH-S

Intel SVT DbC USB Debug Cable

Target State: Available

JTAG Connection Details

OpenIPC: Established

Processor: Running

Unlock: DAM (Locked)

[INFO] Target State Changed - Available.

[INFO] Target State Changed - Unavailable.

[INFO] Target State Changed - Checking Target.

[INFO] Detected Alder Lake / Alder Lake PCH-S

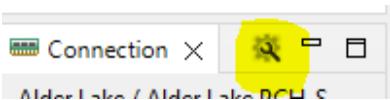
12th Gen Intel® Core™ processor (Alder Lake) (A0) - Available

Intel 600 Series Desktop Chipset (Alder Lake PCH-S) (A0) - Available

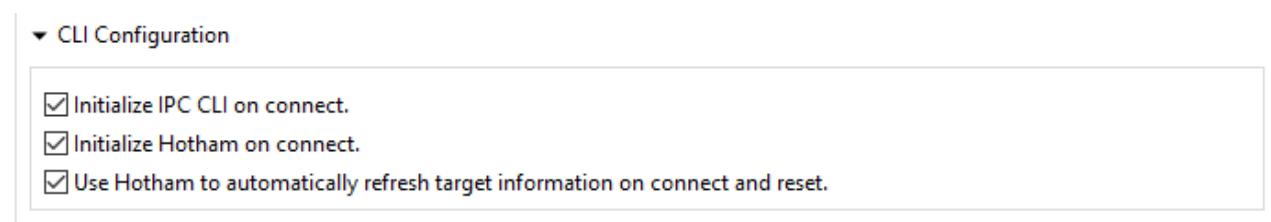
[INFO] Target State Changed - Available.

In [1]:

To interact with the Intel® CSME firmware and generate and manage the applied token, open the Connection Editor



ensure all required CLI options are enabled (checked):



▼ CLI Configuration

Initialize IPC CLI on connect.

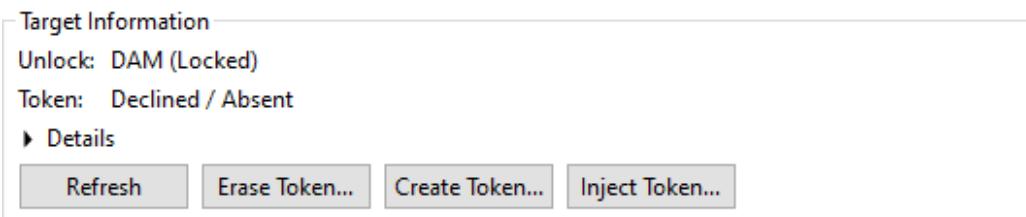
Initialize Hotham on connect.

Use Hotham to automatically refresh target information on connect and reset.

Note

If this is not the case, you need to disconnect, check the checkboxes, save the configuration and connect again.

and scroll down to the ‘Target Information’ box:



Target Information

Unlock: DAM (Locked)

Token: Declined / Absent

► Details

Refresh Erase Token... Create Token... Inject Token...

There you can:

- Refresh

To load the current unlock state and token consent from the target.

- Erase Token

To erase a currently applied token from a target.

⚠ Warning

Erasing the token will have impact on the ability to debug your target and cannot be reverted except by injecting a valid token again.

- Create Token

To create a new token.

- Inject Token

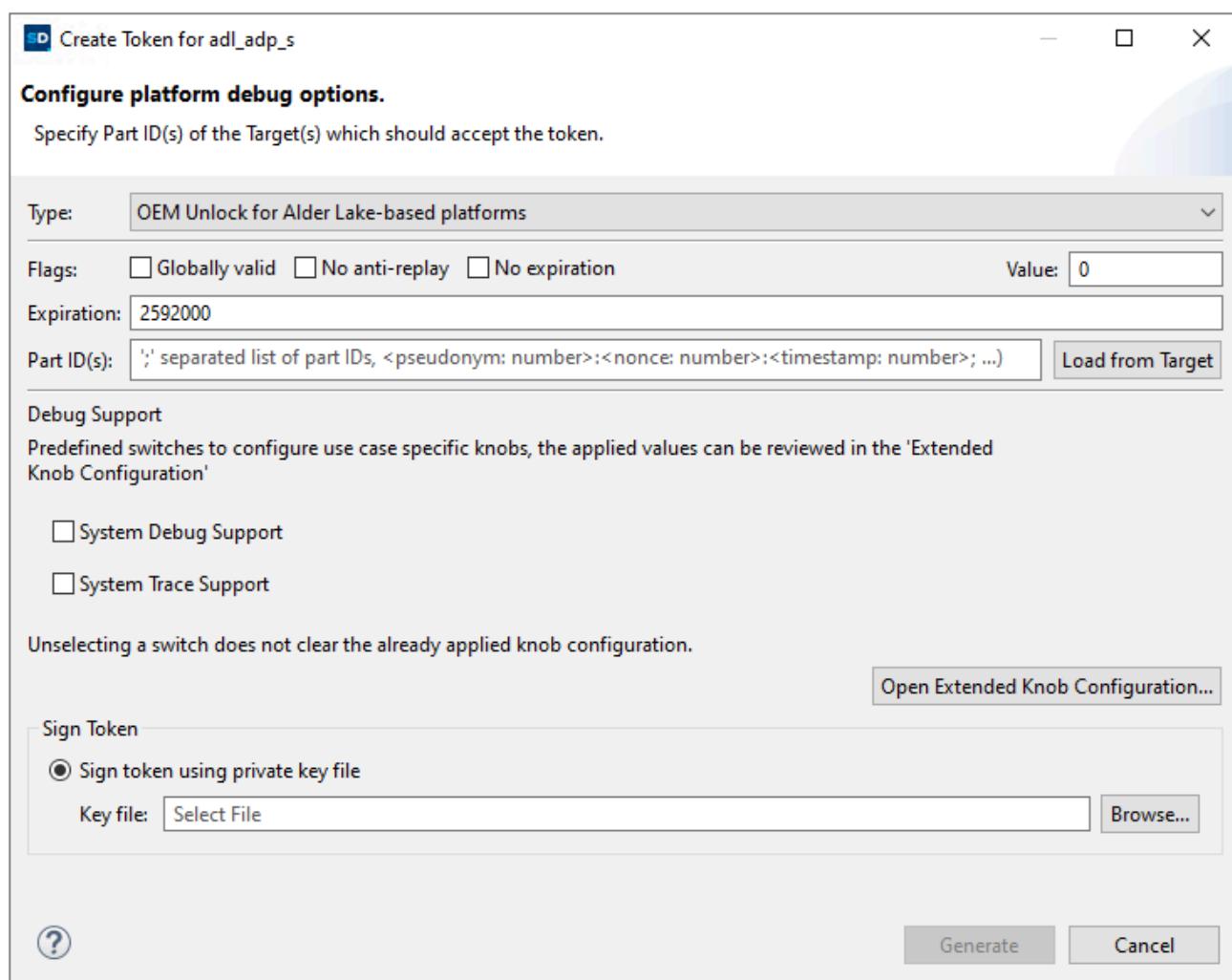
To update the token on the target.

⚠ Warning

Injecting a wrong token will have impact on the ability to debug your target and cannot be reverted except by injecting a valid token again.

Create Token

1. Click on *Create Token* to open the token generation dialog



2. Select the Type

The token type controls which configuration options and levels of debug supports are available. The token type must match how the token is signed, and an Intel token must be signed by Intel while the regular OEM token must be signed by the OEM.

3. Configure Flags

If required, use flags to configure the acceptance behavior of a token. No configured flags mean the best safety for the token. See [Flags](#).

4. Configure Expiration Time

Select the expiration time in seconds. The expiration time is bound to a Part ID and begins the moment the Part ID is generated.. See [Expiration](#).

5. Configure Part ID(s)

Provide your target's part ID. Click the 'Load from Target' button to load the Part ID from the currently connected hardware. Or use a Part ID received previously. To use multiple part IDs, enter them in the Part ID field separating them using a semicolon. See [Part ID \(a.k.a. Part Data or PID\)](#) for information on the format.

! Note

Requesting a new part ID might invalidate a currently applied token. If the current token has not been created with the 'No anti-replay' flag enabled it will be rejected on the next boot after the new Part ID has been generated.

6. Configure Debug Support

- System Debug Support, applies the knob configuration required to do system debugging, this includes to configure OEM Unlock or Intel Unlock (based on type) and to enable Run Control.
- System Trace Support, applies the knob configuration required to do system tracing, this includes to configure OEM Unlock or Intel Unlock (based on type) and to enable Run Control and enable CSE Trace messages.

! Note

Based on token type and platform, the knob configuration applied through a switch differs. To ensure the right knobs are configured, manually review the 'Extended Knob Configuration'.

7. Sign Token

Select your private key authorized by the firmware to sign the token

Check the *Signing and Manifesting User Guide* and the *Tokens Guide* for more information about: how to generate an own private key and include it into the firmware. These documents are available in the Intel® CSME firmware kit.

Inject Token

1. Click on *Inject Token*
2. Accept the disclaimer.
3. Select the token to be injected.

Debug Token usage CLI

Using Hotham and Debug Token Library (DTL)

- Debug Token Library (DTL) - Python* library to generate and sign Intel® CSME tokens.
- Hotham - Communication channel using the Debug Support Command Protocol (DSCP) on top of ipccli.

Start Python* environment

All the following instructions should be executed in Python, for convenience this documentation uses IPython*.

- Open a new command interpreter (cmd) or shell
- Source the <installation-directory>/isd_env.bat environment script
- Call ipython

Fetch Part ID from target

While the target is connected and available, Hotham can connect to the Intel® CSME firmware and read the part ID from the target.

```
from intel.hotham import hotham
import intel.dtl as dtl

# Load Part ID using Hotham
code, data = hotham.loader.dscp.get_part_id()
if code != 0:
    raise RuntimeError("Failed to read Part ID from target.")
pid_pseudonym = int(''.join(data['PART_ID'][::-1]), 16)
pid_nonce = data['NONCE']
pid_timestamp = data['TIMESTAMP']

# Create DTL part ID object
part_id = dtl.PartID(pid_pseudonym, pid_nonce, pid_timestamp)
```

Configure Token using presets

DTL provides token presets for a convenient usage.

```
import intel.dtl as dtl

# Select the token type, intel or oem
token_type = dtl.TokenType.OEM

# Select the platform, Alder Lake, Meteor Lake...
token_project = dtl.Project.ALDERLAKE

# Select a knob preset
token_preset = dtl.Preset.SYSTEM_DEBUG
# token_preset = dtl.Preset.SYSTEM_TRACE

token = dtl.Token.from_preset(token_preset, token_project, token_type)

# Apply part ID to the token (see snippet above)
token.manifest_extension.part_ids.add(part_id)
```

Sign Token

The Intel® CSME firmware requires a token to be signed.

DTL supports different ways of signing:

- Use a private key file

```
import intel.dtl as dtl
import pathlib

sign_method = dtl.signing.LocalKey(pathlib.Path("<local path>"),
dtl.signing.CredentialScheme.CMD_LINE)
```

- Intel internal signing options.

Inject token

While the target is connected and available Hotham can connect to the Intel® CSME firmware and inject the token.

```
from intel.hotham import hotham

hotham.inject_token("C:\\\\token.bin", False, False)

print("Manually reset target to apply changes.")
```

Example Scripts:

- Enable debug and trace using OEM Token on Alder Lake

```
from intel.hotham import hotham
import intel.dtl as dtl
import pathlib

# Load Part ID using Hotham
code, data = hotham.loader.dscp.get_part_id()
if code != 0:
    raise RuntimeError("Failed to read Part ID from target.")
pid_pseudonym = int(''.join(data['PART_ID'][::-1]), 16)
pid_nonce = data['NONCE']
pid_timestamp = data['TIMESTAMP']

# Create token based on debug token Library presets
token_type = dtl.TokenType.OEM
token_project = dtl.Project.ALDERLAKE
token_preset = dtl.Preset.SYSTEM_DEBUG
token = dtl.Token.from_preset(token_preset, token_project, token_type)

# Apply part ID to the token
token.manifest_extension.part_ids.add(dtl.PartID(pid_pseudonym, pid_nonce, pid_timestamp))

# Use a private key file to sign token
sign_method = dtl.signing.LocalKey(pathlib.Path("C:\\key_private.pem"),
dtl.signing.CredentialScheme.CMD_LINE)

# Write token to file
token.to_file(path=pathlib.Path("C:\\token.bin"), sign_method=sign_method)

# Inject token via hotham
hotham.inject_token("C:\\token.bin", False, False)

print("Manually reset target to apply changes.")
```

Offline Debug Token creation using Flash Programming Tool (FPT)

! Note

FPT is an Extendable Firmware Interface (EFI) application which runs directly on the target

These instructions cover:

1. Prepare USB Pendrive to use FPT
2. Obtain Part ID using FPT
3. Create Debug Token for Offline connection
4. Inject Token using FPT

Prepare USB Pendrive to use FPT

On Host Computer:

1. Format USB Pendrive with FAT32
2. Fpt.efi to USB Pendrive
 - Browse your Intel® CSME firmware kit to obtain Fpt.efi
 - Location: `OWR/CSME/External_..../Tools/System_Tools/FPT`

Obtain Part ID using FPT

A **Part ID** uniquely identifies a platform. This allows to associate the created debug token to an individual platform. It is highly recommended to limit the usage of debug tokens to individual platforms.

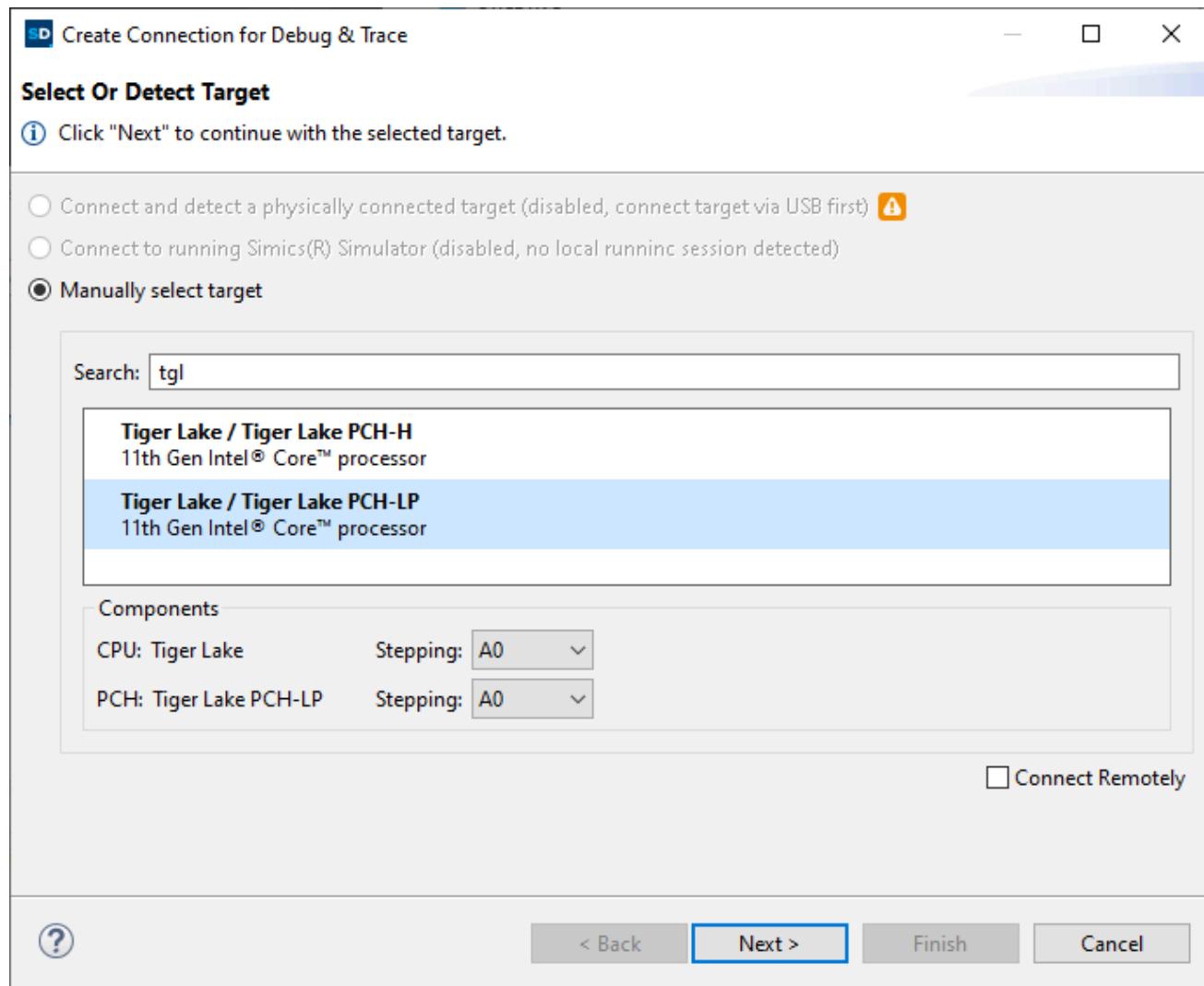
On Target Computer:

1. Plug in the USB drive with Fpt.efi binary
2. Power on, enter BIOS setup (F2), and in the boot menu enable boot to EFI shell, save settings
3. Boot to EFI shell
4. Go to the USB device, for example type `fs0:`
5. Run Fpt.efi to get the device PID: `Fpt.efi -GETPID PID.bin` this will create PID.bin file with the part ID.

Create Debug Token for Offline connection

1. Start Intel® System Debugger using a new workspace.
2. Create a target selection / connection manually selecting target

- Search the platform for which a debug token should be created.
- If the stepping is known, select the correct stepping. Try default stepping otherwise (should not have impact on token creation).
- Click Next.



3. Select any connection type

- All connection types support token creation
- Debug token injection using is only supported for DCI probes (e.g. DbC)
- Click Next.

Select Connection Configuration

Please choose a connection configuration, then click on the "Next" button to continue to the connection summary page.



Connect to a SIMICS® Simulator session

Offline Target Dump

Offline (display existing data)



Load an existing target dump and analyze it offline.

Offline (display existing data)

No active simulation, no active target connection.

Hide disconnected probes



< Back

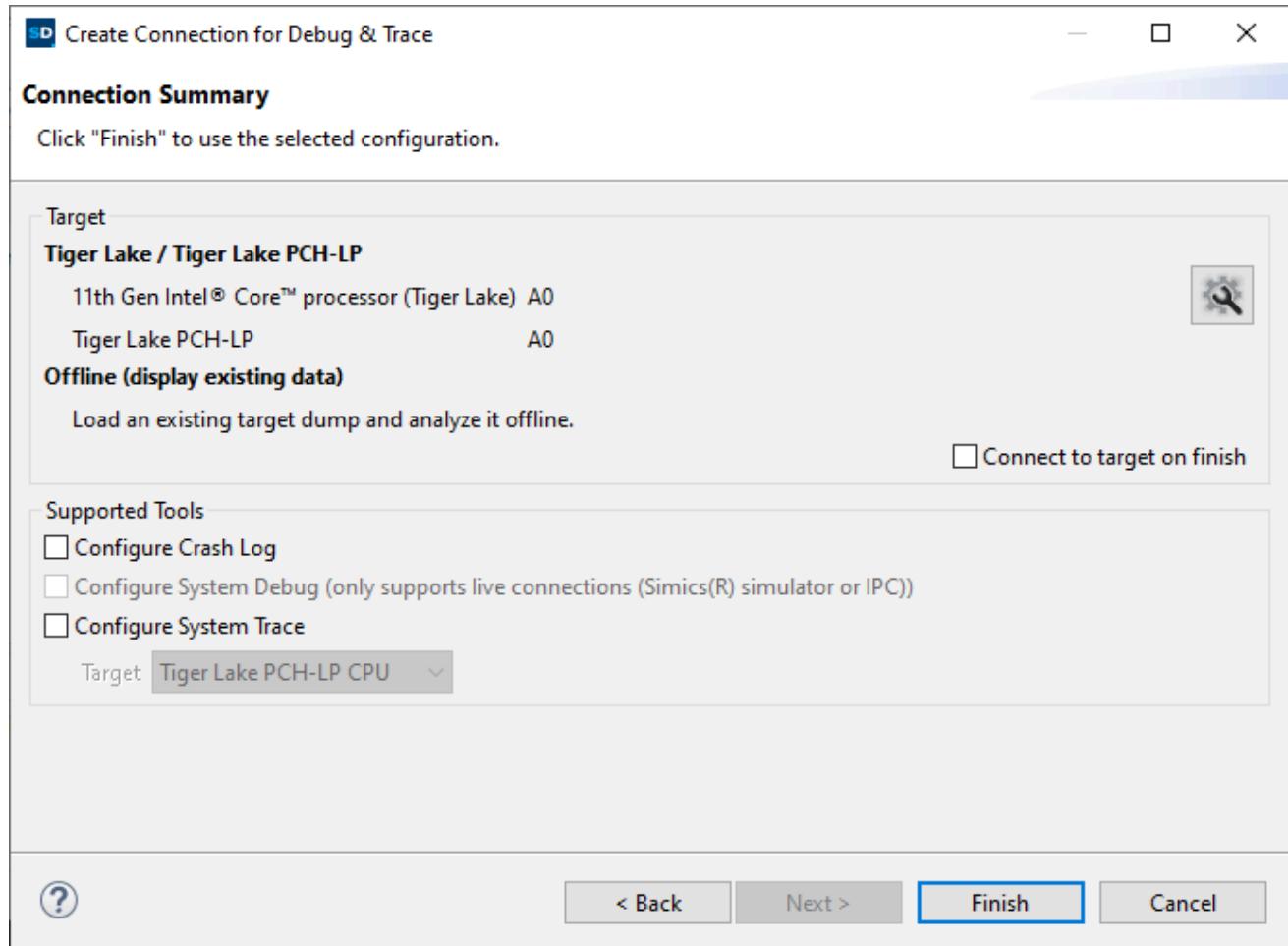
Next >

Finish

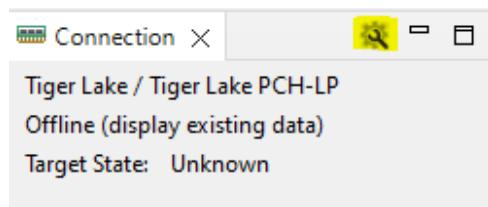
Cancel

4. Finalize configuration

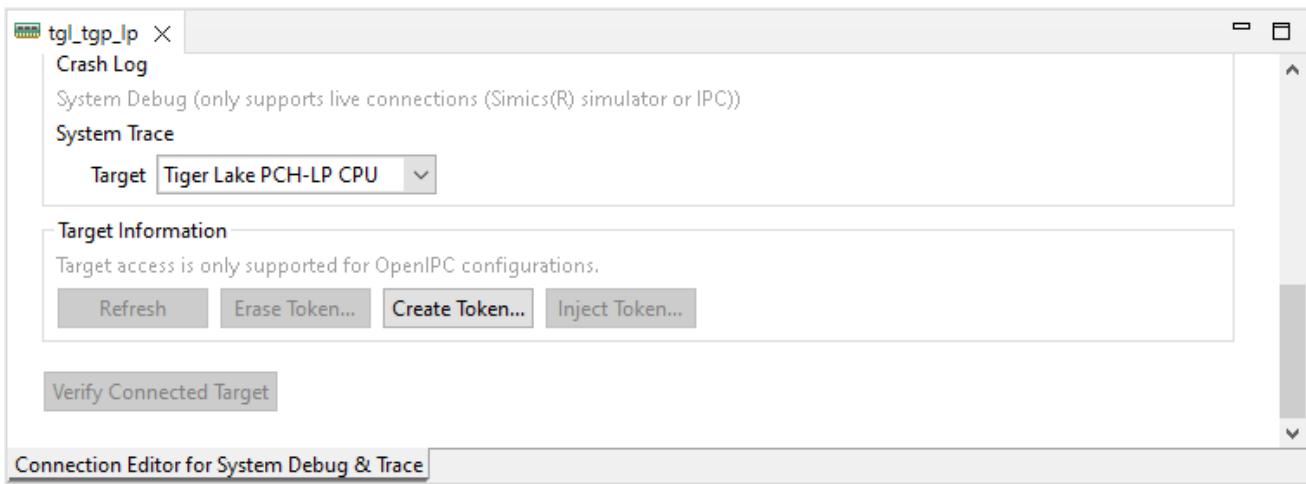
- For only token creation Crash Log / System Debug / System Trace support can be disabled (optionally)
- Click *Finish*



5. Click the gear icon to open connection editor

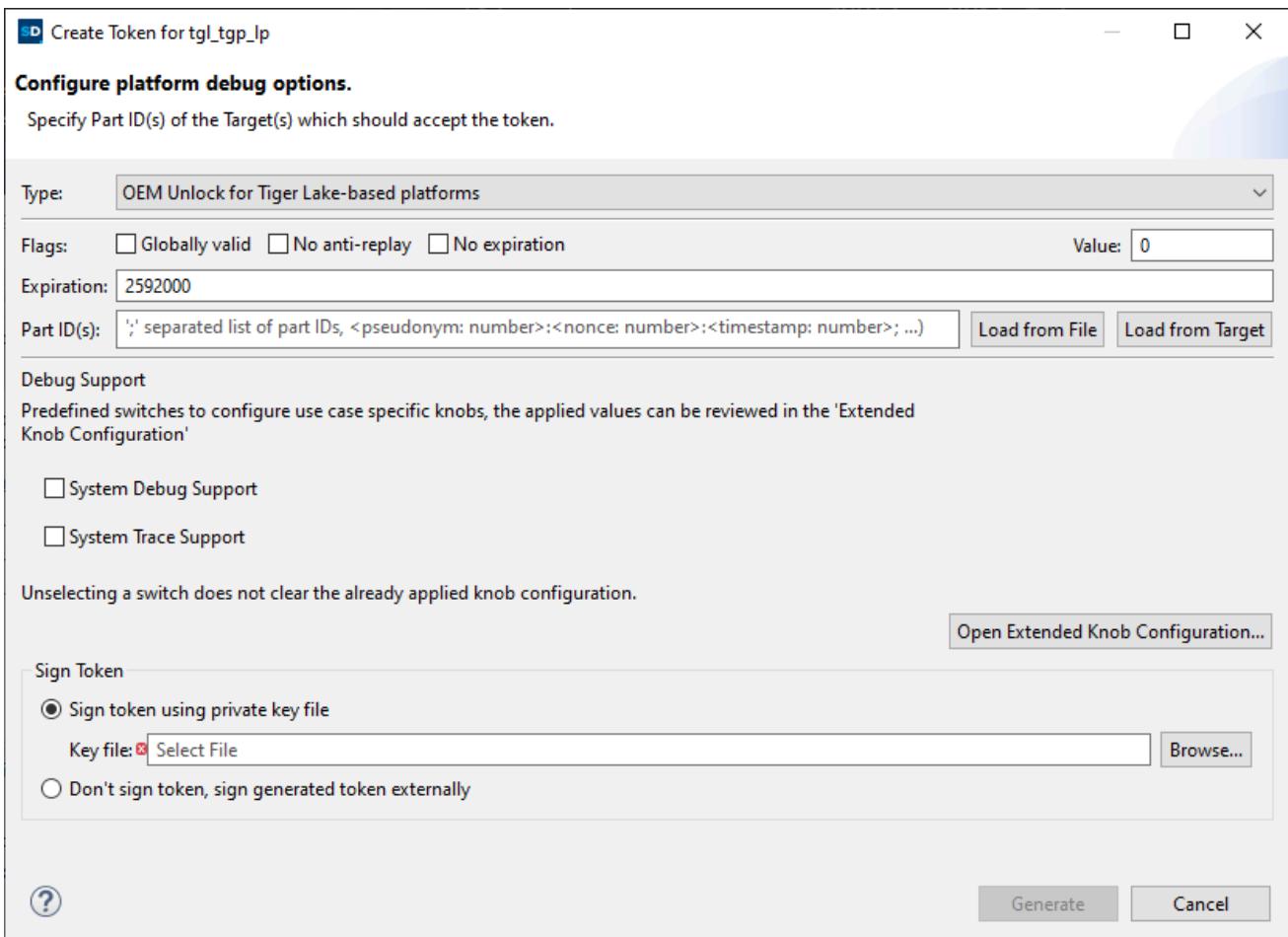


5. Scroll to *Target Information* and click *Create Token*



6. Create Token

- Use *Load from File* to import an existing Part ID binary (e.g. created using FPT)
- Follow generic [Create Token](#) instructions.



Inject Token using FPT

On Host Computer:

1. Copy created debug token binary (.bin) to the USB Pendrive (which already contains FPT)
2. Unplug the USB drive (use [Safely remove hardware in Windows*](#) to prevent possible data loss)

On Target Computer:

1. Plug in the USB drive with Fpt.efi binary and debug token binary (.bin)
2. Power on, enter BIOS boot menu (F10) and boot to EFI shell
3. Go to the USB device, for example type “fs0:”
4. Run: `Fpt.efi -writetoken tmt_tmp_token_full.bin` to write the token
5. Run: `Fpt.efi -greset` to reset target

Debug Token creation and Firmware upload using DnX

Note

This is a preview and changes might happen to this document or the API in upcoming releases.

Intel® Download and Execute (Intel® DnX) can be used to download Firmware or perform debug token flows via a USB 2.0 when the Intel® Direct Connect Interface is not available (Intel® DCI).

This documentation only describes the usage of Intel® DnX in the Intel® System Debugger environment.

Please check the detailed Intel® DnX User Guide and the Intel® Signing and Manifesting User Guide from the Intel® CSME firmware kit for more information about Intel® DnX.

Use DnX to read Part ID and inject debug token in UI

1. Start and setup Intel® System Debugger to use the correct target

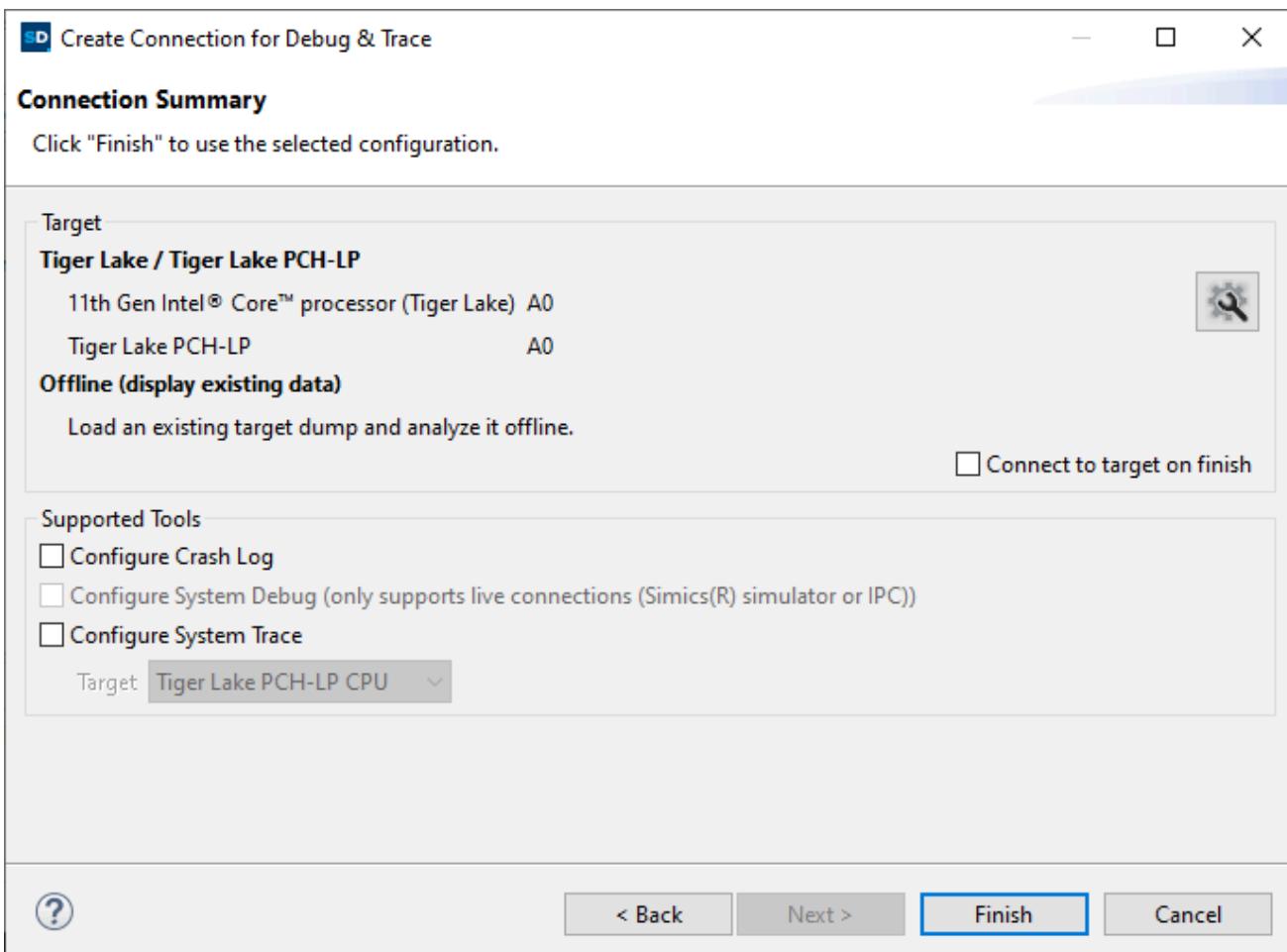
If the target is connected via a debug probe and can be enumerated follow the auto detection steps.

If the target is already in Intel® DnX mode or not connected via a debug probe follow the *Manually select target* steps.

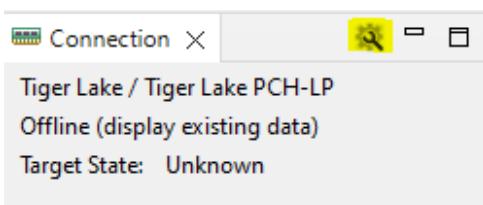
Note

When performing connection selection un-check *Hide disconnected probes* to select the expected debug probe. If no debug connection is required afterwards continue with *Offline*.

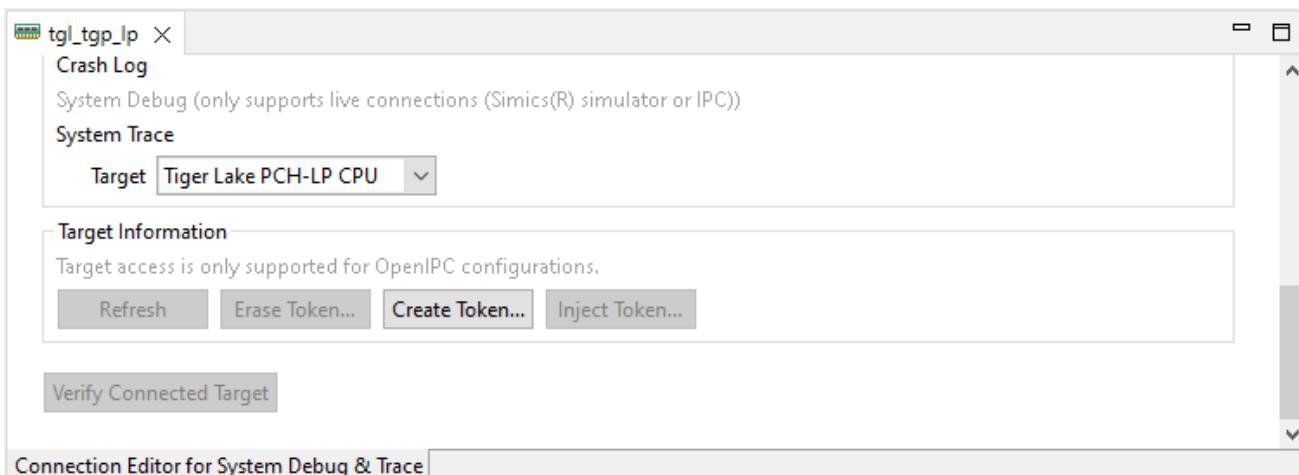
For debug token creation only, Crash Log / System Debug / System Trace support can be disabled (optionally) Click *Finish*



1. Click the gear icon to open connection editor



1. Scroll to *Target Information* and click *Create Token*



1. Boot platform into Intel® DnX mode

What triggers are supported to enter DnX mode depend on the specific platform and vendor choices.

2. Follow generic [Create Token](#) instructions to configure the debug token. Don't fetch another Part ID using the UI!
3. Open a cmd window side by side to the Debug Token Creation Window to fetch Part ID from Intel® DnX
 - a. Open CMD window and navigate to the Intel® System Debugger installation
 - b. Run `isd_env.bat` to source the Intel® System Debugger environment
 - c. Run `ipython`

```
import intel.dnx as dnx

# Enumerate DnX and Configure Device
# -----

# Get available DnX devices
devices = dnx.get_devices()
# We expect to have only one device connected
device = devices[0]

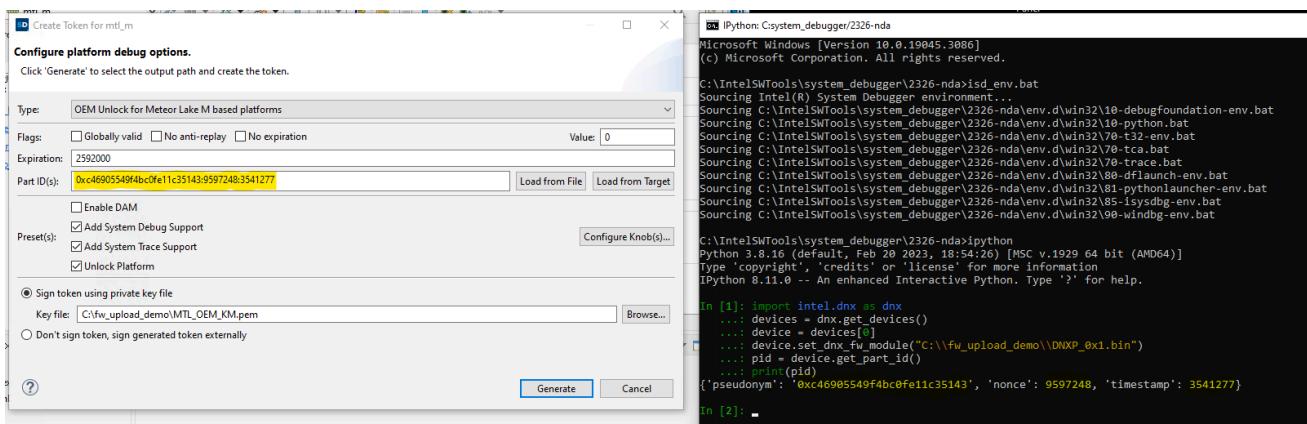
# Configure platform / firmware specific DnX Module
device.set_dnx_fw_module("C:\\\\fw_upload_demo\\\\DNXP_0x1.bin")

# Get Part ID
pid = device.get_part_id()

print(pid)
```

- d. Keep the cmd window open to continue with the debug token injection afterwards

4. Copy the Part ID to the Debug Token Creation Window



5. Click Generate to create the Debug Token

Note

Using a short path without spaces is recommended to simplify selection in Python for injection afterwards.

6. Inject the Debug Token using Intel® DnX

- Go back to the already running ipython session and run:

```
device.inject_debug_token("C:/fw_upload_demo/token_mtl_m_1687447606601.bin")
```

7. Power Cycle Platform without DnX Trigger to apply / use Debug Token

Use DnX to read Part ID and inject debug token in CLI

1. Boot platform into Intel® DnX mode

What triggers are supported to enter DnX mode depend on the specific platform and vendor choices.

2. Create Debug Token, read part ID and inject the debug token via DnX

- Open CMD window and navigate to the Intel® System Debugger installation
- Run isd_env.bat to source the Intel® System Debugger environment
- Run ipython

```
import intel.dnx as dnx
import intel.dtl as dtl
import pathlib
import os

# Enumerate DnX and Configure Device
# -----

# Get available DnX devices
devices = dnx.get_devices()
# We expect to have one device connected
device = devices[0]

# Configure platform / firmware specific DnX module
device.set_dnx_fw_module("C:\\\\fw_upload_demo\\\\DNXP_0x1.bin")

# Get Part ID
pid = device.get_part_id()

# Generate OEM Debug Token Using Debug Token Library
token = dtl.Token.from_preset(dtl.TokenConfigs().set_config(config="mtlm_oem_unlock_token",
    preset="system_trace"))
token.manifest_extension.part_ids.add(dtl.manifest.PartID(pseudonym=int(pid.pseudonym, 16),
    nonce=pid.nonce, time_base=pid.timestamp))
sign_method = dtl.signing.LocalKey(pathlib.Path("C:\\\\fw_upload_demo\\\\MTL_OEM_KM.pem"),
    dtl.signing.CredentialScheme.CMD_LINE)
token.to_file(path="C:\\\\fw_upload_demo\\\\mtlm_oem_token.bin", sign_method=sign_method)

# Inject DnX Capabilities Token
device.inject_debug_token("C:\\\\fw_upload_demo\\\\mtlm_oem_token.bin")
```

3. Power cycle platform without DnX trigger to start the normal boot flow and apply / use Debug Token

Use DnX to download firmware in CLI

1. Prepare OEMKeyManifest and DnX Firmware Image

This is only a quick summary of the steps to actually do the preparation. There will be a detailed DnX User Guide compatible with Intel® System Debugger after the PoC phase.

Check the *Signing and Manifesting User Guide* for more information about how to generate an own private key and include it into the firmware.

These documents and the mentioned tools are available in the Intel® CSME firmware kit.

Setup OEMKeyManifest

- a. Use Intel® MEU tool to create OEMKeyManifest template
- b. Add “OemDnxIfwiManifest” and “OemDebugManifest” into OEMKeyManifest.xml
- c. Use Intel® MEU tool to generate a signed OEMKeyManifest binary
- d. Update Intel® IFWI using Intel® MFIT to include OEMKeyManifest

Create DnX Firmware Image

- a. Open Intel® MFIT tool
- b. Select Intel® <platform> - DnX Recovery Image Layout
- c. Configure all fields and generate a DnX Firmware Recovery Image

2. Boot platform into Intel® DnX mode

What triggers are supported to enter DnX mode depend on the specific platform and vendor choices.

3. Authorize and download DnX Firmware Recovery Image

- a. Open CMD window and navigate to the Intel® System Debugger installation
- b. Run isd_env.bat to source the Intel® System Debugger environment
- c. Run ipython

```

import intel.dnx as dnx
import intel.dtl as dtl
import pathlib
import os

# Enumerate DnX and Configure Device
# -----

# Get available DnX devices
devices = dnx.get_devices()
# We expect to have one device connected
device = devices[0]

# Configure platform / firmware specific DnX Module
device.set_dnx_fw_module("C:\\\\fw_upload_demo\\\\DNXP_0x1.bin")

# Prepare Firmware Upload
# -----

# Download OEM Key Manifest, required to authenticate following binaries
device.download_oem_key_manifest("C:\\\\fw_upload_demo\\\\OEMKeyManifest_signed.bin")

# Get Part ID
pid = device.get_part_id()

# Create DnX Capabilities Token (flags are configured to be executed only once)
token = dtl.Token.from_preset(dtl.TokenConfigs().set_config(config="mtlm_oem_unlock_token"))
token.manifest_extension.flags = 9
token.manifest_extension.part_ids.add(dtl.manifest.PartID(pseudonym=int(pid.pseudonym, 16),
nonce=pid.nonce, time_base=pid.timestamp))
token.set_knob("dnx_capability", "31", True)
sign_method = dtl.signing.LocalKey(pathlib.Path("C:\\\\fw_upload_demo\\\\MTL_OEM_KM.pem"),
dtl.signing.CredentialScheme.CMD_LINE)
token.to_file(path="C:\\\\fw_upload_demo\\\\dnx_capabilities_token.bin", sign_method=sign_method)

# Inject DnX Capabilities Token
device.inject_capabilities_token("C:\\\\fw_upload_demo\\\\dnx_capabilities_token.bin")

# Download DnX Firmware Recovery Image
# -----

# This takes time (~5 min for a 32 MB image on MTL-P)
device.write_firmware("C:\\\\fw_upload_demo\\\\dnx_recovery_image.bin")

# Manually reboot platform

```

Learn More about Debug Probes and Technologies

This topic explains the major differences between modern debug technologies and probes supported by Intel® System Debugger.

Debug Technologies

Technologies used for system debugging can be divided into two main classes: closed and open chassis.

Open Chassis

Provides access to the JTAG (Joint Test Action Group) communication protocol via the Intel® In-Target Probe (Intel® ITP) - XDP3. Main disadvantages include relatively high price and low bandwidth compared to the closed chassis probes.

Closed Chassis

Provides access to the JTAG via USB-based probes.

- USB 2 (also known as USB High Speed)

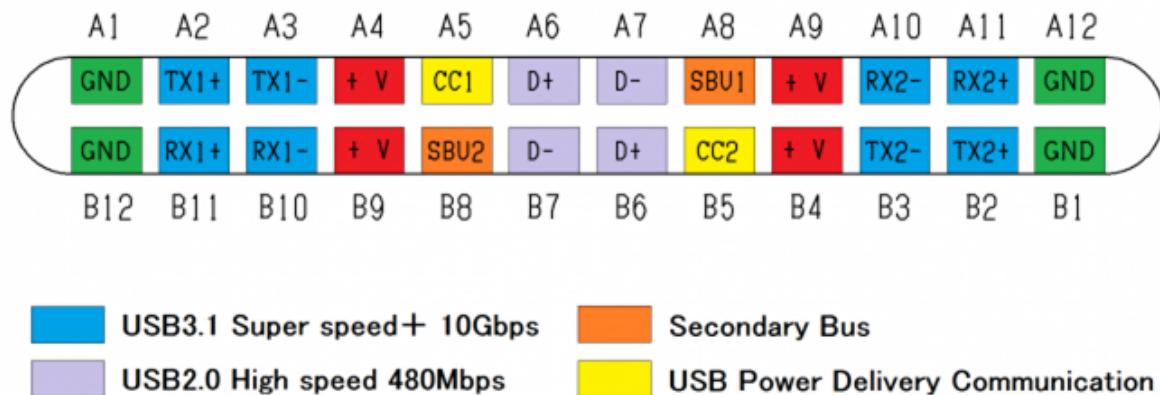
Advantage: early “enablement” in boot flow

- USB 3

Advantage: high performance tracing and memory access

- USB Type-C*

Can act as a power source (Downstream Facing Port or DFP) or as a power sink (Upstream Facing Port or UFP) depending on the termination of CC1 and CC2 control signals (yellow in the diagram below):



In contrast, USB Type-A* ports usually act as a power source (DFP) and USB Type-B* ports as a power sink (UFP).

Intel® Direct Connect Interface (Intel® DCI)

Way of accessing JTAG Target Access Port (TAP) network via USB port (closed chassis).

Communication with the bridge can be done with two protocols:

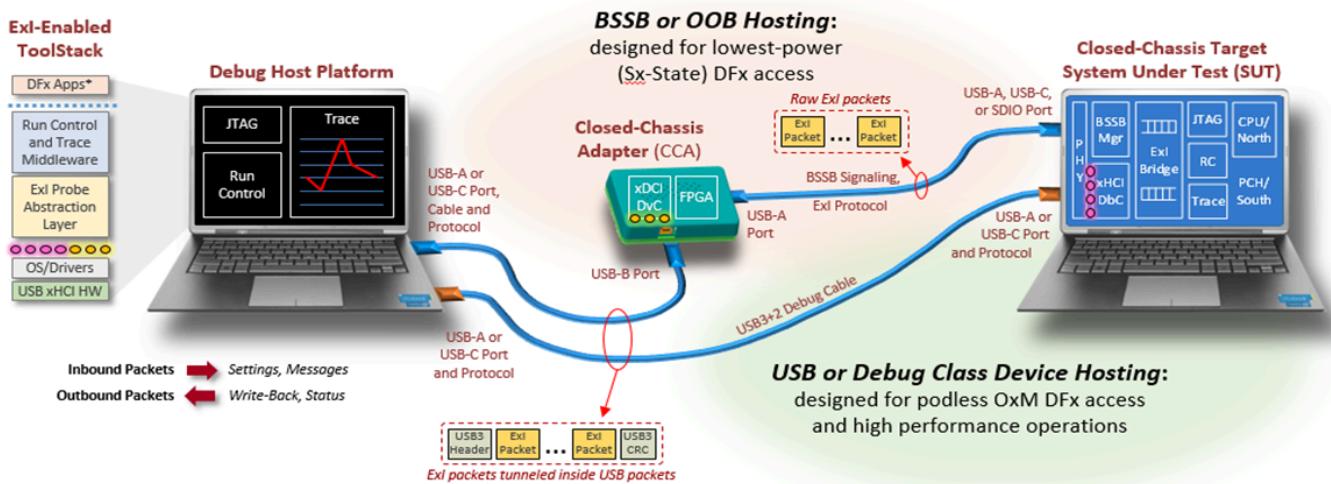
- BSSB (Boundary Scan SideBand interface) also known as Intel® Direct Connect Interface (Intel® DCI) Out of Band (OOB), used by Closed Chassis Adapter (CCA)
 - Over USB Type-A* (BSSB 4-wire) or over USB Type-C* (BSSB 2-wire)
- USBx.Dbc (Debug Class) used by the USB debug cable

Provides different endpoints:

- GP1 - Kernel mode debug
- GP2 - Direct Memory Access (DMA)
- Trace - NorthPeak trace streaming
- DFx - interaction with the interface to the JTAG/TAP network

DbC mode is exclusive

See the following schema of the Intel DCI Setup (high level)



Debug Probes

You can connect your host system (where the debugger is running) to the target system (to be debugged) using different probes.

The following are the common types of probes supported for system debugging:

Probe Name	Chassis Type	Debug Me
Intel® ITP-XDP3	Open	JTAG / ITF
Intel® Silicon View Technology (Intel® SVT) Closed Chassis Adapter (CCA)	Closed	Intel DCI /
Intel® SVT Debug Class (DbC) USB 2/3 A-to-A	Closed	Intel DCI /
Intel® SVT DbC USB 2/3 A-to-C (DFP and UFP)	Closed	Intel DCI /

For more information, contact [Customer Support](#).

Host Operating Systems

Intel® System Debugger 2023 NDA supports the following host operating systems:

- Microsoft Windows* 11
- Microsoft Windows* 10 21H1/21H2
- Microsoft Windows* Server 2019
- Microsoft Windows* Server 2022
- Ubuntu* 20.04 LTS
- Ubuntu* 22.04 LTS
- Rocky Linux* 8, 9
- Fedora Linux* 37, 38, 39
- RHEL 9
- SUSE Linux Enterprise 15 SP
- Debian* 11

Due to compatibility issues, Intel® System Debugger 2023 NDA does not support older Linux versions than those mentioned above.

For questions and assistance, contact [Customer Support](#).

Get Help

To submit an issue, go to the [Online Service Center](#) and select Intel® System Bring-up Toolkit NDA.

Alternatively, you can report issues at [Intel® Premier Support](#). For additional support resources, go to <https://software.intel.com/content/www/us/en/develop/tools/support.html>.

Trainings

Recommended training videos for Intel® System Debugger usage.

- [Debug and Tracing using Intel® System Debugger](#)

Language: English

Audience: BIOS Engineers, Platform Debug Engineers

This demo covers a typical debug use-case of post-code debugging. Using Intel® System Debugger, we plant an IO breakpoint at port 80 to break in at the exact point in BIOS source code when the post code is displayed on the board. We use the converged debug and trace interface to capture AET traces simultaneously which will record the IO transactions on port 80. We then use the ISD tool to step through C source code and inspect various registers and MSRs.

- [Establishing a debug connection to Simics® simulator](#)

Language: English

Audience: BIOS, Firmware developer

This demo shows how to debug a local Simics® simulator session and how to connect to a remote Simics® simulator session over the network.

- [Crash Log: Basics](#)

Language: English

Audience: BIOS Engineers, Platform Debug Engineers

Crash Log provides platform-level diagnostic information. It captures and preserves the hardware state immediately after a crash and persists across a set of platform resets. This demo shows how to manually trigger, extract, and analyze the data.

- [System Debug: Basics](#)

Language: English

Audience: Intel Internal, NDA customers

This video gives a brief overview of various features offered by the Intel® System Debugger

- System Debug component. It shows how to create a debug configuration, do run control, and set up breakpoints. It also explores registers, memory, and execution trace views offered by the debugger.

- [System Debug: Demo of Processor Trace and PCI Viewer](#)

Language: English

Audience: Intel Internal, NDA customers

The Intel® System Debugger (ISD) has many advanced debugging features. The demo illustrates how to use Intel® Processor Trace in ISD, which shows the CPU's execution traces to understand the execution flow. The second part of demo shows how to dump PCI device configurations using ISD, which shows all PCI devices. The user can modify each device's config space using the tool.

- [System Trace: Basics](#)

Language: English

Audience: NDA customers

This short video highlights how Intel® System Debugger's System Trace can be used to trace data. System Trace provides an ongoing record of hardware and software events that occur during system initialization and operation.

- [System Trace: Early Boot Trace Capture](#)

Language: English

Audience: CSME Engineers, Platform/Power-on Debug Engineers

This demo will show how Intel® System Debugger command line script can be used to capture early boot traces, including CSME ROM/RBE messages over G3/Reset cycles.

- [System Trace: Enable Early Boot Trace Capture using a debug token](#)

Language: English, Mandarin

Audience: Validation, Debug Engineers

For customer boards with only USB connections, use of USB2.Dbc to collect ITH traces is critical for debug. With OEM Debug token enablement, the customer is required to get early boot ITH messages for token verification. This training provides information on how to collect ITH traces using USB2.Dbc and also to collect early trace messages using the same cable.

- [System Trace: Capture and Decode Release BIOS Traces](#)

Language: English

Audience: BIOS Engineers, Platform/Power-on Debug Engineers

This demo will show how to capture and decode Release BIOS traces using ISD tool. It also shows how to correlate decoded Release BIOS traces with CSME traces and understand the timing difference between different trace messages. This method of tracing release BIOS helps in debugging issues on production platform without having to re-flash the debug BIOS image. It is also useful in triaging timing issues which are typically seen on release BIOS versions but not reproducible on debug BIOS versions.

- [Windbg-ext: Basics](#)

Language: English

Audience: Platform Debug Engineers

Intel® Debug Extensions for WinDbg supports the issues that basic WinDbg could not. This demo shows how to efficiently launch live debug when you enable hyper-v as the target.

The following trainings cover key user scenarios and concepts in video and/or text formats. To access the materials, you will need to login to the [Resource & Documentation Center](#) (contact [Customer Support](#) for assistance).

- [Solving early power on issues using System Trace](#) (content ID: 634203)
- [Using In-Circuit Emulator Breakpoint \(ICEBP\) for EFI source line debugging in Intel® System Debugger](#) (content ID: 646277)
- [Intel® System Debugger Over Simics® Simulator](#) (content ID: 634171)
- [How to Capture Trace Log on Simics® Simulator using Intel® System Debugger](#) (content ID: 682413)
- [How to Capture Intel® Trace Hub \(Intel® TH\) Early Boot Trace over USB2 Debug Class \(DbC\) Cable](#) (content ID: 642042)

System Debug

Intel(R) System Debugger - System Debug is a modern, GUI-based architecture system software debugger. It presents a view of the system state (architecturally defined processor registers and platform registers) along with the software being executed on the target system.

As a software debugger, most of the System Debug features are consistent with other GUI-based debuggers such as Eclipse* or Microsoft Visual Studio*. Menus, context menus and toolbar buttons provide easy access to the debugger's functionality. You can run and stop the target, view instructions or source code, view and modify memory, registers, program variables, and much more.

As an architectural debugger, the System Debug provides a powerful set of features for exploring the state of the processor. For example, you can directly access and modify CPU registers and platform registers or easily debug in the System Management Mode (SMM).

The debugger GUI provides complete control of the debugging process. You can access most of the basic functions, such as single-step, step-through-function, run and display memory, by clicking toolbar buttons. The GUI supports multiple source windows, evaluating expressions and changing their values, and dragging and dropping expressions into the Evaluations window.

Before You Begin

Before you start using the Intel(R) System Debugger, do the following:

1. Ensure that the binaries you want to debug are compiled with debug information and without optimization. Otherwise, compiler optimization might affect debug experience.

The debugger will check optimization for PE/COFF binaries and will warn you if the debug experience is affected.

See [Recommended Compiler Flags](#).

2. Check [supported platforms and probes](#) for the present release.
3. Complete the [startup procedure](#): configure and connect the target.

⚠ Warning

Before establishing a target connection, ensure that the [Breakpoints view](#) does not contain any breakpoint instances. Otherwise, it might cause the target halt when the connection is created.

Recommended Compiler Flags

For Microsoft Visual C++ (MSVC), make sure you use the following compiler flag combination:

```
/Od /Zi
```

If a small binary is a must, use the following combination:

```
/Od /Os /Zi
```

The `/Os` flag might affect the debug experience as well. Use this flag only if the binary cannot be executed without it.

Avoid the following flags as they impact debug experience:

- `/O1` and `/O2`
- `/Ob` with non-zero values
- `/Ot`
- `/Oy` on 32-bit
- `/Ox`

Set up System Debug

To launch a debug session with System Debug, complete the following steps:

1. Complete [prework steps](#).

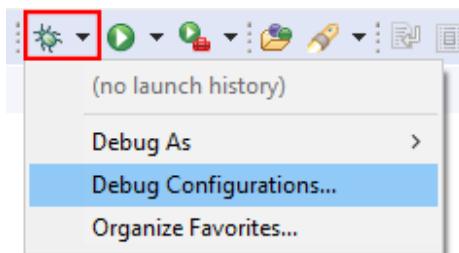
2. Once the target is connected, click the  Debug button in the main Eclipse* toolbar to open primary System Debug views.

3. (Optional) [Customize the debug configuration](#).

Change System Debug Configuration

Create a new debug configuration.

1. In the main Eclipse* toolbar, expand the drop-down menu next to the  Debug button and click **Debug Configurations**.



The latest System Debug configuration is opened.

2. (Optional) Configure parameters in **Debugger** tab:



The path to the debug engine is specified in the **Debugger Path** field.

- To rename the debug configuration, type in a new name into the **Name** field and click **Apply**.
- You can also filter the hardware threads to be displayed.

For instructions, see [Filter Hardware Threads](#).

- You can allow the debugger to generate a call stack trace (unwind the stack) when no debug symbols are loaded. To enable it, check the **Call stack unwinding without debug symbols** box.

If you leave the box unchecked (default), only top-most stack frame will be shown if no debug symbols are loaded.

For advanced configuration options, see [Configure Call Stack](#).

- To load symbol files for each hardware thread separately (by default, symbols for all threads are loaded at once), check the **Per-thread memory map and debug symbols** box.

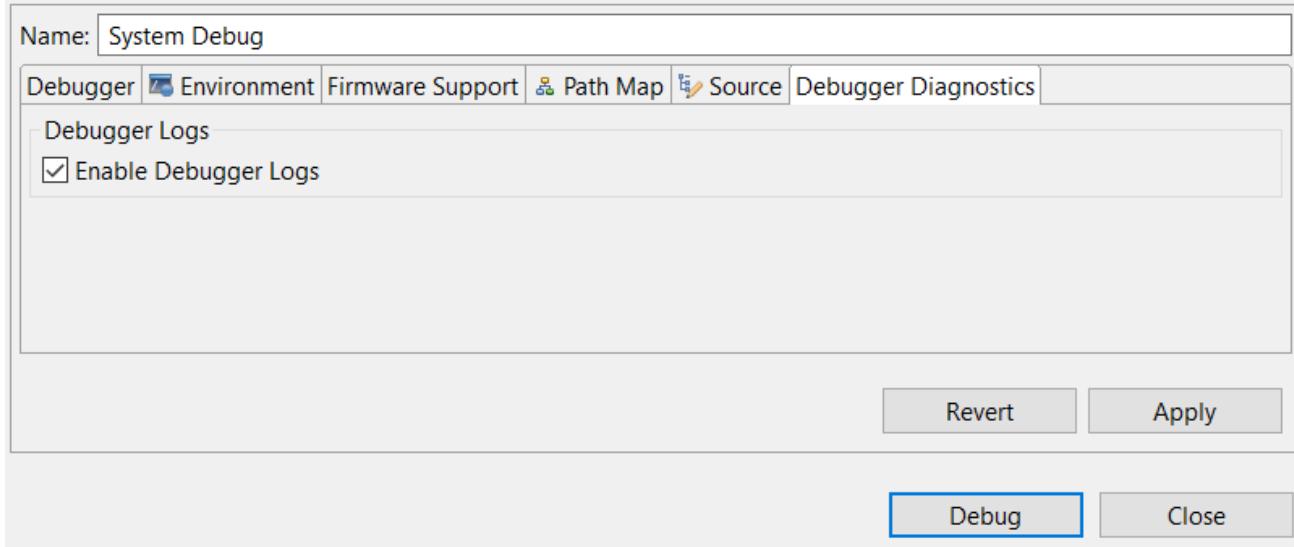
With this box checked, each debug context will get its own memory map.

- To re-enable hardware breakpoints over reset or S3 enter/exit flows, please select this check box. By default hardware breakpoints get disabled over reset.

Generic Configuration

Call stack unwinding without debug symbols
 Per-thread memory map and debug symbols
 Re-enable hardware breakpoints over reset

3. Optionally, you can enable logs collection. To do this, go to the **Debugger Diagnostics** tab and check the **Enable Debugger Logs** box. The location of the log file appears under the **Create, manage, and run configurations**.



4. Click **Debug**.

Now you can follow basic debugging steps described in the [corresponding section](#).

Disconnecting from the Debugger

To terminate the debugging session and disconnect the target, follow the steps below:

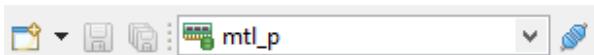
1. Stop the debugger by right-clicking the active debug configuraton and selecting the **Disconnect Debugger** option in the dropdown menu:



Important

The debug session is terminated now but your JTAG or Intel(R) Direct Connect Interface (Intel(R) DCI) connection is still alive. To disconnect the target, do the next step.

2. Ensure that the debugger is terminated and then disconnect the target by clicking the  button next to the Connection field in the Eclipse* toolbar:



The Target Connection Assistant terminates the connection.

JTAG Debugging Specifics

JTAG debugging requirements and limitations.

! See also

Platform Closed Chassis Debug User Guide. Sign in to [Resource & Documentation Center](#) and search by document number [626623](#).

JTAG Debugging Requirements

Software debugging over the Joint Test Action Group (JTAG) has some fundamental requirements:

- A reliable target connection
- A JTAG-enabled CPU
- Knowledge of the implementation of the platform being debugged

Reliable Target Connection

With a reliable target connection, the signaling requirements of JTAG are met, and communication at the JTAG protocol level is possible. Additionally, there may be sideband signals that are not part of the generic JTAG spec but are required for debugging Intel(R) processors. Specific requirements for each platform are described in the “Debug Port Design Guide” available from Intel.

JTAG-enabled CPU

Some Intel processors are not factory-enabled for JTAG debugging, while others may limit JTAG-accessible functions depending on fusing, firmware configuration, board strappings, or other factors. Note that “enabled” refers to the processor core, which is required for software debugging. Even if the processor is disabled, it is still possible for the fundamental JTAG connection to work, and access to other SoC resources to be possible.

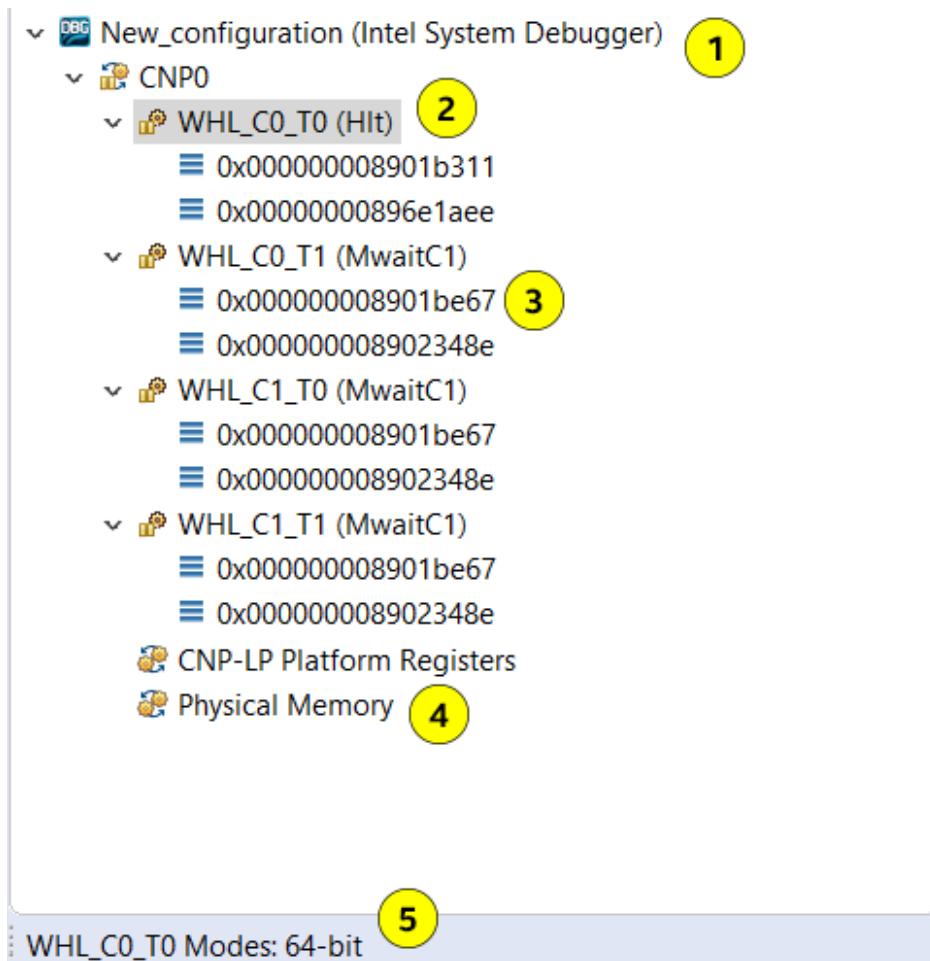
Platform-specific Knowledge

The last 10% of the solution requires platform-specific knowledge to keep the debugger out of trouble. Examples include knowledge of watchdog timers, memory or flash layout, PCI topology, or other platform implementation-specific details that can inhibit or disrupt the debug session.

Debugging Basics

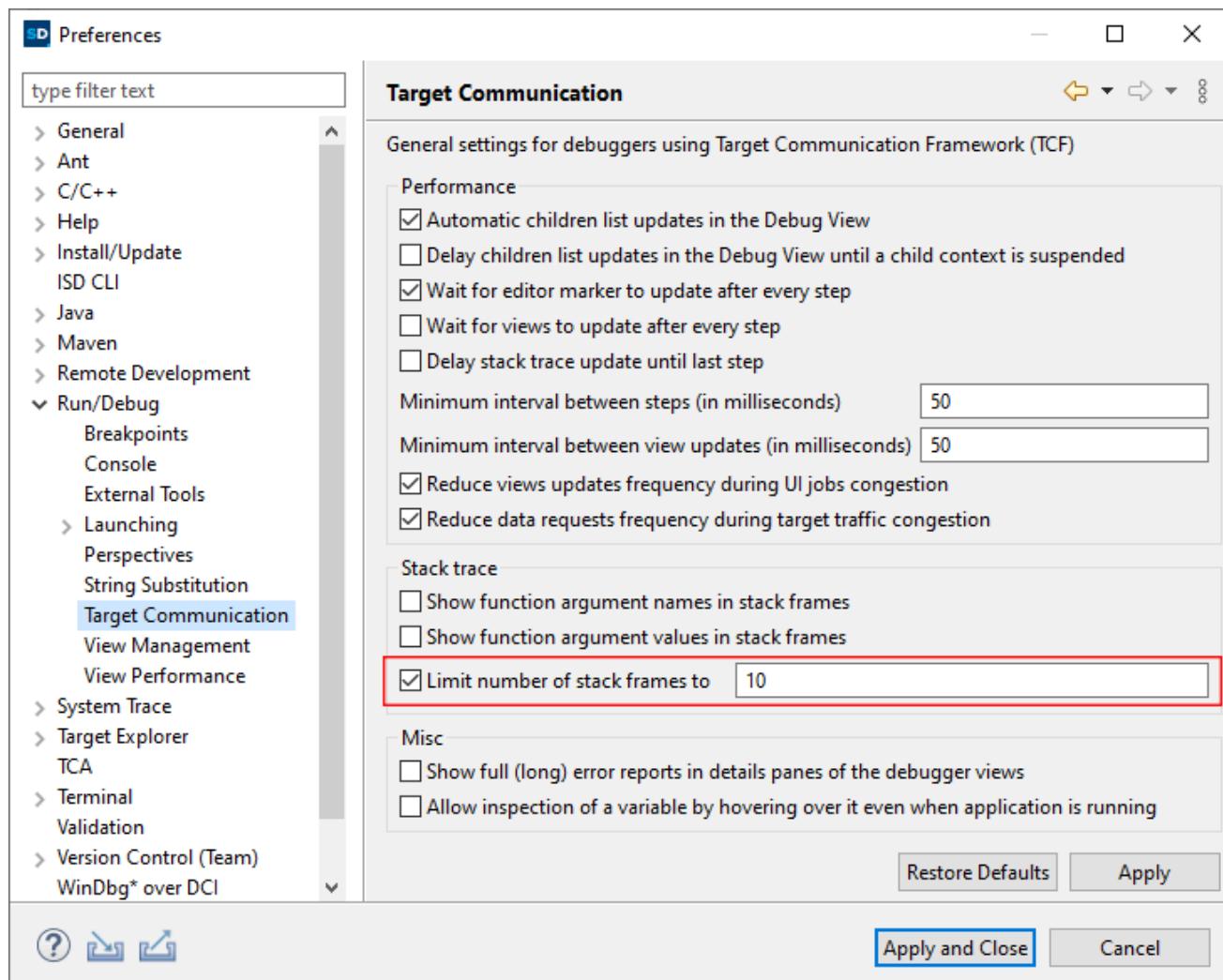
This section describes basic concepts and operations that you can apply in different debugging tasks.

After you connect to the target, launch the debugger and halt the target, you see the following in the [Debug view](#) by default:



1. Debug configuration that you [created](#) after connection to the target.
2. Hardware threads. You can filter the threads by name as described [here](#).
3. Stack frames.

By default, maximum three stack frame instances are displayed. To change the limit, open **Window > Preferences** menu, go to **Run/Debug > Target Communication** pane, and change the specified number:



Click **Apply and Close** to save the changes and close the dialog box.

- Physical memory node. If you select it, you will be able to work with physical memory in the [Memory Browser view](#).

For more information, see [Working with Memory](#).

- CPU operating mode of a particular thread. For more information, see [CPU Operation Modes](#).

To enable debugger functionality, select the debug context you want to work with in the [Debug view](#). If no debug context is selected, most of the buttons in the Eclipse* toolbar are greyed out.

Target Run Control

The first step in debugging with the Intel(R) System Debugger is controlling the execution of the target. Similar to any other software debugger, you can run, stop, and reset the target. When the target is stopped, you can perform various debugging operations at both assembly and source levels.

Details about the target, the connection method, and the connection status are displayed in the [ISD Shell](#).

To control the target, select the desired hardware thread in the [Debug view](#) and use the following buttons in the toolbar or the **Run** menu:

Icon	Name	Action
	Suspend	Halts execution of the selected thread in
	Resume	Resumes execution of the currently susp
	Terminate	Ends the selected debug session or proc
	Reset	Does a warm reset of the target.
	Power-cycle	Does a target power-cycle using the Pow
	Load the Current Module (formerly named Load This)	Loads debug information for the selected
	Find the Bootstrap processor (BSP)	Searches for the BSP in all hardware thre
	Step Into	Executes the current line, including any r
	Step Over	Executes the current line, following exec
	Step Return	Continues execution to the end of the cu

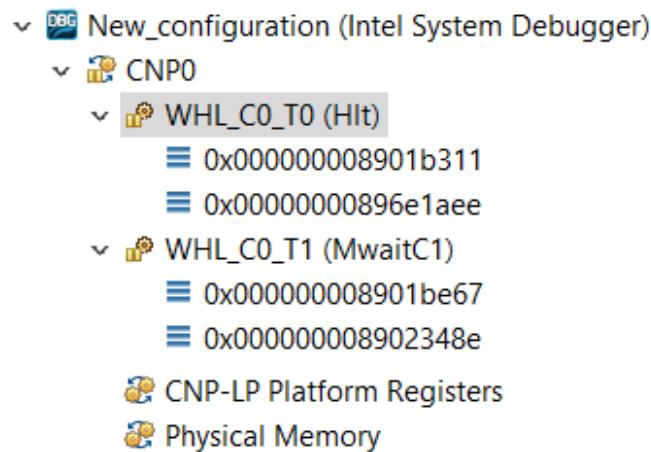
You can also execute multiple steps in a row with a [scripting solution](#).

For solutions on fixing target run control issues, refer to the [Troubleshooting](#) section.

CPU Operation Modes

When the target is suspended, you can see the CPU operating modes for a particular thread. Select the thread in the [Debug view](#) and see the mode information in the status bar at the bottom:

The list below shows available modes:



WHL_C0_T0 Modes: 64-bit

- **Protected** - the native operating mode of the processor.
- **Real** (or Real-address) - the mode that provides the programming environment of the Intel 8086 processor with extensions.
- **Virtual86** (or Virtual-8086) - a quasi-operating mode that allows the processor execute 8086 software in a protected, multitasking environment.
- **Compatibility** - the IA-32e submodule that allows most legacy protected-mode applications to run unchanged.
- **64-Bit** - the IA-32e submodule that provides 64-bit linear addressing and support for physical address space larger than 64 GBytes.
- **VMXroot** - the mode for VMX root operations for Virtual Machine Monitors (VMM)
- **VMXguest** - the mode for VMX non-root operations for the guest software.
- **SMM** - System Management Mode that provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features.

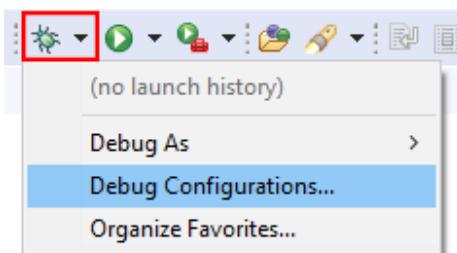
See also

[Intel\(R\) 64 and IA-32 Architectures Software Developer Manual Volume 3: System Programming Guide](#)

Filter Hardware Threads

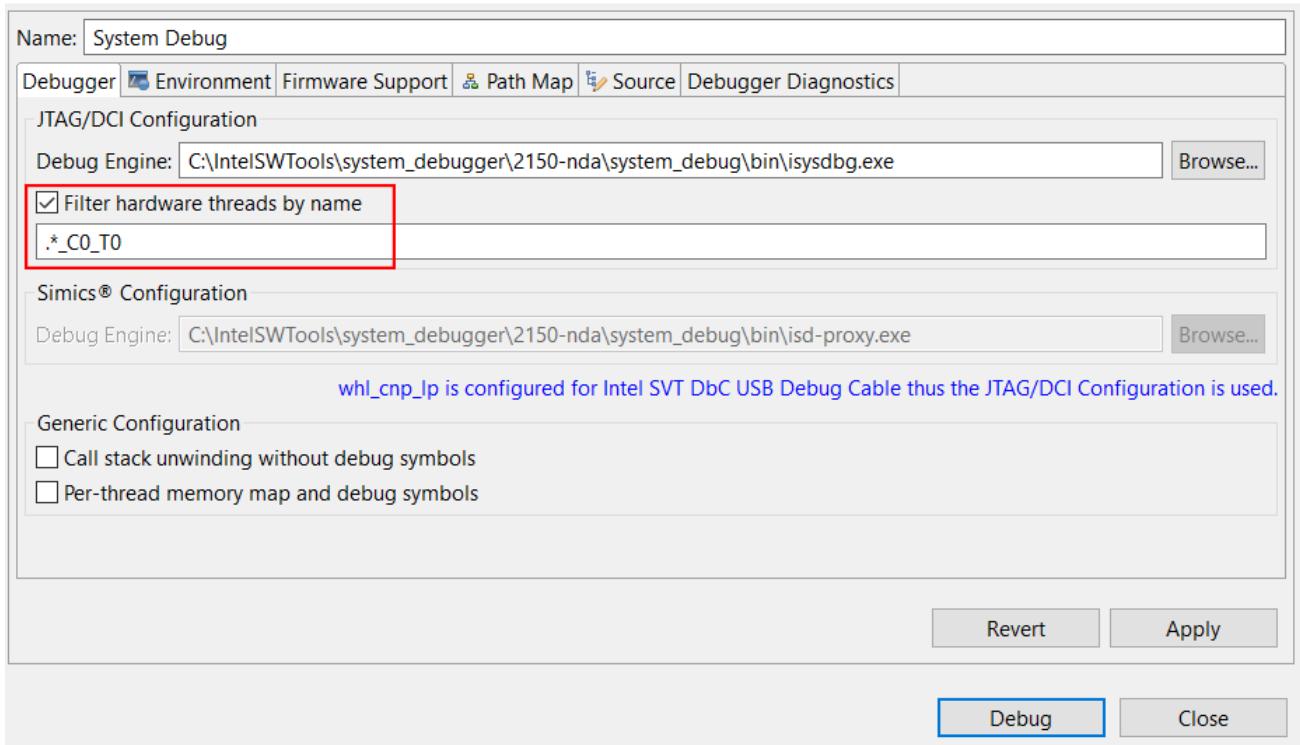
If you want only certain hardware threads to be displayed in the **Debug view** (for example, only the first thread of the first core), you can filter threads by name as follows:

1. Halt the target and open the **Debug Configuration** menu:



If you have not created a debug configuration yet, follow the steps [here](#).

2. Open your debugger configuration. Under **Debugger** tab, check the **Filter hardware threads by name** box.



3. In the text field, specify the regular expression to match the desired thread names.

Example:

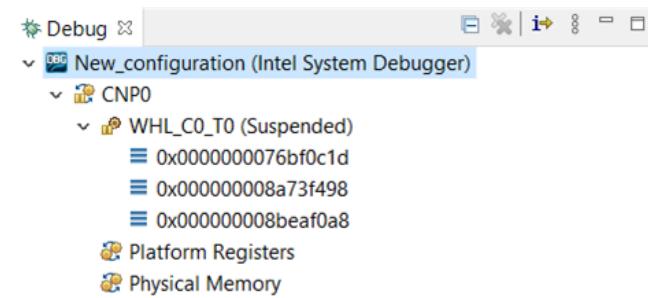
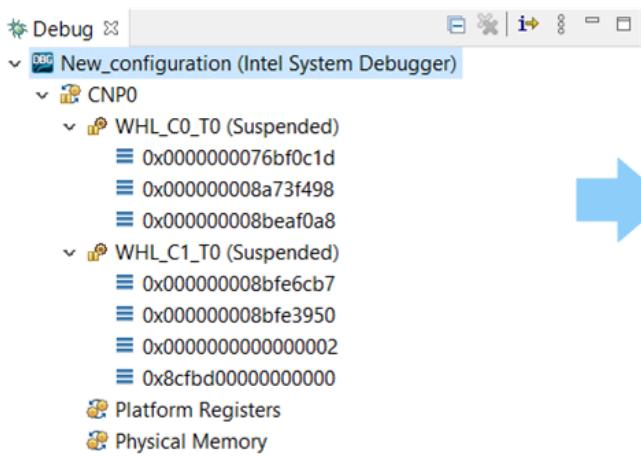
- `.*_C0_T0` matches the first thread of the first core.
- `.*_C0_T.` matches all threads of the first core.

4. Click **Apply** and **Debug**.

5. Resume the target and check the threads displayed in the **Debug view**.

To change the filter setting, you need to halt the target before editing the debug configuration - it is not updated dynamically.

See an example of initial and filtered hardware threads displayed in the **Debug view** below:



Loading Debug Information

The Intel(R) System Debugger allows you to automatically load debug information for the executing target software.

Important

Module information found via automatic loading methods (in the table below, all columns except the last one) is read-only and must not be edited in the GUI.

Depending on what you want to debug and with what toolchain (Microsoft Visual C++ (MSVC), GNU Compiler Collection(GCC)), the following features are supported:

Debuggee	Toolchain	Load PEIMs	Load EFI Images	Load Current
UEFI BIOS	MSVC	Yes	Yes	Yes
Intel(R) Slim Bootloader (Intel(R) SBL)	MSVC	Yes	No	Yes
Intel(R) Slim Bootloader (Intel(R) SBL)	GCC	N/A	N/A	N/A
coreboot*	GCC	N/A	N/A	N/A
Linux* Kernel Modules	GCC	N/A	N/A	N/A

This section describes the default and generic ways of loading debug information. For customization options, refer to [Customizing the Loading of Debug Information](#).

For loading debug information for UEFI boot phases, refer to the [Debugging UEFI BIOS](#) section.

Prerequisite: File Path Mapping

Before loading symbols, you should check if the paths to symbol files match the actual paths on your system. If they do not match, create a path mapping rule:

1. In the main Eclipse* toolbar, expand the drop-down menu next to the  Debug button and click **Debug Configurations**.

Alternatively, in the main Eclipse* menu bar, click **Run** and select **Debug Configurations**.

2. Click your debug configuration under the  Intel(R) System Debugger icon on the left.

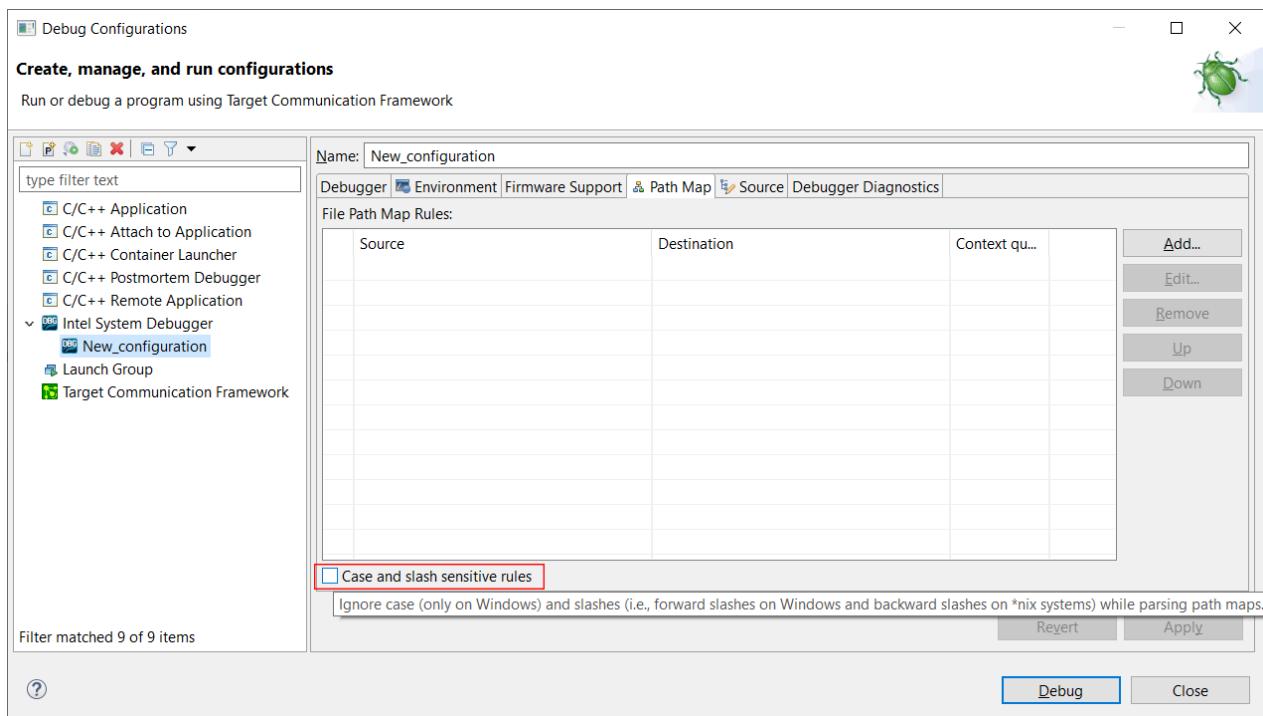
If you do not have existing debug configurations, follow the steps described in the [Launching the Debugger](#) section.

3. Open the  Path Map tab and click **Add**.

4. In the opened dialog box specify the path fragment used in the map file (Source) and the one used on your system (Destination).

Click **OK**.

5. By default, path mapping rules ignore the symbol case (only on Windows*) and slash directions used in the file path. If you want to change this behavior, check the **Case and slash sensitive rules** at the bottom.

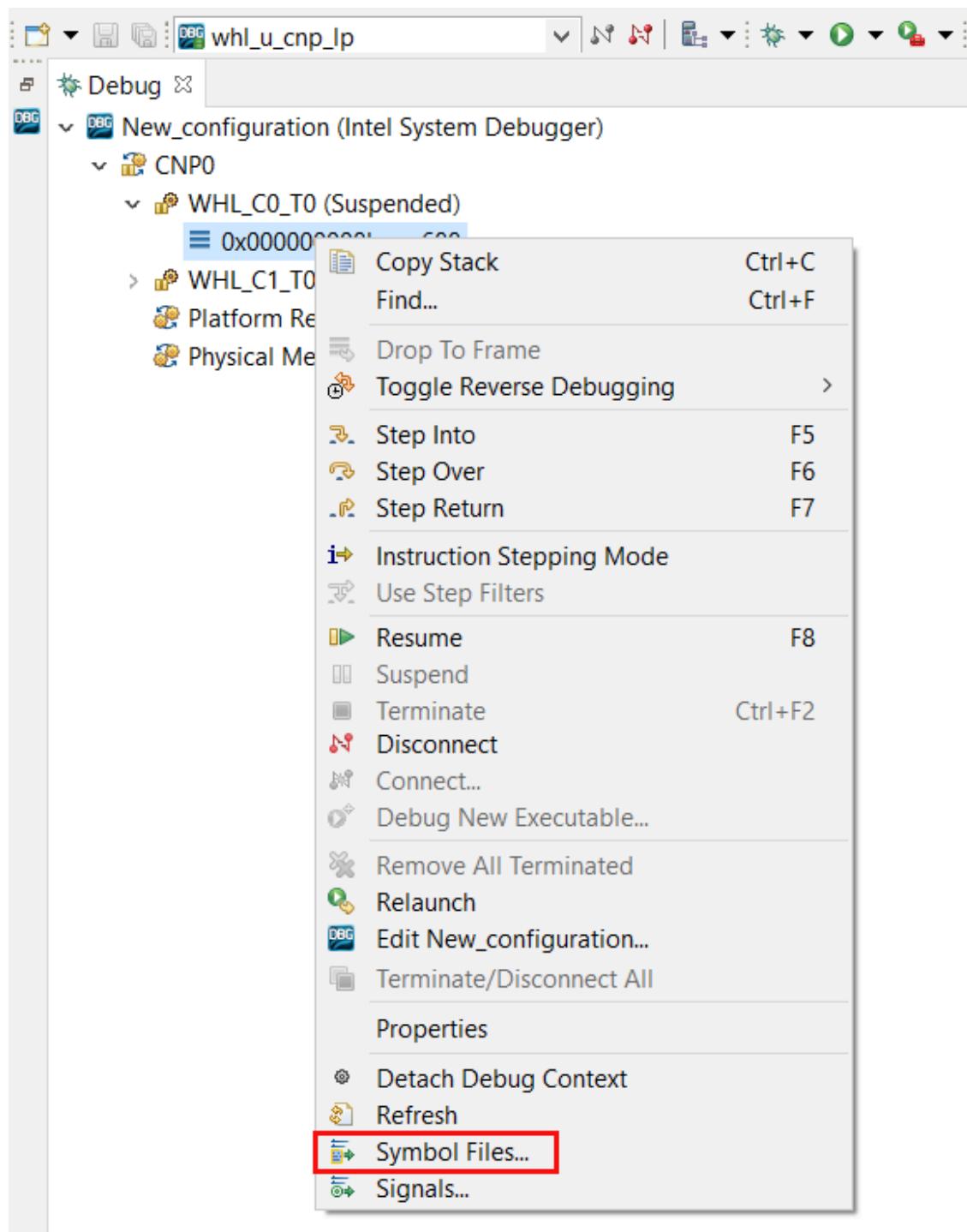


6. When you finish adding path rules, click **Apply** on the bottom of the tab.

Manual Loading

To load debug information manually, do the following:

1. Right-click the desired debug context in the **Debug view** and select **Symbol Files**.



2. In the opened dialog box, click **Add** or **Edit** to add a new or configure existing symbol file.
3. In the opened **Symbol File** dialog box, specify the following:

- The path to the symbol file to be loaded.
- (Optional) The memory address where the file is mapped.

When you are finished, click **OK**.

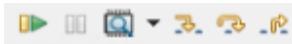
4. When all required symbol files are added, click **OK**.

If you want to load debug information for all threads at once, refer to the [scripting sample](#).

Automatic Loading

The automatic method is based on scanning the memory for PE/COFF module headers.

To load debug information automatically, select the desired debug context in the [Debug view](#) and click the  Load the Current Module (formerly named **Load This**) button in the Eclipse* toolbar:



The selected context defines the address range that will be searched for debug information:

- If you select a  hardware thread, the memory scan will start from the program counter of the thread.

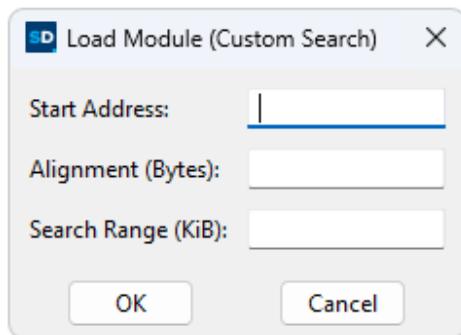
The value is equal to the program counter of the first stack frame of the thread.

- If you select a  stack frame, the start point will be its program counter (displayed in the view).

Important

Clicking **Load the Current Module** results in loading the debug information only for the currently executing EFI module (at the time of the target halt). If you want to load debug information for multiple modules, use [map files](#) with required debug information.

For fine-tuning the search parameters, select **Load Module (Custom Search)**. If left empty, any argument will take the default value.



If the search is successful, you should see the following changes:

- In the [ISD Shell](#), the following information is displayed under the  tag:
 - Memory addresses that the debugger searched between.
 - The detected module, its address, and its size in bytes.
 - The file name of the loaded module, for which a memory mapping is created.
- In the [Modules view](#), a new entry is displayed specifying the path, the address, and the size of the loaded file.
- In the [Debug view](#), new stack frames are displayed under the selected debug context.

Note

During one execution of **Load the Current Module**, the debugger loads only one file and stops searching for other modules after the first one is found.

For details on fixing issues with symbols loading, refer to the [Troubleshooting](#) section.

Intel(R) Slim Bootloader (Intel(R) SBL) Support

Intel(R) System Debugger supports Firmware with Terse Executable (TE) sections such as Intel(R) Slim Bootloader (Intel(R) SBL). To enable automatic loading of debug information with the Intel(R) SBL supported, do the following:

1. Open the environment file for editing:

- For Windows* OS: `<install_dir>/system_debugger/<version>/env.d/win32/*-isysdbg-env.bat`
- For Linux* OS: `<install_dir>/system_debugger/<version>/env.d/linux/*-isysdbg-env.sh`

2. Add the following environment variables:

- `ISYSDBG_FLASH_BASE` - set it to the base address for the flash memory, in hex format.
- `ISYSDBG_FLASH_SIZE` - set it to the total size of flash memory, in hex format.

3. Save the file.

Now you can use the  Load the Current Module button to load debug information automatically.

For more information, see the [Intel\(R\) Slim Bootloader \(Intel\(R\) SBL\) product page](#).

Loading BIOS ROM File

If you want to debug BIOS, the recommended way to load debug information is by parsing a ROM file (`.rom` or `.bin` file generated during BIOS build).

To load the ROM file:

1. In the main Eclipse* toolbar, expand the drop-down menu next to the  Debug button and click **Debug Configurations**.

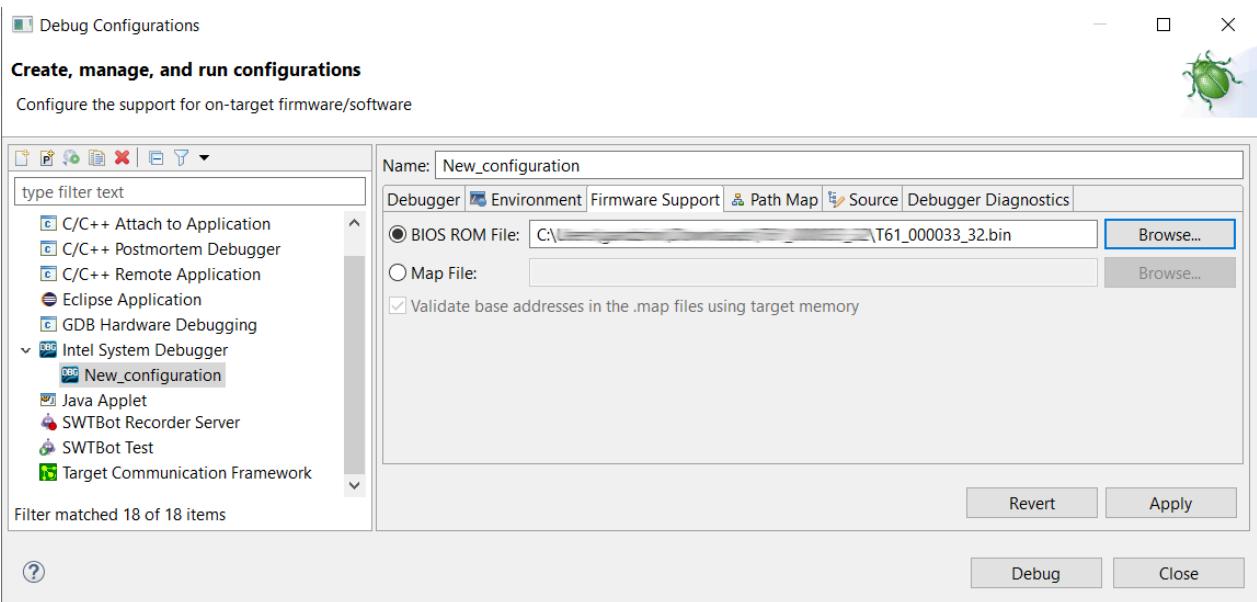
Alternatively, in the main Eclipse* menu bar, click **Run** and select **Debug Configurations**.

2. Click the debug configuration under the  Intel(R) System Debugger icon on the left.

If you do not have existing debug configurations, follow the steps described in the [Launching the Debugger](#) section.

3. Navigate to the **Firmware Support** tab.

4. To load the ROM file, select **BIOS ROM File** radio button, click **Browse**, and select the required `.bin` file.



5. Click **Apply** and then **Debug** to launch the debugger.
6. Verify that the loaded information in the Modules view.

Using Map Files

To load debug information for modules, you can specify map files that contain module information.

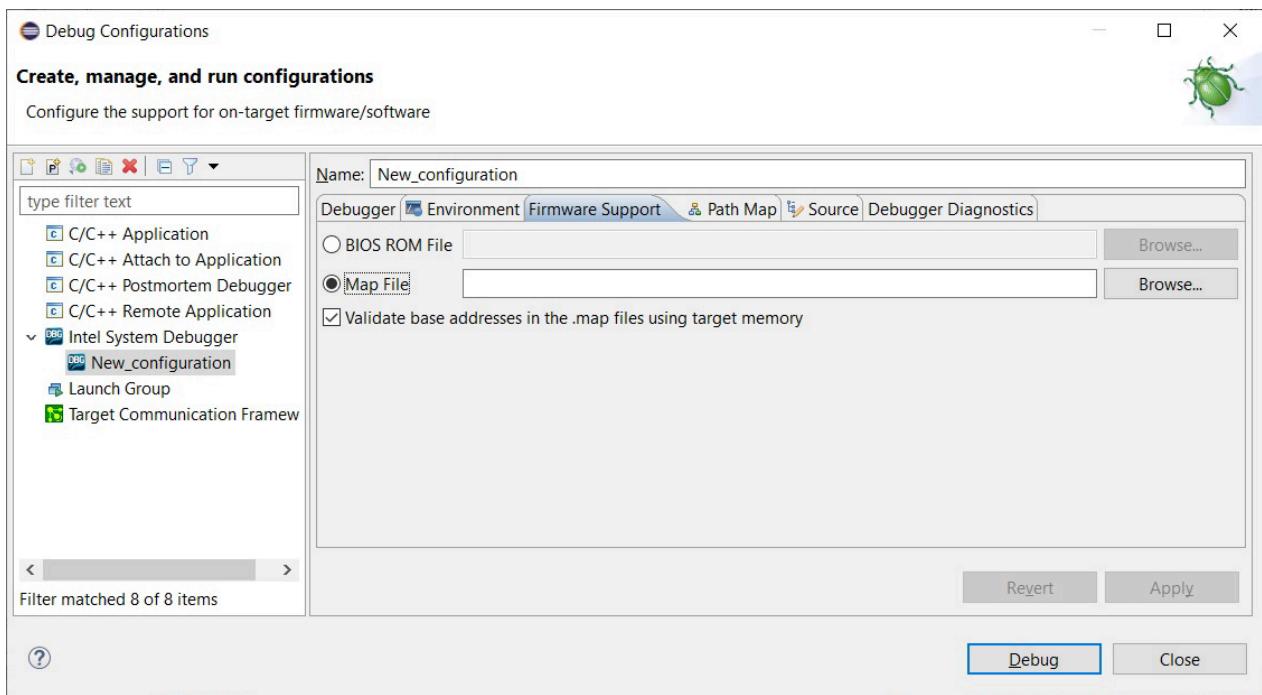
1. In the main Eclipse* toolbar, expand the drop-down menu next to the Debug button and click **Debug Configurations**.

Alternatively, in the main Eclipse* menu bar, click **Run** and select **Debug Configurations**.

2. Click the debug configuration under the Intel(R) System Debugger icon on the left.

If you do not have existing debug configurations, follow the steps described in the [Launching the Debugger](#) section.

3. If the paths in the .map file do not match the actual paths on your system, you need to [create a path mapping rule](#).
4. Navigate to the **Firmware Support** tab.
5. To load the map file, select **Map File** radio button, click **Browse**, and select the required files.



6. Choose the desired with with **.map** extension and click **Open**.
7. Ensure that the **Validate base address** box is checked. With this option enabled, headers in target memory are verified to ensure that the base addresses defined in a map file for modules are correct.
8. Click **Apply** and then **Debug** to launch the debugger.
9. To verify that the information from your map file has been loaded successfully, open the **Modules** view and ensure that all loaded **.efi** files are displayed correctly. See the example output below:

File Name	Address	Size
N:\targetsw\...efi	0x0fef450d	0x00000cc0
N:\targetsw\...efi	0x0fef45ca	0x00005940
N:\targetsw\...efi	0x0fef4b6a	0x00001960
N:\targetsw\...efi	0x0fef4cf2	0x00002e80
N:\targetsw\...efi	0x0fef4fcfa	0x00001020
N:\targetsw\...efi	0x0fef50ba	0x000006a0
N:\targetsw\...efi	0x0fef5113	0x00007920
N:\targetsw\...efi	0x0fef5894	0x00003480
N:\targetsw\...efi	0x0ffce914	0x00004a60

Loading Linux* Kernel Modules

Intel(R) System Debugger allows you to load Linux* kernel symbols and any desired modules including ones you compile separately (out-of-tree modules).

Important

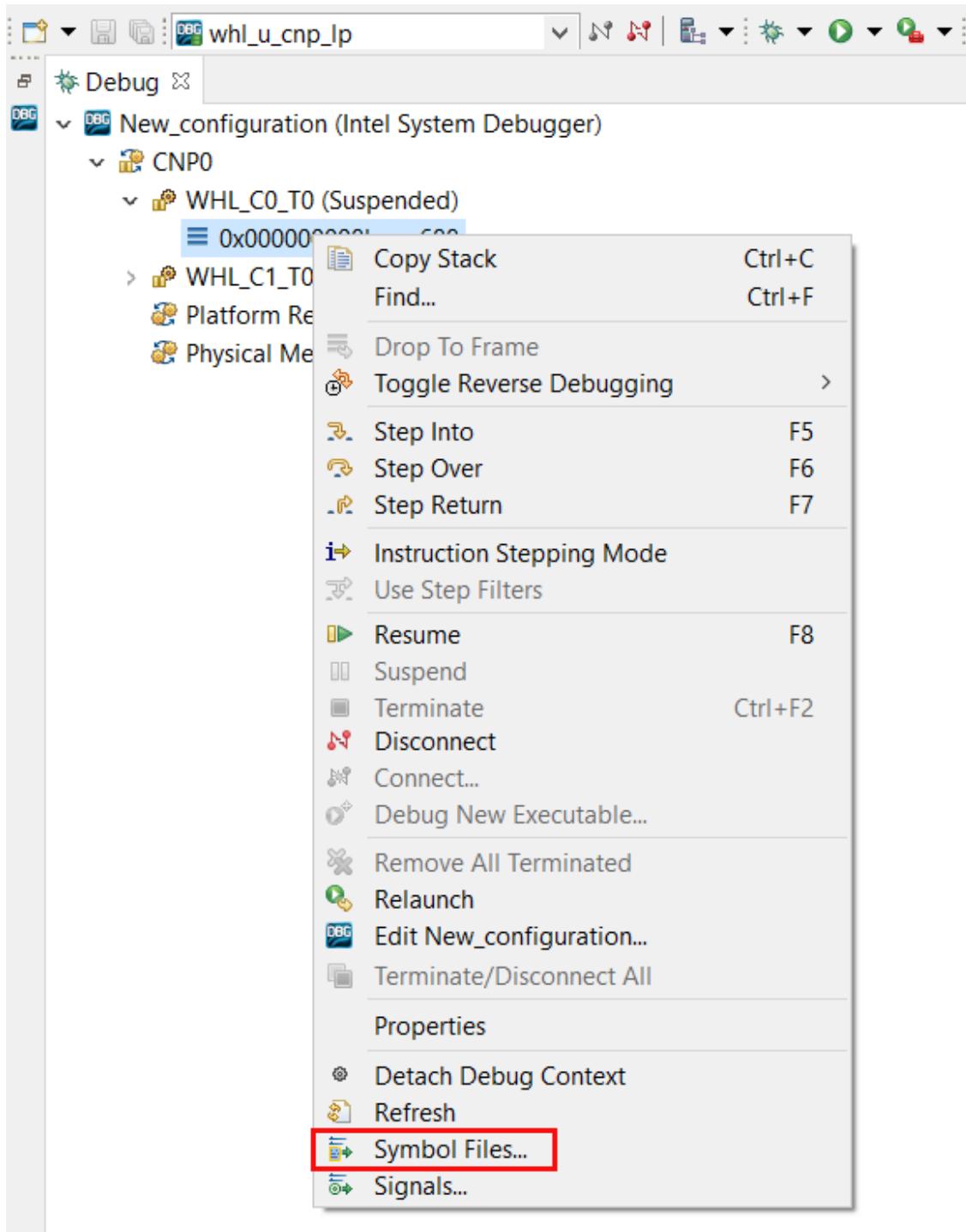
Ensure that kernel has been compiled without Kernel Address Space Layout Randomization (KASLR). On versions where the compile-time configuration option is not available, KASLR can be disabled by specifying the **nokaslr** kernel parameter.

To load Linux* kernel modules:

1. Halt the target.

2. Load the Linux* kernel (`vmlinuz`):

1. Right-click a desired hardware thread in the **Debug view** and select **Symbol Files**



2. In the opened dialog box, click **Add** and specify the path to `vmlinuz`.

3. Click **OK** and exit the dialog.

3. Once the symbols are ready, follow any of the methods below to load modules (you can combine different methods):

- To load module files located next to the kernel image:

Expand the drop-down menu of the  **Load the Current Module** button and select  **Load Linux Kernel Modules from Kernel Image Folder**.

The debugger locates the path where the kernel image resides and starts scanning memory for module names.

- To load module files located in a different directory:

- Expand the drop-down menu of the  **Load the Current Module** button and select  **Load Linux Kernel Modules from Folder**.
- Navigate to the folder where the `modules.order` file resides and click OK.

- To load out-of-tree modules (for example, the ones compiled separately and not included into `modules.order`):

- Expand the drop-down menu of the  **Load the Current Module** button and select  **Load Linux Kernel Modules from Folder**.
- Navigate to the folder where the `.ko` files reside and click OK.

If no `.ko` files are found in the specify location, the debugger tries to scan for the `modules.order` file.

4. When the scan is completed, all currently running modules are displayed in the [Modules view](#).

Break at the Initialization of a Linux* Kernel Module

Follow the steps below to break at the initialization of an out-of-tree (separately compiled) Linux* kernel module.

Prerequisites:

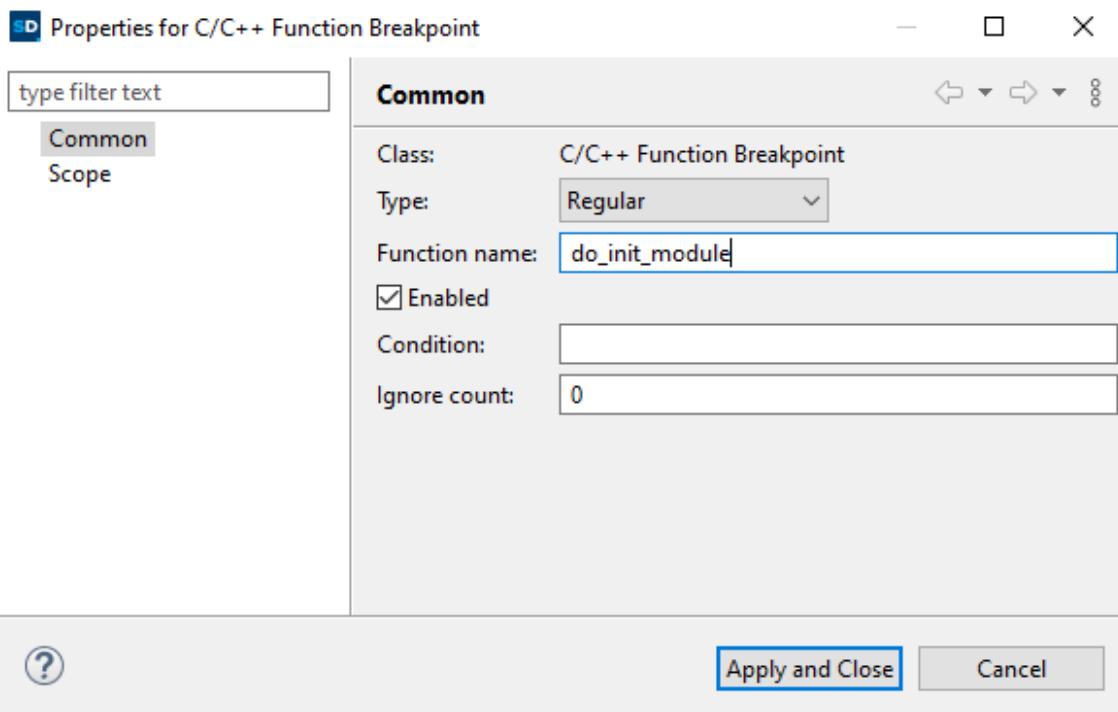
- The kernel has been built with Kernel Space Layout Randomization (KASLR) disabled, or when the compile-time configuration option is not available, the kernel is run with the `nokaslr` parameter.
- Target connection is successfully established.

1. Halt the target.

2. Load Linux* kernel symbols. Follow the instructions above.

3. Set a function breakpoint on the kernel initialization function:

- In the Breakpoints view, click  View Menu and select **Add Function Breakpoint**.
- In the opened dialog box, select the **Regular** breakpoint type and specify `do_init_module` as the function name.
- Click **Apply and Close** to save changes.

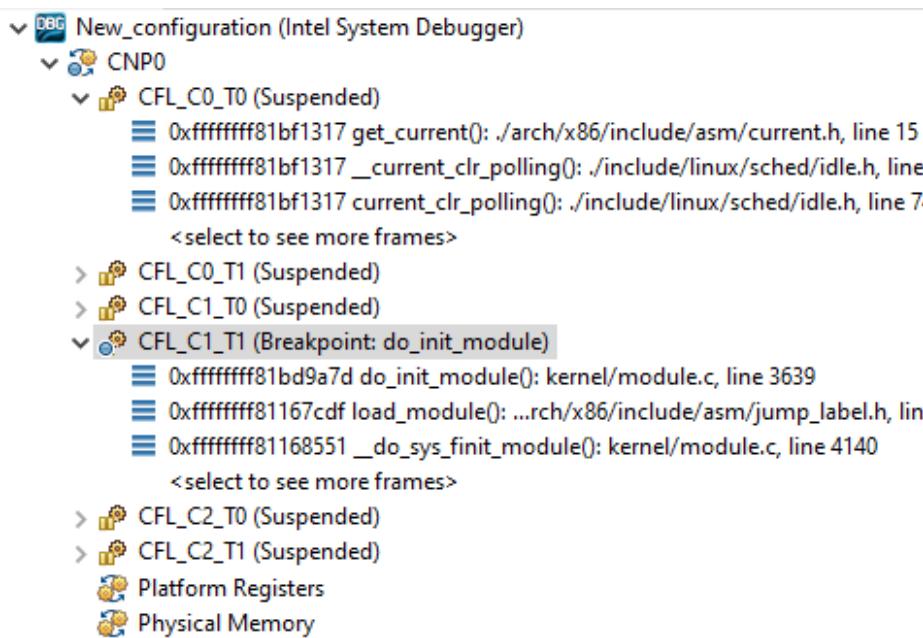


4. On the target system, launch the shell and execute the following command to insert an out-of-tree module:

```
sudo insmod <module>
```

where `<module>` is the target out-of-tree module (`.ko` file)

5. Return to the Intel(R) System Debugger running on the host system, resume the target, and wait for the breakpoint to get hit.



6. Once the target is halted, inspect the mod parameter of the function in any of the following ways:

- In the Variables view, expand the mod node, locate the init row, and copy its address from the Value column.

Name	Type	Value
mod	struct module *	{state=1, list={nex...
(x)= state	enum module_state	MODULE_STATE_COMING
> list	struct list_head	{next=0xfffffffffa0e...
> name	char [56]	"kdummy\000\000\000...
> mkobj	struct module_kobject	{kobj=(name=0xfffff8...
> modinfo_attrs	struct module_attribute *	{attr=(name=0xfffff...
version	const char *	0x0000000000000000
> srcversion	const char *	"CE788C6637C108C8F1...
> holders_dir	struct kobject *	{name=@fffff823...
syms	const struct kernel_symbol *	0x0000000000000000
etable	struct exception_table_entry *	0x0000000000000000
init	int (*)()	0xfffffffffa0cb2060
> core_layout	struct module_layout	{base=0xfffffffffa0c...
> init_layout	struct module_layout	{base=0xfffffffffa01...

- In the Expressions view, add the following expression: `mod->init`

Name	Type	Value
&do_init_module	int (*) (struct module *)	0xffffffff81bd9a77
mod->init	int (*)()	0xfffffffffa0cb2060
+ Add new expression		

7. Set a breakpoint at the address obtained in the previous step:

- In the Breakpoints view, click Add Breakpoint.
- In the opened dialog box, select the **Instruction Breakpoint** type and insert the address.
- Click **OK**.

8. Resume the target.

9. Once the breakpoint is hit at the module initialization step, you can load the module's symbols using the **Load Linux Kernel Modules from folder** button (see instructions in the previous topic).

You can automate the described process using the [sample script](#).

Loading Individual Modules

You can load any module individually by calling the following function:

```
threads[N].load_symbol_file(filename="<filename>", [address, size, offset, files])
```

where

- *N* is the ordinal number of a hardware thread
- *filename* is the name of a file to load (for example, an [.efi](#) file). Use the *filename* or *files* parameter, not both.
- *address* (optional) is the location of the file in the target memory
- *size* (optional) is the size (in bytes) of the image
- *offset* (optional) is the integer offset between the base address of the module in the target memory and the link-time address (created during the file compilation)
- *files* (optional) is the [BinaryFile](#) object or a list of [BinaryFile](#) objects to load. Use the *filename* or *files* parameter, not both.

Examples

1. The debugger loads the specified module using the link-time address.

```
threads[0].load_symbol_file(filename="my_vmlinu")
```

2. The debugger loads the specified module using the absolute address passed.

```
threads[0].load_symbol_file(filename="my_vmlinu", address=0xffffffff1234)
```

3. The debugger loads the specified module using the offset on top of the link-time address.

```
threads[0].load_symbol_file(filename="my_vmlinu", offset=0x1234)
```

4. This function throws an exception because *address* and *offset* parameters are mutually exclusive.

```
threads[0].load_symbol_file(filename="my_vmlinu", address=0xF0CA, offset=0xcafe)
```

See also

[Create simple CLI commands for frequent debug tasks](#)

Using Breakpoints

Follow the instructions presented in this section to set breakpoints of different types while using the Intel(R) System Debugger.

For the list of icons representing breakpoint types, refer to the [Breakpoints view](#).

Breakpoint Status Indicators

Depending on the current breakpoint status, the icons in the [Breakpoints view](#)

- The icon is greyed out - the breakpoint is disabled. To enable it, click the checkbox next to the breakpoint.
- The icon is colorful and the blue tick is present on the bottom-left - the breakpoint is enabled and active.
- The icon is colorful but the blue tick is missing - the breakpoint is unchecked and is not active. It usually happens after a system reset. To fully enable the breakpoint, click the checkbox next to the breakpoint twice - to disable and enable it back.

Additionally, you can see the breakpoint status report. Right-click the breakpoint in the [Breakpoints view](#), select **Breakpoint Properties**, and check the **Status** pane.

Setting a Breakpoint on a Line or Function

To set a breakpoint, halt the target and click the required line or function in the source code. Alternatively, follow these steps:

1. Halt the target by pressing the Suspend button.
2. In the [Breakpoints view](#), click View Menu and select **Add Line breakpoint** or **Add Function Breakpoint**.
3. Choose the breakpoint type:
 - **Regular** - to let the tool automatically choose the type.
 - **Hardware**
 - **Temporary** or **Hardware Temporary** - to remove the breakpoint after the hit.
4. For a line breakpoint, specify the path to the source file and its line number.

For a function breakpoint, specify the function name.

5. Set additional parameters:
 - **Enabled** box (checked by default) - uncheck if you want to keep the breakpoint disabled now
 - **Condition** - a custom condition for hitting the breakpoint.

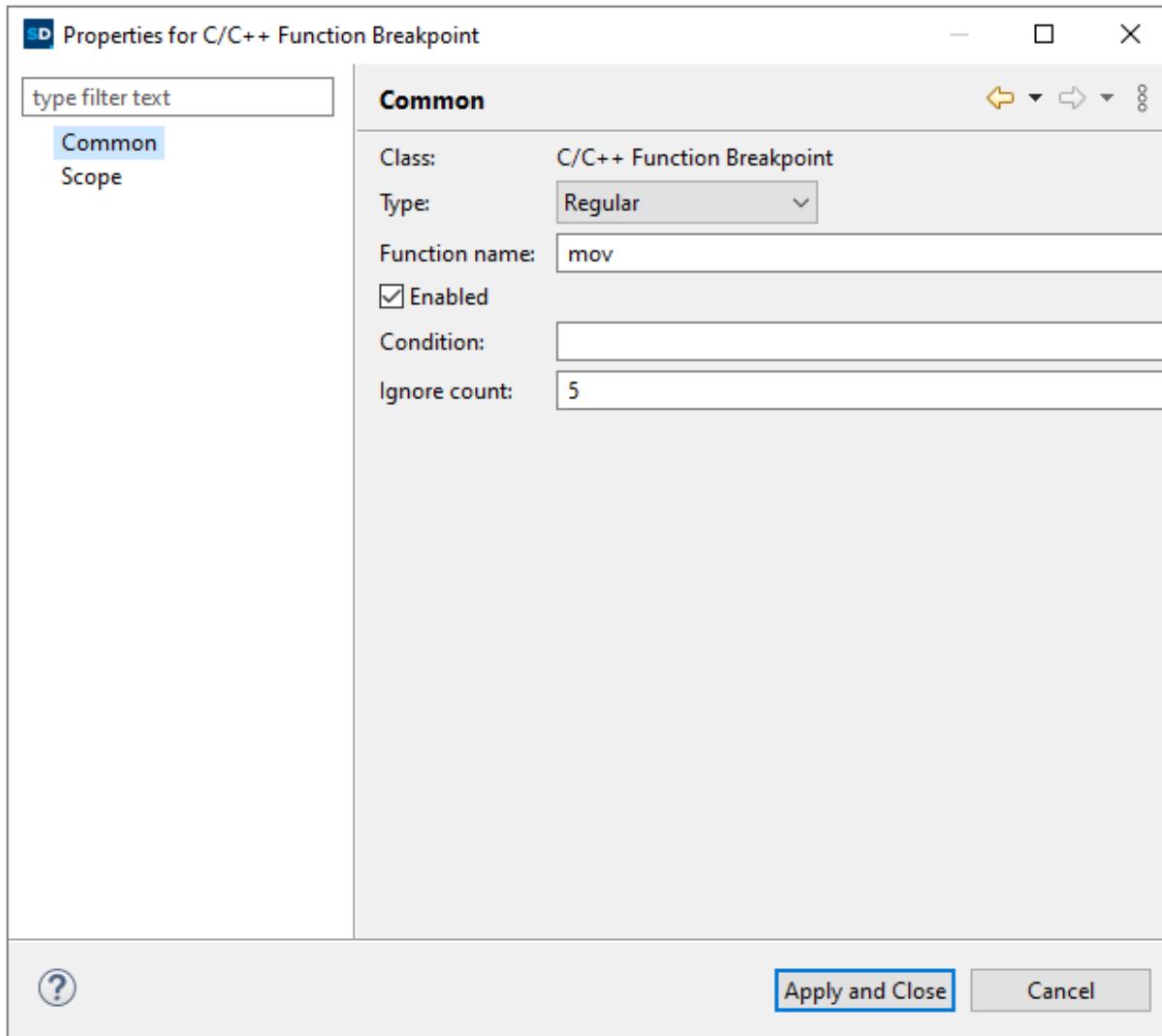
Example: condition `var == 3` determines that the target will be suspended only if the variable `var` equals 3 when the breakpoint is met. See [Setting Breakpoint Conditions](#) for a complete description of the allowable syntax.

- **Ignore count** - the number of times a breakpoint hit must be ignored.

Example: if the parameter equals 5, the target will be suspended only when the watchpoint is hit the 6th time.

6. Click **Apply and Close**.

See an example of a function breakpoint below:



Now you can resume the target and see how your breakpoints work. When (or if) the breakpoint is hit, the target gets halted.

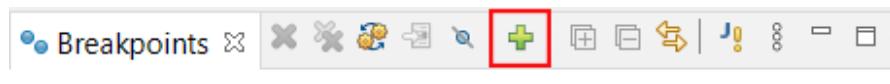
Setting a Breakpoint on an Instruction Address

To set an address breakpoint on an assembler instruction, halt the target and double-click the marker bar located in the left margin in the [Disassembly view](#). A new software breakpoint is created by default and displayed in the [Breakpoints view](#). You can also see information about a particular breakpoint by hovering over its icon in the beginning of a corresponding code line.

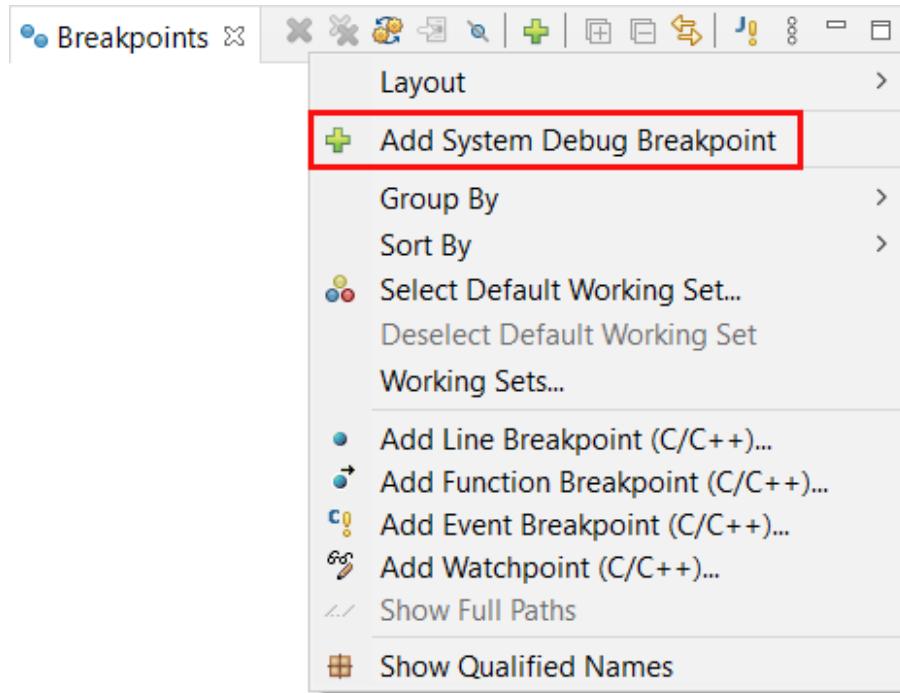
Alternatively, follow these steps to set an address breakpoint:

1. Halt the target by pressing the Suspend button.

2. In the [Breakpoints view](#), click the Add Breakpoint button.



Alternatively, click View Menu and select Add System Debug Breakpoint.

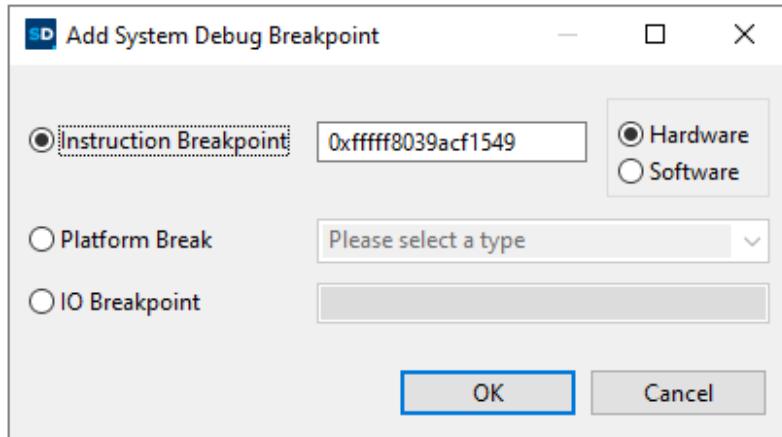


3. In the opened dialog box, select the **Instruction Breakpoint** radio button and specify the breakpoint address.

4. Choose the breakpoint type: software or hardware.

5. Click **Ok**.

See an example of an instruction breakpoint below:



Now you can resume the target and see how your breakpoints work. When (or if) the breakpoint is hit, the target gets halted.

You can make jump to the specified address in the [Disassembly view](#). To do it, right-click the address breakpoint in the [Breakpoints view](#) and select Go to Disassembly. The Disassembly view is refreshed and the specified address is displayed at the top.

Setting Watchpoints

Watchpoint is a special breakpoint type that suspends the program execution whenever the value of a specified *expression* (variable, memory address, and so on) changes.

Note

Watchpoints cannot track a **local** variable by its **name** as it exists only within a particular function. When the function returns, a non-static local variable is de-allocated from the stack and a watchpoint on it starts watching a bogus memory (stack) address. See the tip in step 3 for a workaround.

To set a watchpoint on an expression:

1. Halt the target by pressing the Suspend button.
2. In the [Breakpoints view](#), click View Menu and select Add Watchpoint (C/C++).
3. In the opened dialog box, specify the expression to watch: a memory address or a global variable name.

Tip

To watch a local variable, specify its virtual memory address instead of its name.

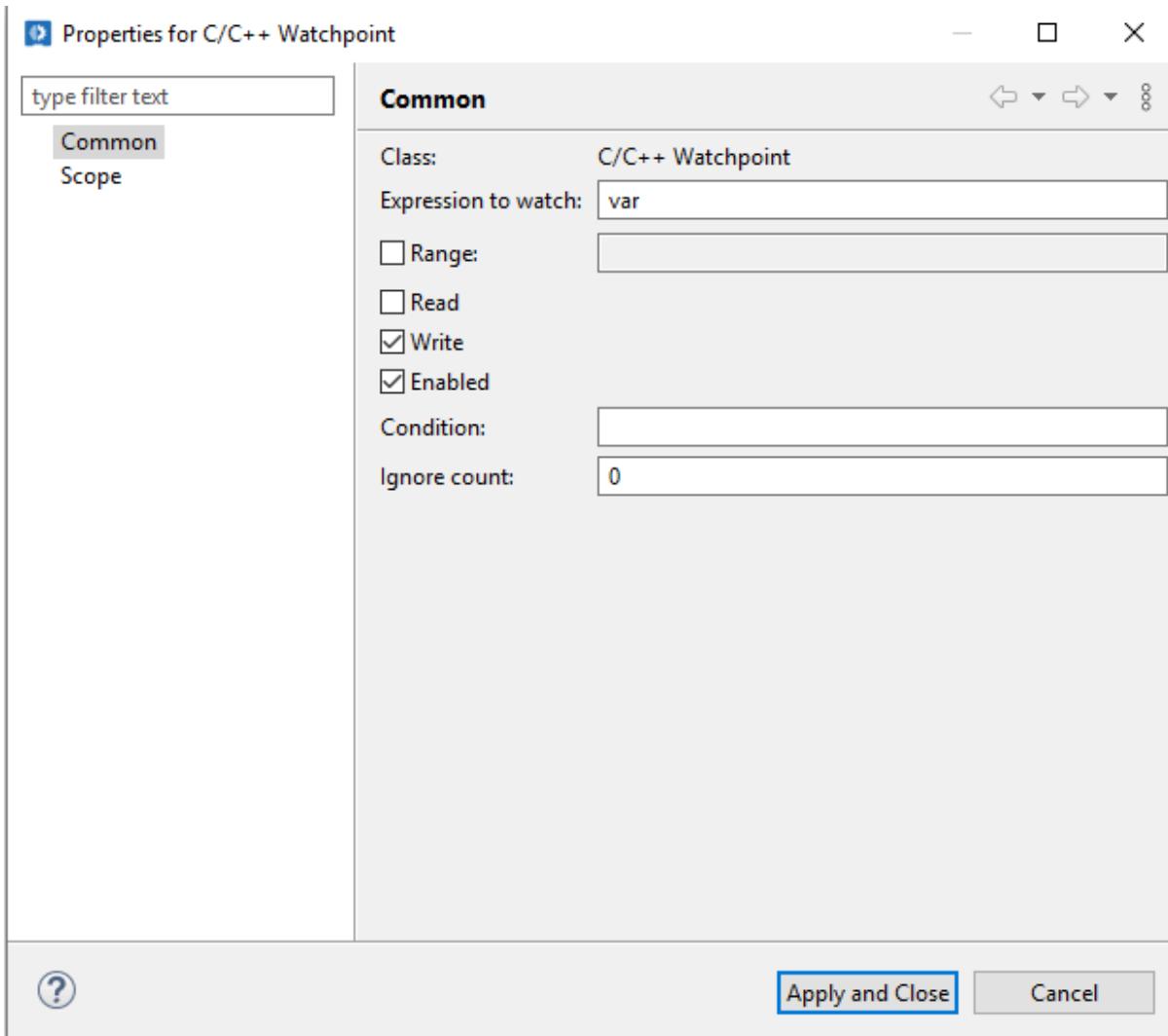
4. Set the watchpoint parameters:

- **Read** box - check to suspend program execution when the expression is read.
- **Write** box - check to suspend program execution when the expression is written to
- **Enabled** box (checked by default) - uncheck if you want to keep the watchpoint disabled now
- **Condition** - a custom condition for hitting the watchpoint.

Example: condition `var == 3` determines that the target will be suspended only if the variable `var` equals 3.

- **Ignore count** - the number of times a watchpoint hit must be ignored.

Example: if the parameter equals 5, the target will be suspended only when the watchpoint is hit the 6th time.



5. Click **Apply and Close**. The watchpoint is added to the [Breakpoints view](#).

The basic icon image depends on the specified mode:

- Read-only mode.
- Write-only mode.
- Read-write mode.

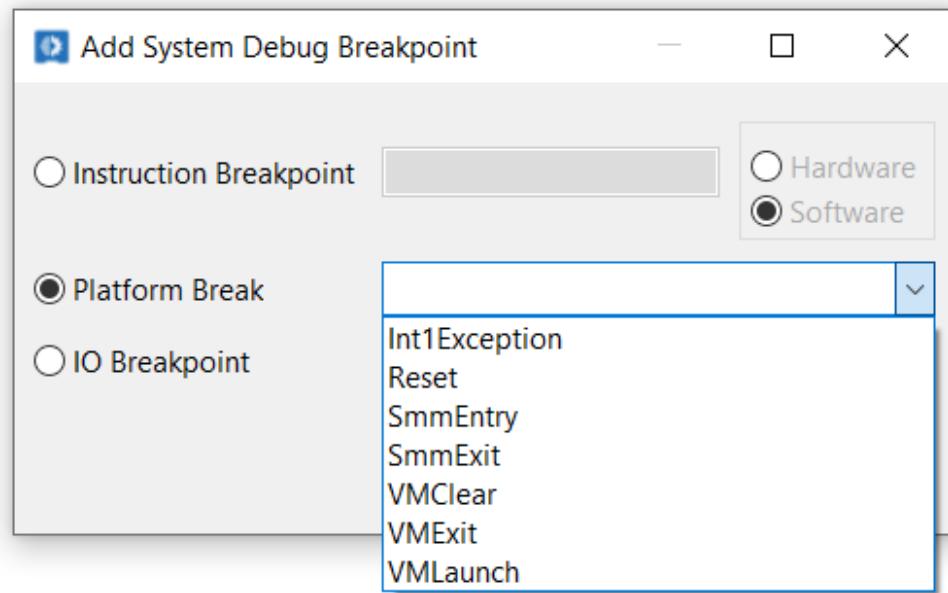
Setting Platform Breaks

The new Intel(R) System Debugger allows you to intercept system-level events, namely platform breaks. This type of breaks include events like system reset and start or end of the System Management Mode. To set a platform breakpoint:

1. Halt the target by pressing the Suspend button.
2. In the upper-right corner of the [Breakpoints view](#), click Add Breakpoint.

Alternatively, click View Menu and select Add System Debug Breakpoint.

3. In the **Add Breakpoint** dialog, select the **Platform Break** radio button.
4. In the **Platform Type** drop-down menu, select the break condition:



- To capture the system reset, select **Reset**.
- To capture the entry to the System Management Mode, select **SmmEntry**.
- To capture the exit from the System Management Mode, select **SmmExit**.
- To capture Virtual Machine Extensions (VMX) related events select:
 - **VMClear** to capture the event of clearing Intel(R) Virtual Machine Control Structure Shadowing (Intel(R) VMCS Shadowing)
 - **VMExit** to capture the exit from the virtual machine
 - **VMLaunch** to capture the virtual machine launch.

5. Click **OK**.

Now you can resume your target and see how the specified event is caught.

When the event is hit, the target is halted and the intercepted event is shown in parenthesis next to the corresponding hardware thread in [Debug view](#).

See also

[Intel\(R\) 64 and IA-32 Architectures Software Developer's Manual - Volume 3](#). See this manual for more information on VMX.

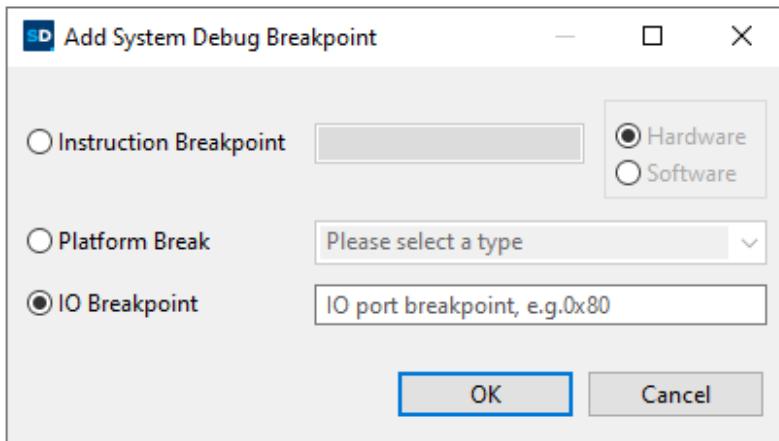
Setting the IO Port Access Breakpoint

To set a breakpoint for intercepting the IO port access event:

1. Halt the target by pressing the Suspend button.
2. In the [Breakpoints view](#), click the Add Breakpoint button.
3. In the opened dialog box, select the **IO Breakpoint** radio button and specify the breakpoint address.

Note

You can enter the address in a non-hex format and it will be automatically converted into the hex equivalent.



1. Click **OK**.

Now you can resume your target and see how the specified event is caught. When (or if) the breakpoint is hit, the target gets halted.

You can also view and edit IO port values using a [scripting solution](#).

Setting a Breakpoint at Module Entry

To add a breakpoint to intercept the module initialization event, follow the steps below:

Important

You can only add four breakpoints for module entries due to hardware breakpoint limitations.

1. Load necessary modules.

For instructions, see [Loading Debug Information](#) or [Debugging UEFI BIOS](#) (for loading PEI or DXE modules).

2. Halt the target and select the desired thread in the [Debug view](#).
3. Open the [Modules view](#) and check if all modules for the selected debug contexts are loaded successfully.
4. Right-click the desired module and select Add Breakpoint at Module Entry.
5. Open the [Breakpoints view](#) and check the newly added breakpoint.
6. Resume the target and wait until the breakpoint is hit. You can check the break location in the [Source view](#).

When (or if) the breakpoint is hit, the target gets halted.

Note

The debugger might break in the `__ModuleEntryPoint` function, which is one or more steps before the module main function. To jump to the proper function, use Step Into.

Setting Breakpoint Conditions

Breakpoint conditions can be specified by right-clicking the breakpoint in the [Breakpoints view](#), selecting **Breakpoint Properties**, and specifying the condition under **Condition**.

The syntax for breakpoint conditions supports standard C/C++ expressions together with the following extensions:

- CPU registers: \$, where is a register name, e.g. `$rax`. The allowable register names are architecture-specific.

Examples of valid expressions:

- `x > 0` (where is a variable)
- `f(x) && (y == 0)` (where f is a function and x and y are variables)
- `($rax == 0xa0) || ($rax == 0xa1)`

Viewing and Modifying Registers

Follow the instructions presented in this section to monitor and modify values of architectural and platform registers.

Architectural Registers

Architectural target registers (CPU registers) are displayed in the [Registers view](#).

To monitor the value of an architectural register:

1. Halt the target by pressing the Suspend button.
2. Open the [Registers view](#) and navigate to the desired register by expanding the register tree.
3. Right click the table row with register information and select **Watch In Expressions**.

Name	Hex	Decimal	Description	Mnemonic
rax	00000000...	0		
rdx	00000000...	0		
rcx	00000000...	0		
rbx	00000000...	160		
rsi	00000000...	0		
rdi			Select All	Ctrl+A
rbp			Copy Registers	Ctrl+C
rsp			Find...	Ctrl+F
r8			Watch In Expressions	
r9			Detach Debug Context	
r10			Refresh	
r11			Symbol Files...	
r12			Signals...	
r13			Display Register As Vector Of...	>
r14			View Memory	
r15	00000000...	0		

The [Expression view](#) opens displaying the selected register value.

To modify an architectural register:

1. Halt the target by pressing the Suspend button.
2. In the **Registers** view, navigate to the register you want to modify by expanding the register tree.
3. Under the **Hex** or **Decimal** column click the value you want to change and enter the new one.
4. The register tree is refreshed and the changed register is highlighted yellow.

Alternatively, you can use the **Value** column in the [Expression view](#) to modify the register value.

Viewing Registers for Different Hardware threads

To display registers associated with different hardware threads simultaneously, configure the Intel(R) System Debugger GUI the following way:

1. Halt the target by pressing the Suspend button.
2. Open the [Registers view](#).
3. By clicking the Open New View button, create as many **Registers** views as needed.
4. For each **Registers** view:
 1. In the [Debug view](#), select a hardware thread.
 2. In the [Registers view](#), click Pin to Debug Context to pin the view to the chosen hardware thread.

Platform Registers

You can interact with platform registers using the following views:

- [Platform Register Dictionary view](#) that lists all registers associated with a particular debug context.
- [Platform Register Watch view](#) that allows you to monitor register values during runtime.
- [Platform Register Editor view](#) that displays a graphical representation of a particular register.

You can start monitoring the value of a platform register from two views:

- From the [Platform Register Watch view](#):

1. Halt the target.
2. Open the register dictionary by clicking  Show Register Dict.

Remember that the dictionary only contains the registers associated with the debug context selected in the [Debug view](#).

3. Navigate to the register you want to add by expanding the tree or type the register name into the **Search** field.
4. Select the register and click **OK**.

- From the [Platform Register Dictionary view](#):

1. Halt the target.
2. Navigate to the register you want to add by expanding the tree or type the register name into the **Search** field.
3. Select the register and press  Add to Watch.

The [Platform Register Dictionary view](#) opens displaying the selected register value.

Working with Memory

To monitor process memory:

1. Open the [Memory Browser view](#).
2. In the search field, specify the memory you want to access by providing a particular address (for example, `0x00000000fffff0` for hexadecimal format) or an expression (for example, `$rsp + 0x10`).
3. Click **Go** or press **Enter**.

The memory data is displayed in a new tab within the view.

View Physical Memory

To view physical memory in [Memory Browser view](#), select  **Physical Memory** node in [Debug view](#).

See also

Importing and Exporting Memory

To export memory to a file:

1. In the [Memory Browser view](#), click  Export.
2. In the opened **Export Memory** dialog box, do the following:
 - In the **Format** drop-down menu, choose the output format.
 - Check the start and end addresses.
 - In the **Length** field, enter the size in bytes of the memory being exported.
 - In the **File name** field, specify the path to the output file.
3. Click **OK**.

The output file only contains the bits you specified.

To import memory from a file:

1. In the [Memory Browser view](#), click  Import.
2. In the opened **Import Memory** dialog box, do the following:
 - In the **Format** drop-down menu, choose the input format.
 - Check the restore addresses, which will handle the imported memory.
 - In the **File name** field, specify the path to the input file.
 - To scroll the view to display the specified address on top, check the **Scroll to restore address** box.
3. Click **OK**.

The changed memory bytes (if any) are highlighted red.

Collecting Execution Trace

Typical debugging workflows can be described as follows:

- Backwards traversal:
 - a. Determine the symptoms of the issue.
 - b. Place a breakpoint at the place where the symptoms become apparent.
 - c. Retrace the execution backwards by unwinding the call stack and checking callers.

Drawback: In some cases, the call stack might not provide enough information to identify the issue, because it might have originated in a separate execution branch.

- Forward traversal:
 - a. Determine the hierarchy of function calls that lead to the symptom of the issue.
 - b. Place a breakpoint at a function that leads to the issue.
 - c. Single-step the execution until the symptoms are apparent.

Drawback: Forward traversal might not work if you have to step through a large number of executions of a code piece to identify the issue. It can also be destructive to issues like race conditions, deadlocks, and others.

In such cases, execution trace can be helpful. Using execution trace, you can place a breakpoint at the location where the issues become apparent and review the code lines that have been executed up to this point.

Launch Trace Capturing

Intel(R) System Debugger supports the following methods for collecting execution trace:

- Last Branch Record (LBR) - a register-backed hardware feature with limited jump history.

! Note

Certain distributions of Linux* OS running on the target disable LBR, which does not allow Intel(R) System Debugger use this method for tracing. If you face such issue, contact your OS vendor or Intel.

- Intel(R) Processor Trace (Intel(R) PT) - a feature to that allows you to trace thousands of instructions and provides options for configuring the trace.

If you debug virtual target simulation, see [Collecting Execution Trace with Simics\(R\) Simulator](#).

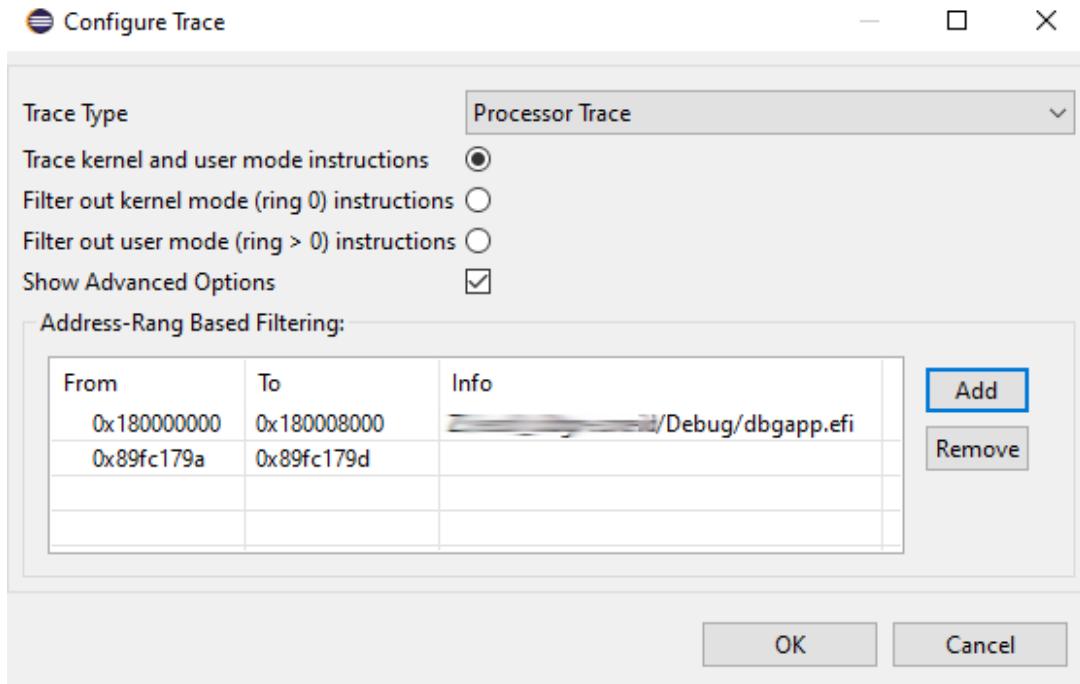
To record trace with Intel(R) System Debugger:

! Note

The target must be halted before configuring, starting, or stopping instruction tracing.

1. Halt the target and select the desired debug context in the [Debug view](#).
Each thread uses specific mechanisms for tracing.
2. Open the [Instruction Trace view](#) and click Configure Instruction Trace.
3. In the opened dialog box, select the desired method from the **Trace Type** drop-down list.
4. You can filter out kernel or user mode instructions or choose to trace all (default) using radio buttons.
5. If you have selected the Intel(R) PT feature, you can further customize trace parameters. Check **Show Advanced Parameters** box to expand the dialog.
 - a. To specify the search address range, click **Add** on the right.
 - b. In the new dialog box, select the method of defining the address range:

- Choose to trace instructions belonging to a specific module only. The information on modules currently loaded is displayed in the dialog.
 - Specify the address range manually by entering the values in hexadecimal format.
- c. Click **OK** to save changes.



- Click **OK** to close the configuration dialog and start trace capturing by clicking Start Instruction Trace.
- Resume the target.
- To stop trace capturing, halt the target and, in the [Instruction Trace view](#), click Stop Instruction Trace.
- Wait for the captured trace to be decoded.

If the console returns the message `ITrace: Processor Trace decoding timed out`, see the [troubleshooting page](#) for suggestions on how to fix the issue.

Instruction and Power Event Tracing Support

If your target platform supports `PTWRITE` instruction and Power Event tracing, you can get corresponding data in the captured trace results. The event payload is displayed in the **Event Description** column of the [Instruction Trace view](#).

If no event information is displayed, refer to the [Troubleshooting](#) section.

See also

[Intel® 64 and IA-32 Architectures Software Developer Manual, Volume 3C, section 36.2.3](#)
“Power Event Tracing”

Advanced Debugging Scenarios

This chapter describes additional use cases that you can use for certain debugging tasks. For basic instructions like using breakpoints, refer to the [Debugging Basics](#) section.

Executing External Python* Scripts

To import and execute external Python* scripts from an IPython shell provided by the [ISD Shell](#):

1. Launch the [ISD Shell](#).
2. Navigate to your script file location by executing the following script:

```
import os  
  
os.chdir('<directory_path>')
```

where `<directory_path>` points to the folder containing the Python script you want to execute.

3. Import your script as a Python module:

```
import <script_name>
```

where `<script_name>` equals your Python file name without extensions.

Now you can proceed with executing your script.

Automating Tasks on Launch

To modify the debugger boot process and specify a behavior to occur during the connection attempt:

1. Create a new `.py` file and define a function `on_launch(target)`.

For example, the following code snippet enables automatic loading of available debug symbols on launch:

```

_target = None
_threads = None

def on_launch(target):
    """
    Define in this function actions to be executed on Launch during
    the connection attempt. This function is automatically executed
    after the Sysdbg CLI is started. For example, global variables
    _target and _threads used in this module can be initialized.
    If defined, this function must have the definition above, so
    that it can be correctly executed: "def on_Launch(target)".

:param target: target variable on global scope
:type target: Session
:returns: None
"""

# If _target and _threads global variables are used in
# other functions in this module, initialize them here.
global _target
global _threads
_target = target
_threads = target.threads
# Now we Load available debug symbols on Launch.
for t in _threads:
    t.load_this()

```

2. Specify the path of the custom script in any of the following ways:

- If the **ISD Shell** view is used, set the environment variable `ISYSDBG_CLI_CUSTOM_SCRIPT` before launching the Intel(R) System Debugger.

```
ISYSDBG_CLI_CUSTOM_SCRIPT = <custom_script_path>/<custom_script_name>.py
```

The environment variable can be declared in the same terminal before launching the Intel(R) System Debugger, in the system environment variables, or in the environment file:

- For Windows* OS: `<install_dir>/system_debugger/<version>/env.d/win32/*-isysdbg-env.bat`
 - For Linux* OS: `<install_dir>/system_debugger/<version>/env.d/linux/*-isysdbg-env.sh`
 - If the ISD CLI is used, the custom script can be specified in the connect command as follows:
- ```
sysdbg.connect(custom_script="<custom_script_dir>/<custom_script_name>.py")
```
- If the Sysdbg CLI entry point is used, the custom script can be specified in the following way:
- ```
intel_sysdbg --custom_script="<custom_script_path>/<custom_script_name>.py"
```

See also

Customizing the Loading of Debug Information

The default way of loading debug information by pressing the  Load the Current Module (formerly named **Load This**) button scans the memory in a default way and loads symbols for a selected hardware thread only. However, you can customize the process the following ways:

- [Configure the scan for debug information](#) to use alternative scan parameters.
- Create a loop for [loading debug information for all threads at once](#).

For instructions on fixing issues related to this functionality, see the [Troubleshooting](#) section.

Configure Call Stack

You can allow the debugger to generate a call stack trace (unwind the stack) when no debug symbols are loaded. By default, only top-most stack frame will be shown if no debug symbols are loaded.

You can enable it by checking the **Call stack unwinding without debug symbols** box in the Debug Configuration editor (see [Launching the Debugger](#)).

Alternatively, you can enable it by setting the following environment variables:

1. Open the script located at `<install-dir>\system_debugger\<version>\env.d\win32*-isysdbg-env.bat`.
2. Add or uncomment the line:

```
set ISYSDBG_DBGHELP_PERMISSIVE_MEM_ACCESS=1
```

3. Additionally, you can change the call stack size. See an example below:

```
set ISYSDBG_DBGHELP_STACK_SIZE=0x1000
```

Debugging UEFI BIOS

The features provided by the Intel(R) System Debugger allow for UEFI BIOS source-level debugging.

To debug scenarios on this level, you can load symbols for UEFI boot phases:

- [Load BIOS ROM file](#).
- [Load Pre-EFI Initialization modules \(PEIMs\)](#).

Alternatively, you can use [map files](#) for loading PEIMs.

- [Load EFI images for DXE phase and onwards.](#)

Loading Pre-EFI Initialization Modules (PEIMs)

To load modules for Pre-EFI Initialization phase, follow the steps below:

1. Halt the target and ensure that it has stopped in the flash memory region 0xff600000 - 0x100000000.
2. Select a desired hardware thread in the [Debug view](#).
3. Expand the drop-down menu of the  Load the Current Module button and select **Load PEI Modules**.

The selected context defines the address range that will be searched for debug information:

- If you select a  hardware thread, the memory scan will start from the program counter of the thread.

The value is equal to the program counter of the first stack frame of the thread.

- If you select a  stack frame, the start point will be its program counter (displayed in the view).

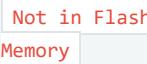
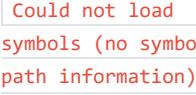
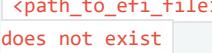
4. When the search and loading is completed, check the [Modules view](#) for the list of successfully loaded modules.

If the view is empty, no modules have been found or successfully loaded.

5. Launch the [ISD Shell](#) and check the error messages to determine possible issues.

If a module has been added successfully, the shell returns a corresponding message.

Error Messages

Error Message	Failure Cause
 Not in Flash Memory	The target has been halted too late in the UEFI boot sequence to search for PEI modules.
 Could not load symbols (no symbol path information)	The BIOS does not contain debug information.
 <path_to_efi_file> does not exist	The  .efi file does not exist or cannot be found by the debugger.

NOTE: A PEI module may be in compressed state in flash memory, and is only later decompressed and loaded into the DRAM. In this particular scenario, the module will not be discovered either by loading BIOS ROM file or Load Pre-EFI Initialization Modules. Hence, to load symbols for a compressed module, the user must manually track when the module is decompressed and loaded in

to the DRAM. As in the case of BIOS, the user can use break on *PeiDispatcher* function in *../Pei/Dispatcher/Dispatcher.c* and track when the module is loaded in DRAM. Then can use the address and load the symbol file manually.

Loading EFI Images for DXE Phase and Onwards

To load EFI images for later UEFI boot phase like DXE, follow the steps below:

1. Halt the target and ensure that it has stopped late enough in the UEFI boot sequence (not in flash memory region).
2. Select a desired hardware thread in the [Debug view](#).
3. Expand the drop-down menu of the  Load the Current Module button and select **Load EFI Modules**.

The tool searches for EFI images and attempts to load symbols for each of them. This takes more time than usual loading of debug information.

4. When the search and loading is completed, check the [Modules view](#) for the list of successfully loaded modules.

If the view is empty, no modules have been found or successfully loaded.

5. Launch the [ISD Shell](#) and check the error messages to determine possible issues.

If a module has been added successfully, the shell returns a corresponding message.

Error Messages

Error Message	Failure Cause	Solution
<code>DRAM was not initialized, cannot search for the ERI system table pointer</code>	The target has been halted too early, before DXE phase.	Resume the target and wa
<code>Could not load symbols (no symbol path information)</code>	The BIOS does not contain debug information.	Compile the modules with
<code>No EFI system table pointer found in memory</code>	The debugger could not find the <code>EFI_SYSTEM_TABLE_POINTER</code> in the default memory address range.	Configure the search usin,

Breaking on POST Codes

The execution of the BIOS also involves writing a specific code to the IO port address `0x80`, for instance, by means of the instruction `OUT 0x80, EAX`. The value written is called the POST (Power On Self Test) code. Breaking on a specific POST code can therefore be achieved by setting an **IO breakpoint** at the port address and specifying a **condition** that checks the value of the input register against the chosen code.

For the example, if the instruction for the IO write is `OUT 0x80, AX` and the code to be written is `0xa123`, this means adding an IO breakpoint with address equal to `0x80` and condition equal to `($rax & 0xffff) == 0xa123`. Since POST codes are normally 16-bit values, it is imperative to use the `0xffff` mask on the `RAX` register in the expression. As breakpoint conditions can be arbitrary expressions, custom ranges of codes can also be captured with conditions like `((($rax & 0xff00) == 0xa200) && (($rax & 0xff) > 0x20))`.

Note that IO port breakpoints are normally cleared by reset flows, so please see the [section on breakpoint restrictions](#).

Debugging System Management Mode

System Management Mode (SMM) is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

For more information, see the [Intel\(R\) 64 and IA-32 Architectures Software Developer Manuals](#).

Breaking on SMM Entry

Intercepting the SMM entry point to start debugging the SMM.

System Management Mode can only be entered through System Management Interrupt (SMI). To intercept SMI:

1. Halt the target by clicking the Suspend button in the main Eclipse* toolbar.
2. In the [Breakpoints view](#), set a platform break of type **SmmEntry** as described in [Setting Platform Breaks](#).
3. Resume the target by clicking the Resume button and wait till the SMI is hit.

If the SMM entry has not been intercepted, you can force SMI by writing to port 0xb2 on most platforms. Remember, the thread needs to be halted to execute the write operation on the IO Port. In the [ISD shell](#), run the command below and repeat step 3:

```
threads[0].io(0xb2, 1, 1)
```

Once the debugger has stopped at the SMM Entry break, it will automatically restore the debug registers. Updated registers are highlighted in the **Registers** view.

! Note

Software breakpoints cannot be set in SMM. Make sure you only set address breakpoints of type **Hardware**.

Accessing the SMRAM State Save Map

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel(R) 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H]. The Intel(R) System Debugger allows you to access register values of a particular thread.

To display values of all registers associated with a particular thread, run the following code in the [ISD shell](#):

```
threads[0].get_arch_registers()
```

You get a list of pairs (namely, tuples) representing a register name and its value.

For more information of the SMRAM State Save Map, refer to the [Intel\(R\) 64 and IA-32 Architectures Software Developer Manuals](#).

Debugging the Virtual Target Simulation

You can use Simics(R) software as a simulation target to work with Intel(R) System Debugger.

Prerequisites

- Java* 11 runtime (64-bit).

Once installed, it should be available in your `PATH` environment variable. Execute `java -version` in the command line to check the installation.

- Python* 3.6 or higher (64-bit).

Once installed, it should be available in your `PATH` environment variable. Execute `python --version` in the command line to check your system-wide Python installation.

- The `pip` and `ipython` packages for Python.

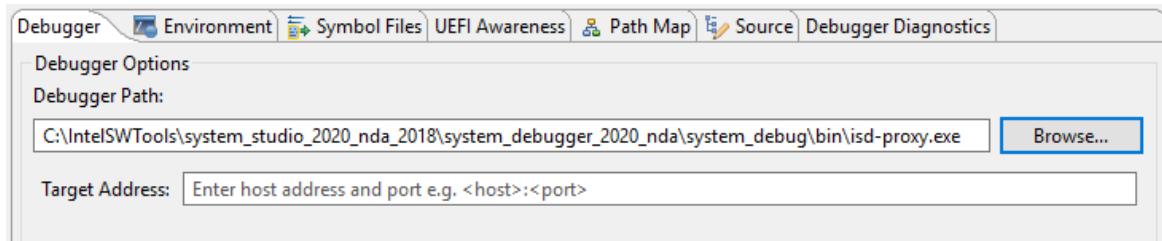
Connect to the Virtual Target

1. Launch the Simics(R) simulator and connect the debugger to it. Follow the [startup procedure](#).
2. (Optional) To check the path to the **isd-proxy**, do the following:

1. In the main Eclipse* toolbar, expand the drop-down menu next to the  Debug button and click **Debug Configurations**.

The latest System Debug configuration is opened.

2. The path is displayed in the **Debugger** pane.



3. Click **Debug**.

Debugging Capabilities

See the table below for available Intel(R) System Debugger features:

Feature	Supported in Post-Silicon (Hardware)
Target run control (halt and resume)	Yes
Target Reset	Yes
Source-level stepping	Yes
Instruction-level stepping	Yes
Breakpoints (on execution and memory access)	Yes
Platform breaks	Yes
Memory access	Yes
Access to general purpose registers	Yes
Access to model-specific registers (MSR)	Yes
IO Port access	Yes
CPUD	Yes
Target software awareness (loading PEI/EDL modules, using map files)	Yes
Instruction trace capturing and decoding	Yes
Last Branch Record (LBR)	Yes
Intel(R) Processor Trace (Intel(R) PT)	Yes

Feature	Supported in Post-Silicon (Hardware)
PCI Devices	Yes
Descriptor tables (GDT, LDT, IDT)	Yes
Logs	In the Shell view

Reverse Stepping

To debug the Simics(R) simulator, you can execute reverse stepping, which is only available for simulation (pre-silicon) targets. For default stepping instructions, see [Target Run Control](#).

1. [Connect to Simics\(R\) simulator](#).
2. Halt the target by pressing Suspend.
3. In the main toolbar, press Reverse Toggle to activate the reverse stepping mode.
4. If needed, press Instruction Stepping to enter instruction level.

Once the reverse stepping mode is activated, the following buttons are available:

- Reverse Step Into
- Reverse Step Over
- Uncall

For more information about debugging Simics(R) simulator, contact customer support (find details in Release Notes).

Loading Symbols Automatically

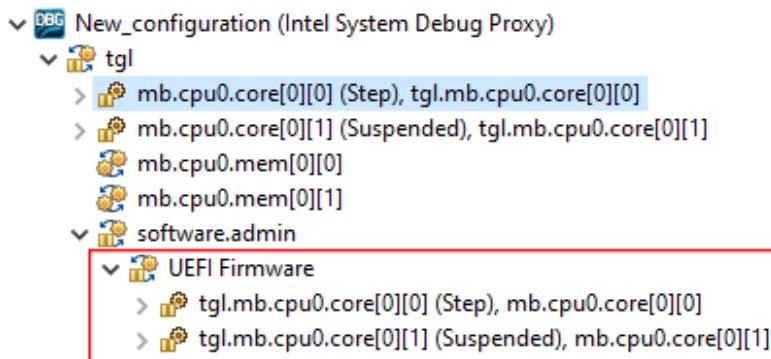
To load symbols automatically while debugging a Simics(R) simulation, you can use a UEFI tracker. Follow the steps below:

1. Enable the UEFI tracker. Add the following commands to the Simics simulation launch script or execute them one by one in the Simics simulation console (the example below is for the Tiger Lake simulation):

```
load-module uefi-fw-tracker
tgl.software.insert-tracker tracker = uefi_fw_tracker_comp
tgl.software.tracker.detect-parameters -overwrite
tgl.software.tracker.load-parameters uefi.params
tgl.software.enable-tracker
```

2. [Connect the debugger to the simulation](#).

3. When the connection is established, ensure that the UEFI Tracker is enabled successfully: check if the **software.admin** node in the Debug view contains the **UEFI Firmware** child:



4. Run and halt the target. Afterwards, the modules must be loaded automatically and displayed in the Modules view.

With UEFI tracker enabled, you do not need to use other manual methods for loading modules (for example, loading BIOS ROM file).

Collecting Execution Trace with Simics(R) Simulator

Like for the real hardware target, you can collect execution trace using the Last Branch Record (LBR) and Intel(R) Processor Trace (Intel(R) PT) mechanisms on the Simics(R) simulation.

Follow the steps below to launch trace capturing:

1. Configure the Simics simulation to enable the tracing mechanism:

- To use Intel PT, configure the Simics simulation BIOS to enable it.
- To use LBR, execute the following command in the Simics simulation console:

```
enable-lbtrs
```

2. Launch the simulation and create target connection. See [Connecting to Simics\(R\) Simulator](#).

3. In the [Instruction Trace view](#), click **Configure Instruction Trace** and select LBR or Intel PT from the **Trace Type** drop-down list.

See detailed instructions on [launching trace capturing](#).

Collecting Simulation Logs

To check logs for the running Simics(R) simulator, open the Console view, click the dropdown on the top-right and select **Simics(R) Simulator Log**.

The displayed table contains logs for the running simulation (IP and port are displayed on top).

The screenshot shows the Simics(R) Simulator Log window with the title "Simics(R) Simulator Log [TCP:127.0.0.1:12224]". The table has columns for Level, Object, Type, Cycle, and Message. The log entries are as follows:

Level	Object	Type	Cycle	Message
4	board.mb.sb.uhci[0]	info	260151462000...	reading 8 bytes from PCI memory space
4	board.mb.nb.remap_unit[0]	info	260151462000...	pass through transaction to 0xde811300, translation
4	board.mb.sb.uhci[0]	info	260151462000...	Reading 8 bytes from address 0xde811300
4	board.mb.sb.uhci[0]	info	260151462000...	reading 8 bytes from PCI memory space
4	board.mb.nb.remap_unit[0]	info	260151462000...	pass through transaction to 0xde811480, translation
4	board.mb.sb.uhci[0]	info	260151462000...	Reading 8 bytes from address 0xde811480
4	board.mb.sb.uhci[0]	info	260151462000...	reading 16 bytes from PCI memory space
4	board.mb.nb.remap_unit[0]	info	260151462000...	pass through transaction to 0xde810000, translation
4	board.mb.sb.uhci[0]	info	260151462000...	Reading 16 bytes from address 0xde810000
4	board.mb.sb.uhci[0]	info	260151462000...	handling transfer descriptor at 0xde810000, context .
4	board.mb.sb.uhci[0]	info	260151462000...	Signal already lowered for pin 0 (INT#A)

To extract logs in the CSV format, click Save.

You can increase the default level of logging displayed by executing the `log-level` command in the Simics simulator console. For example, the command below enabled level 4 logs displayed like in the screenshot above.

```
log-level 4
```

Debugging Trusted Domain Extention (TDX)

Intel(R) Trust Domain Extensions (Intel(R) TDX) is introducing architectural elements to help deploy hardware-isolated, virtual machines (VMs) called trust domains (TDs). Intel TDX is designed to isolate VMs from the virtual-machine manager (VMM)/hypervisor and any other non-TD software on the platform to protect TDs from a broad range of software.

For more information, see the [Intel\(R\) Trust Domain Extensions \(Intel\(R\) TDX\)](#).

Note

For debugging to work properly the system needs to be under debug (SUD). Please see [TDX debugging prerequisites](#) for more details.

Breaking on SEAM Entry

Intercepting the SEAM entry point to start debugging the SEAM root mode where the TDX module resides.

To intercept the SEAM entry point:

1. Halt the target by clicking the Suspend button in the main Eclipse* toolbar.
2. In the [Breakpoints view](#), set a platform break of type **VMExit_Seamcall** as described in [Setting Platform Breaks](#).
3. Resume the target by clicking the Resume button and wait till the platform break is hit.

Once the debugger has stopped at the VMExit_Seamcall break, it will be in SEAM Root mode. Updated registers are highlighted in the **Registers** view.

Breaking on TDCall

Intercepting the entry point into SEAM root mode when the **TDCall** instruction is called from the TD guest to start debugging the TDX module.

To intercept the TDCall:

1. Halt the target by clicking the Suspend button in the main Eclipse* toolbar.
2. In the [Breakpoints view](#), set a platform break of type **VMExit_TDCall** as described in [Setting Platform Breaks](#).
3. Resume the target by clicking the Resume button and wait till the platform break is hit.

Once the debugger has stopped at the VMExit_TDCall break, it will be in SEAM Root mode. Updated registers are highlighted in the **Registers** view.

Prerequisites for TDX debugging

Assuming the target has a valid TDX setup, there are some additional steps which needs to be done before the system is under debug (SUD).

To debug software in SEAM using probe mode, the debugger must declare that the platform is under debug prior to [MCHECK](#). This can be done through one of the following mechanisms:

- Red Unlocking the target
- Enable SGX debug mode in bios

Note

The system being in SUD or not does not prevent the TDX-module from enabling debug capabilities like debug breakpoints, single stepping, LBR, processor trace, etc. from being enabled in SEAM VMX-root or SEAM VMX-non-root operation. The architectural debug capabilities continue to function normally. The SUD only controls whether the TDX-module and the TDs can be debugged using probe mode.

The table below described the different debugging behavior based on operation mode and SUD state:

	not in SUD	In SUD
SEAM VMX-non-root	Causes TD Guest to exit.	Work as usual
SEAM VMX-root	Inhibited/Ignored	Work as usual
Legacy VMX	No change	No change

Debugging coreboot*

coreboot* is an open source firmware project, describing a phase-based initialization infrastructure for Intel(R) architecture and other processor architectures.

To start debugging coreboot* with Intel(R) System Debugger, you need to load symbol files manually. The default loading by pressing the  Load the Current Module button is not supported in this scenario.

1. Halt the target and right-click the desired  thread or  stack frame in the [Debug view](#).
2. In the popup menu, select **Symbol Files**.
3. In the opened dialog box, specify the path to  .debug files that contain Coreboot debug information.
4. Click **OK**
5. When all symbol files are added, close the dialog by clicking **OK**.

With necessary debug information loaded, you can start debugging coreboot*.

See also

- [coreboot* documentation](#)
- [Loading debug information manually](#)

Scripting Samples

This section contains scripting samples that you can use to perform debugging tasks that are beyond basic GUI options.

Creating Simple Commands for Frequent Debug Tasks

Create simple commands for performing frequent debug tasks in the CLI. You can define custom functions in any of the following ways:

- Define functions directly in the IPython console.

Use the [ISD Shell](#) view (it support proper indentation for defining functions), for example:

```
In [1]: def step_into():
...:     return threads[0].step_into(instruction=True)
...:

In [2]: step_into()
```

- Define functions in a separate file that will be imported automatically after the Sysdbg CLI is connected.

1. Create a new `.py` file and define all required functions there.

Do not forget to reference global variables properly. See the example file below.

2. Specify the path of the custom script in any of the following ways:

- If the **ISD Shell** view is used, set the environment variable

`ISYSDBG_CLI_CUSTOM_SCRIPT` before launching the Intel(R) System Debugger.

```
ISYSDBG_CLI_CUSTOM_SCRIPT = <custom_script_path>/<custom_script_name>.py
```

The environment variable can be declared in the same terminal before launching the Intel(R) System Debugger, in the system environment variables, or in the environment file:

- For Windows* OS: `<install_dir>/system_debugger/<version>/env.d/win32/*-isysdbg-env.bat`
- For Linux* OS: `<install_dir>/system_debugger/<version>/env.d/linux/*-isysdbg-env.sh`

- If the ISD CLI is used, the custom script can be specified in the connect command as follows:

```
sysdbg.connect(custom_script="<custom_script_dir>/<custom_script_name>.py")
```

- If the Sysdbg CLI entry point is used, the custom script can be specified as follows:

```
intel_sysdbg --custom_script="<custom_script_path>/<custom_script_name>.py"
```

3. For example, in the **ISD Shell** view, the custom module can be used in the following way:

```
[INFO      ] Custom module <custom_script_name> was imported.
In [1]: sysdbg.<custom_script_name>.step_into()
In [2]: bp=sysdbg.<custom_script_name>.set_bp(0x18000135c)
[INFO      ] Breakpoint added {ID: 0, Contexts: ['P1.100b'], Enabled: True, Location: 0x18000135c, Type: Software}.
In [3]: sysdbg.<custom_script_name>.remove_bp(bp)
[INFO      ] Breakpoint with ID 0 removed.
Out[4]: True
```

Example Commands

The snippet below contains example script with basic debug functions. You can copy the script into your custom `.py` file and use it as explained above.

```

_target = None
_threads = None

def on_launch(target):
    """
    Define in this function actions to be executed on Launch during
    the connection attempt. This function is automatically executed
    after the Sysdbg CLI is started. For example, global variables
    _target and _threads used in this module can be initialized.
    If defined, this function must have the definition above, so
    that it can be correctly executed: "def on_Launch(target)".

    :param target: target variable on global scope
    :type target: Session
    :returns: None
    """

    # If _target and _threads global variables are used in
    # other functions in this module, initialize them here.
    global _target
    global _threads
    _target = target
    _threads = target.threads
    # Add here other actions to be executed on Launch (optionally).

def read_ioport(port):
    """
    Read an IO port on the first thread of the target.
    :param port: address of the IO port (port number)
    :type port: int
    :returns: int value of the IO port
    """

    return _threads[0].io(1, port)

def write_ioport(port, value):
    """
    Write an IO port on the first thread of the target.
    :param port: address of the IO port (port number)
    :type port: int
    :param value: value to be written
    :type value: int
    :returns: None on successful operation
    """

    return _threads[0].io(1, port, value)

def go():
    """
    Resume the target.
    :returns: True on successful operation
    """

    return _target.go()

def halt():
    """
    Halt the target.
    :returns: True on successful operation
    """

```

```

    return _target.halt()

def step_into():
    """
    Perform a step into operation on the first thread of the target.
    :returns: None on successful operation
    """

    return _threads[0].step_into(instruction=True)

def set_bp(address):
    """
    Add a breakpoint on the first thread of the target.
    :param address: address of the breakpoint to add
    :type address: int
    :returns: the Breakpoint object added
    """

    return _threads[0].set_breakpoint(address=address)

def remove_bp(bp):
    """
    Remove a breakpoint on the first thread of the target.
    :param bp: Breakpoint object or breakpoint ID to remove
    :type bp: Breakpoint object or str
    :returns: True on successful operation
    """

    return _threads[0].remove_breakpoint(bp)

```

! See also

[Automating Tasks on Launch](#)

Executing Multiple Steps in a Row

The default instruction stepping mode available in the [Debug view](#) executes one step at a time (for more information, refer to [Target Run Control](#)).

To execute multiple steps in a row, launch the [ISD Shell](#) and run the following script:

```
threads[0].step(steps=N)
```

where `N` is the number of steps you want to execute.

! See also

[Create simple CLI commands for frequent debug tasks](#)

Configuring the Scan for Debug Information

The functionality of  Load the Current Module (formerly named **Load This**) is based on scanning the memory for PE/COFF module headers.

The default way of loading debug information by pressing  Load the Current Module button (see instructions [here](#)) implies the following conditions to be true:

- You want to load the debug information of the module being currently executed. This means that the start address for the scanning is the current program counter.
- Most of modules to be loaded are aligned at the page boundary. This means, the default stride equals the page size (4KB).
- Modules to be loaded are not bigger than 4KB.

To load information with custom parameters, launch the  ISD Shell and execute the following command:

```
threads[N].load_this([<start>, <alignment>, <search_range>])
```

where:

- N is the number of hardware thread.
- *start* (optional) points to the address to start scanning for debug information. The default value is the current program counter.
- *alignment* (optional) defines the scan stride size in bytes. The default value is 8 bytes when the *start* parameter represents an address in flash memory. Otherwise, the default value is the page size, 4KB.
- *search_range* (optional) defines the scan range size in bytes. The default value is 64KB when the *start* parameter represents an address in flash memory (flash modules). Otherwise, the default value is 8MB.

For instructions on fixing other issues, see the [Troubleshooting](#) section.

Examples

Search from the current program counter with a stride of 1KB and a range of 1MB

```
threads[0].load_this(alignment = 1024, search_range = 1 << 20)
```

Search from a given address using the range of 8MB

```
threads[0].load_this(start = 0x12340000, search_range = 8 << 20)
```

In BIOS PEI/pre-MRC phase, for PEI modules whose size is larger than the default `search_range` value, the search must scan larger memory spaces. For example, search from the current program counter with a stride of 4B and a range of 1MB

```
threads[0].load_this(search_range = 1 << 20, alignment = 4)
```

! See also

[Create simple CLI commands for frequent debug tasks](#)

Configuring the Scan for EFI Images

The [default way of loading EFI images](#) by pressing the  **Load EFI Modules** button launches the scan within a default physical memory address range based on the platform type. If this scan did not locate any modules, you can configure the search range manually.

Use the [ISD Shell](#) to call the `load_dxe()` function with additional parameters as described below.

- `system_table` - the base address (in hexadecimal format) of the EFI system table. Example:

```
threads[0].load_dxe(system_table=<address>)
```

If the specified base address is correct and the debugger is able to locate the table, the following message is returned:

```
INFO: LoadDXE: Loading N EFI images
```

- `top_of_memory` - the starting point (address in hexadecimal format) of the downwards search for the EFI system table pointer structure. The maximum search range is 4GB. Example:

```
threads[0].load_dxe(top_of_memory=<address>)
```

You should see the following message returned:

```
INFO: LoadDXE: Searching for the EFI system table pointer from <your-address> to 0x0
```

```
INFO: LoadDXE: Loading N EFI images
```

If no modules are found and loaded, check [potential failure causes](#).

! See also

[Create simple CLI commands for frequent debug tasks](#)

Load Debug Information for All Threads

To load debug information for all threads at once:

1. Create a loop to scan the memory for debug information for each hardware thread:

```
for t in threads:  
    t.load_this()
```

2. Create a loop to provide each thread with reference to a binary symbol file:

```
for t in threads:  
    t.load_symbol_file(filename=<filename>, <address>)
```

where:

- *filename* is the name of a file to load
- *address* (optional) is the location of the file in the target memory

! See also

- [Loading Individual Modules](#)
- [Create simple CLI commands for frequent debug tasks](#)

Reading and Writing IO Ports

You can access IO port values via the [ISD Shell](#).

To read the IO port, execute the following command:

```
threads[0].io(<offset>, <width>)
```

To write to the IO port, execute the following command:

```
threads[0].io(<offset>, <width>, <value>)
```

! See also

[Create simple CLI commands for frequent debug tasks](#)

Break at the Initialization of a Linux* Kernel Module

You can use the following script to automate breaking at the initialization of an out-of-tree (separately compiled) Linux* kernel module.

```
def linux_module_break(name):
    # get do_init_module function address
    addr = conthreads[0].evaluate('&do_init_module')
    # place a breakpoint on it
    bp = con.set_breakpoint(address=int(addr, 16))
    # Loop until the desired breakpoint is hit (should be just 1 iteration)
    while True:
        # run until the breakpoint gets hit on any of the threads
        hit = con.run_until(bp, 300)

        # ## insert the module on your Linux target now ###

        # get the thread on which the breakpoint got hit
        t = conthreads[bp.index(hit)]
        # check if the name of the module is the expected one and break
        if t.evaluate('mod->name') == ('\"%s\"' % name):
            break

        # get the address of the module's init function
        initaddr = t.evaluate('mod->init')
        # set a breakpoint on it
        initbp = t.set_breakpoint(address=int(initaddr, 16))
        # run until the breakpoint gets hit
        t.run_until(initbp, 300)

        ### now, the GUI can be used to load the symbols for the module
```

Create the history of commands

In order to create the history of executed commands, the following functions, part of the IPython* magics module, can be used in the [ISD Shell](#):

```
In [1]: threads[8].is_halted()
Out[1]: True

In [2]: hex(threads[8].get_pc())
Out[2]: '0x1800013f7'

In [3]: threads[8].step_into(instruction=True)

In [4]: %history -n -o -p
1: >>> threads[8].is_halted()
True
2: >>> hex(threads[8].get_pc())
'0x1800013f7'
3: >>> threads[8].step_into(instruction=True)
4: >>> %history -n -o -p
```

where the following arguments were used:

- **-n**

print line numbers for each input. This feature is only available if numbered prompts are in use.

- **-o**

print outputs for each input.

- **-p**

print classic >>> Python* prompts before each input.

```
In [1]: threads[8].is_halted()
Out[1]: True

In [2]: hex(threads[8].get_pc())
Out[2]: '0x1800013f7'

In [3]: threads[8].step_into(instruction=True)

In [4]: threads[8].set_breakpoint(address=0x180002218)
Out[4]: {ID: 0, Contexts: ['P1.1010'], Enabled: True, Location: 0x180002218, Type: Software}

In [5]: %history -n -o -p 2 4
2: >>> hex(threads[8].get_pc())
'0x1800013f7'
4: >>> threads[8].set_breakpoint(address=0x180002218)
{ID: 0, Contexts: ['P1.1010'], Enabled: True, Location: 0x180002218, Type: Software}

In [6]: %history -n -o -p 1-3
1: >>> threads[8].is_halted()
True
2: >>> hex(threads[8].get_pc())
'0x1800013f7'
3: >>> threads[8].step_into(instruction=True)
```

where specific line numbers were passed to be printed.

```
In [1]: %history -n -o -p -f "/path/to/file.txt"
```

where the argument -f followed by a filename indicates that the output will be redirected to the given file instead of printing it to the screen.

User Interface Reference

Describes units of Graphical User Interface.

Equivalent description of GUI elements is presented in the corresponding Eclipse* documentation. For more information, refer to [Eclipse documentation](#).

Drag and Drop

You can drag a view and drop it to another location within the workspace.

- In general, dragging and dropping only affects the display of views.
- You can undo every drag and drop action.
- You can cancel a drag and drop action by pressing Esc, holding it down, and releasing the mouse button. Alternatively, you can drop the view anywhere where dropping has no effect.
- For some views, you can copy the selection by pressing the **Ctrl** key while dragging the view.

To drag and drop a view:

1. Select a view in the workspace.
2. Click the left mouse button on the highlighted item and hold the mouse button while dragging the mouse pointer into another location.

While you are dragging a view, the workspace shows how its look will be configured if you drop the view where your mouse pointer is currently located.

3. Release the mouse button within the destination location.

The view appears in the destination location.

Views

Views are fundamental units of the Intel(R) System Debugger GUI. This section describes specifics of each System Debug view you can use.

Breakpoints View

Display information on all existing breakpoints.

Menu

Window > Show View > Breakpoints

Toolbar



The **Breakpoints** view displays a list of all breakpoints and platform breaks that are currently set. You can use the view to set, delete, modify, enable, or disable various types of breakpoints and platform breaks.

See the table below for icons and other GUI details for breakpoints:

Icon	Name	Description
	Line Breakpoint	Suspends the program execution before a particular line of
	Function Breakpoint	Suspends the program execution before a particular functio
	Software Address Breakpoint	Suspends the program execution before a particular assem
	Hardware Address Breakpoint	Suspends the program execution before a particular assem
	Platform Break (Event Breakpoint)	Intercepts system-level events like system reset or the Syste
	Watchpoint	Suspends the program execution whenever the value of a g

Note

In the current Intel(R) System Debugger configuration, you must explicitly specify the name of the event to be intercepted by the Platform Break. Only use names provided in the table above and pay attention to spelling and case.

For instructions on working with breakpoints, platform breaks, and watchpoints, see [Using Breakpoints](#).

View Tool Options

You can interact with the **Platform Register Dictionary** view using the following buttons:



Adds a register to be displayed in the [Platform Register Watch view](#).

Debug View

Menu

Window > Show View > Debug

Toolbar



The **Debug** view shows the target debugging information in a tree hierarchy. Each thread is represented as a tree node. When threads are suspended, the **Debug** view displays the stack frame for the threads for each target you are debugging.

See [Target Run Control](#) for instructions on controlling the target state.

The tree nodes are marked with the following icons:

Icon	Session Item	Description
	Launch instance	Launch configuration name and launch type
	Debugger instance	Debugger name
	Thread instance	Thread number and state
	Stack frame instance	Stack frame number, function, file name, and file line number

View Toolbar Options

You can interact with the **Debug** view using the following default buttons:

Remove All Terminated Launches

Removes all terminated process instances from the view.

Instruction Stepping Mode

Enables the Instruction Stepping mode examine a program as it steps into disassembled code.

View Menu

Open a list of the following options:

- **Layout:** provides layout options for the view.
- **Show Debug Toolbar:** displays additional toolbar with options for the target run control.

Debug Toolbar Options

To display additional run control options, select **Show Debug Toolbar** in the **Debug** view menu.

The following buttons are available for interaction with the **Debug** view specifically:

Drop To Frame

Re-enters the selected stack frame in the **Debug** view.

Use Step Filters

Enables using step filters in the **Debug** view.

Disassembly View

Menu

Window > Show View > Disassembly

Toolbar

Disassembly

The **Disassembly** view displays the program as a list of assembler instructions. You can configure the view to display the assembler instructions and the corresponding lines of source code together. The currently executed instruction is highlighted and indicated by a marker in the view.

By default, the **Disassembly** view displays instructions associated with the active debug context. If no active debug context exists, the view is empty.

View Toolbar Options

You can interact with the **Disassembly** view using the following buttons:



Refreshed the view and updates the list of assembler instructions.



Returns to the assembler instructions being executed at the current time.



Displays only instructions associated with the debug context selected in the **Debug** view.



Displays assembler instructions together with the corresponding lines of high-level source code.

Expressions View

Menu

Window > Show View > Expressions

Toolbar



The **Expressions** view allows you to view existing expressions and add new ones. For instructions on setting watchpoints associated with expressions, see [Setting Watchpoints](#).



Before establishing a target connection, ensure that the **Expressions** view does not contain any watched expressions. Otherwise, it might cause the target halt when the connection is created

View Toolbar Options

Show Type Names

Displays type names of expressions.

Show Logical Structure

Displays the logical structure of expressions.

Collapse All

Collapses all currently expanded expression instances in the view.

Create a New Watch Expression

Opens the **Add Watch Expression** dialog that allows you to create a watch expression and add it to the **Expressions** view

Remove Selected Expressions

Removes selected expressions from the view.

Remove All Expressions

Removes all expressions from the view.

Open New View

Opens a new **Expressions** view.

Pin to Debug Context

Pins the selected expression to an active debug context displayed in the **Debug** view.

View Menu

Provides the following configuration options:

- **Layout:** sets an alternative layout for the view.

Global Descriptor Table View

Menu

Window > Show View > Global Descriptor Table

Toolbar

Global Descriptor Table

The view allows you to explore the Global Descriptor Table (GDT). To successfully use the view, make sure the target is halted.

To configure the display of the GDT, fill the following fields:

- **Descriptor Table Address** - the address at which the desired descriptor table is located.
- **Limit**

To update the view according to the defined parameters, click  Refresh.

To reset the view to the initial display settings, click **Reset**.

Interrupt Descriptor Table View

Menu

Window > Show View > Interrupt Descriptor Table

Toolbar



The view allows you to explore the Interrupt Descriptor Table (IDT). To successfully use the view, make sure the target is halted.

To configure the display of the IDT, define the **Descriptor Table Address**: the address at which the desired descriptor table is located.

Working with IDT

Inside the IDT view, you can link addresses displayed in the table with the Disassembly view, set breakpoints, or explore register structure.

To link an IDT entry with a corresponding address in disassembly view, do any of the following:

- Right-click the table row and select  **View Disassembly**.
- Double-click the cell under Index, Linear Address, or Description column.

The [Disassembly view](#) is refreshed and the address you have chosen is displayed on top.

To add set a breakpoint to a particular address, right-click the table row and select  **Set breakpoint**. A new address breakpoint appears in the [Breakpoints view](#) (enabled by default).

If you double-click the cell under Attributes or Type column, the [Platform Register Editor](#) opens.

Instruction Trace View

Displays the history of instructions executed since tracing is enabled.

Menu

Toolbar

Instruction Trace View

The **Instruction Trace View** shows the chronological order of instruction calls captured by the Last Branch Record (LBR).

The view contains five columns:

- **Address:** the assembler instruction address.

Instructions are grouped by the line number. Expandable **Address** items display multiple instructions called from a single code line.

- **Instruction:** the instruction syntax.

Note

If the thread selected in the **Debug** view contains no symbols loaded, the following columns are empty.

- **Module:** the name of the running module.
- **File Name:** the name of the source code file.
- **Line Number:** the number of the source file line that calls the instruction.

To view the line in the source file, double-click the line number. The source file opens in a new view and the selected line is highlighted.

- **File Path:** path to the source file.

View Toolbar Options

Start Instruction Trace

Starts or resumes the terminated instruction trace.

Stop Instruction Trace

Stops the running instruction trace.

Configure Instruction Trace

Opens a dialog box that allows you to configure the trace type. Currently, only Last Branch Record (LBR) is supported.

Refresh View

Refreshes the view and updates the instruction history.

Pin to Hardware Thread

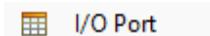
Pins the view to an active hardware thread displayed in the **Debug** view. If pinned, the view shows the instruction trace history only for the chosen thread (the name is displayed above the table) and is not refreshed if another thread is selected.

I/O Port View

Menu

Window > Show View > I/O Port

Toolbar



The **I/O Port** view accesses a given number of I/O ports starting from the specified address and uses a customizable access width.

The screenshot shows the I/O Port view in a debugger. The top bar includes tabs for Variables, Breakpoints, Expressions, Modules, I/O Port, and a close button. Below the tabs, there are icons for copy, paste, and refresh, followed by the Address field (0x20), Size field (288), and a large table area.

Address	Value	Text
0x0020	02 FF FF FF 02 FF FF FF 02 FF FF FF 02 FF AA FF	.ÿÿ.ÿÿ.ÿÿ.ÿÿ
0x0030	02 FF FF FF 02 FF FF FF 02 FF FF FF 02 FF FF FF	.ÿÿ.ÿÿ.ÿÿ.ÿÿ
0x0040	00 FF 00 FF	.ÿ.ÿÿÿÿÿÿÿÿÿÿÿÿ
0x0050	00 FF 00 FF	.ÿ.ÿÿÿÿÿÿÿÿÿÿÿÿ
0x0060	FA 20 00 FF 04 FF 00 FF FF FF FF FF FF FF FF FF	ú .ÿ.ÿ.ÿÿÿÿÿÿÿÿ
0x0070	00 00 12 15 4A 00 12 15 FF FF FF FF FF FF FFJ...ÿÿÿÿÿÿ
0x0080	00 00 FF	..ÿÿÿÿÿÿÿÿÿÿÿÿ
0x0090	FF FF 00 FF	ÿÿ.ÿÿÿÿÿÿÿÿÿÿÿÿ
0x00A0	00 FF FF FF 00 FF FF 00 FF FF 00 FF FF FF FF	.ÿÿ.ÿÿ.ÿÿ.ÿÿ
0x00B0	00 FF FF 00 00 FF FF 00 FF FF 00 FF FF FF FF	.ÿÿ..ÿÿ.ÿÿ.ÿÿ
0x00C0	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x00D0	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x00E0	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x00F0	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0100	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0110	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0120	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x0130	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

The view contains the following columns:

Address

These addresses correspond to the first port of each row. To check the address of a port, hover the value with the mouse cursor. The address will appear in the tooltip.

Numerical values

The table entry covers the I/O port requested data and offers write access to individual I/O ports by editing the displayed values.

Text

This column is the text representation of the numerical values.

If you right-click any entry in the table, the following options pop up:

- **Refresh** - reload, from the target, the specified range of ports
- **Cell Size** - set width of the I/O port access and refresh the table
- **Columns** - set the number of columns in the view without refreshing the table
- **Radix** - set the radix of the numerical representation
- **Endian** - set the endianness
- **Code Page** - set the text coding
- **Show/Hide Text** - enable/disable the display of the text representation
- **Export** - export the I/O port data to a .csv file

Local Descriptor Table View

Menu

Window > Show View > Local Descriptor Table

Toolbar



The view allows you to explore the Local Descriptor Table (LDT). To successfully use the view, make sure the target is halted.

To configure the display of the LDT, fill the following fields:

- **Descriptor Table Address** - the address at which the desired descriptor table is located.
- **Limit**

To update the view according to the defined parameters, click Refresh.

To reset the view to the initial display settings, click **Reset**.

Memory Browser View

Menu

Window > Show View > Memory Browser

Toolbar



The **Memory Browser** view allows you to monitor and modify process memory. For detailed instructions on interacting with memory, see [Working with Memory](#).

Warning

Before establishing a target connection, ensure that the **Memory Browser** view does not contain any memory renderings. Otherwise, it might cause the target halt when the connection is created

In the default (integer) rendering, the **Memory Browser** view contains three panes by default: the memory address column, the hexadecimal rendering pane, and the text-like rendering pane.

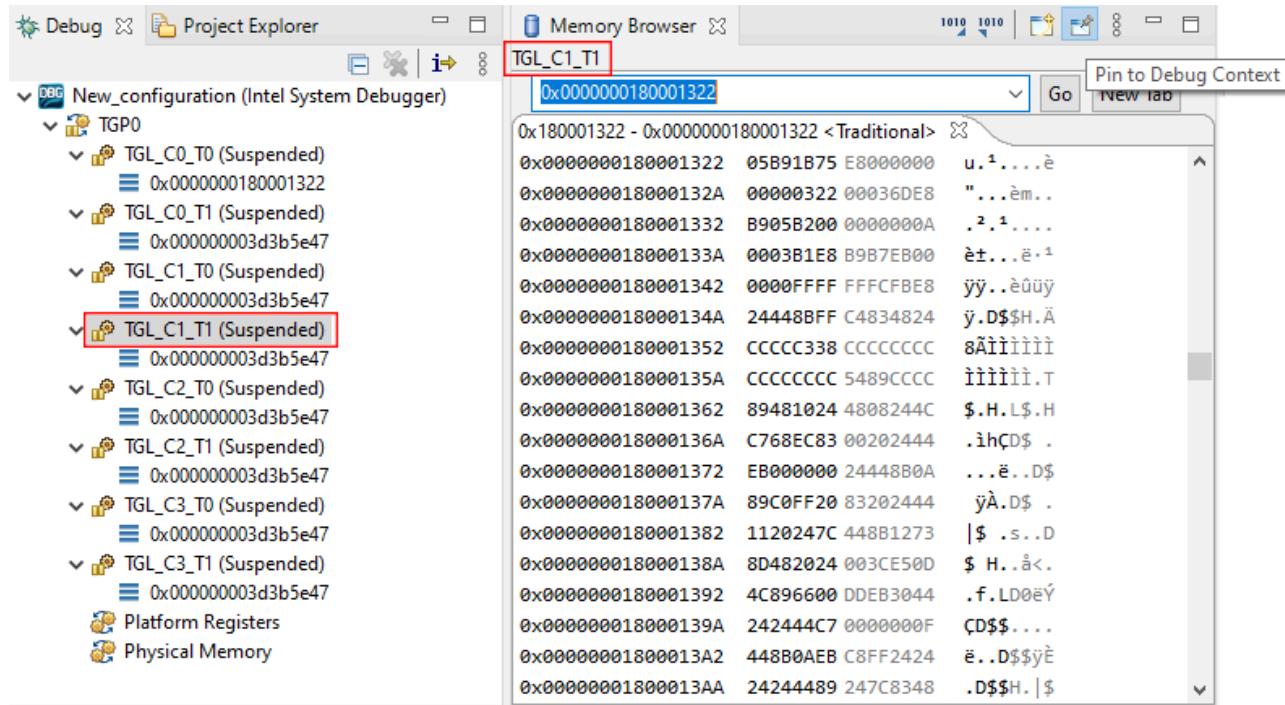
To configure the panes, right-click any place within the rendering tab. You can use the following basic options:

- Panes: select panes to be displayed in the tab.
- Endian: choose the endian for the displayed numerical values.
- Text: choose the format of the text rendering.
- Cell size: choose the number of bytes to be displayed in each cell.
- Radix: choose the memory rendering format.
- Columns: select the number of columns displayed or use the automatic size fit.

Pinning to Debug Context

You can pin the view output to the desired debug context. This can help you, for example, to track physical memory while doing run control on a thread. Follow the steps below:

1. Select the desired thread in the **Debug** view.
2. Go to the **Memory Browser** view and click the **Pin to Debug Context** button.
3. The name of the selected debug context is displayed on top of the **Memory Browser** view:



Once the view is pinned to a particular debug context, clicking on a different thread in the **Debug** view will not refresh the contents of the **Memory Browser** view. See an example below:

The screenshot shows the Intel System Debugger interface. The Project Explorer window on the left lists a project named "New_configuration" under "Intel System Debugger". Inside, there's a tree view of memory regions: TGPO, TGL_C0_T0, TGL_C0_T1, TGL_C1_T0, TGL_C1_T1, TGL_C2_T0 (which is selected and highlighted with a red box), TGL_C2_T1, TGL_C3_T0, TGL_C3_T1, Platform Registers, and Physical Memory. The Memory Browser window on the right displays memory starting at address 0x18000000180001322. A specific row, 0x180000001800013AA, is also highlighted with a red box. The status bar at the bottom indicates memory usage: 1010 1010.

To unpin the view, click the Pin to Debug Context button again to disable it.

Change Memory

You can directly edit memory in the **Memory Browser** using the Traditional Rendering (default) by:

1. Select the memory (cell) which you want to edit
2. Use the keyboard to change the memory value
3. Hit the “Enter” key to apply the value.

Please note that changing the focus to another cell or view will not apply the value.

4. Using the default color theme, the temporary changes are marked in yellow and the applied changes are marked in red.

If you use other rendering than the default Traditional Rendering, the behavior might be different.

View Toolbar Options

Import

Opens the **Import Memory** dialog box allowing you to import memory from a file. For instructions on importing memory, see [Working with Memory](#)

Export

Opens the **Export Memory** dialog box allowing you to export memory to a file. For instructions on exporting memory, see [Working with Memory](#)

Open New View

Opens new **Memory Browser** view.

Pin to Debug Context

Pins the memory to an active debug context selected in the **Debug** view.

View Menu

Opens a list of the following options:

- **Default Rendering:** allows you to choose between the traditional (integer) and floating-point rendering. If the latter rendering is set, the view contains the memory address column and the floating-point rendering pane.
- **Find/Replace:** open the **Find/Replace Memory** dialog box allowing you to modify the memory.
- **Find Next:** allows you to browse the memory rendering fast.
- **Clear Expressions:** cleans the history of the search field.

Modules View

Menu

Window > Show View > Modules

Toolbar

Modules

The **Modules** view allows you to view information about the modules loaded in the current debug session, including executables and shared libraries. The view contains two areas:

- Module tree - displays all modules involved in the current session. To view module internals like functions, global variables and others, expand the module instance.
- Detail pane - displays information on the module selected in the module tree: file name, symbol file name, address, size, flags, and other details.

Important

Module information found via automatic loading methods (in the table below, all columns except the last one) is read-only and must not be edited in the GUI

Example of the view with several modules loaded:

The screenshot shows the Eclipse IDE's Modules view. The top navigation bar includes tabs for Variables, Breakpoints, Expressions, and Modules. The Modules tab is active. Below the tabs is a toolbar with icons for search, filter, and other functions. A table lists the loaded modules:

Name	File Name	Address	Size
> FspSecCoreT.efi	C:/Users/.../D...	0xffebfffb0	0x00009df0
> FspSecCoreS.efi	C:/Users/.../D...	0xffffcfffb8	0x00000440
> CpuIoPei.efi	C:/users/.../do...	0xfffff19740	0x00001920
> PeiCore.efi	C:/Users/.../D...	0xffffd84d1c	0x0000d060
> PcdPeim.efi	C:/users/.../do...	0xfffff11080	0x00003900
> AmtStatusCodePei.efi	C:/users/.../do...	0xffff868c0	0x00000d20
> Dxelpl.efi	C:/Users/.../D...	0xffffd015a0	0x00003ca0
> Tcg2Pei.efi	C:/users/.../do...	0xfffe980	0x00005f40

Below the table, module details are displayed:

File name: C:/Users/.../Documents/TGL_BIOS/tgl/Build/TigerLakeFspPkg/DEBUG_VS2017/IA32/IntelFsp2F
Symbol file: C:\Users\...\Documents\TGL_BIOS\tgl\Build\TigerLakeFspPkg\DEBUG_VS2017\IA32\IntelFsp
Address: 0xffebfffb0
Size: 0x00009df0
Flags: ---

where:

- *File name* is the path of a PE/COFF or ELF file on the disk.

You can expand each module to see the source file name and the path to it.

The screenshot shows the Eclipse IDE's Modules view with the FspSecCoreT.efi module expanded. The expanded view shows the following components:

Name	File Name	Address	Size
> FspSecCoreT.efi	C:/Users/.../D...	0xffebfffb0	0x00009df0
SecCpuLib.iii	C:\Users\...\D...		
SecHostBridgeLib.iii	C:\Users\...\D...		
Flat32.iii	C:\Users\...\D...		
SecGetFspApiParameter.iii	C:\Users\...\D...		
FspApiEntryT.iii	C:\Users\...\D...		
FspHelper.iii	C:\Users\...\D...		

Below the table, module details are displayed:

File name: C:/Users/.../Documents/TGL_BIOS/tgl/Build/TigerLakeFspPkg/DEBUG_VS2017/IA32/IntelFsp2F
Symbol file: C:\Users\...\Documents\TGL_BIOS\tgl\Build\TigerLakeFspPkg\DEBUG_VS2017\IA32\IntelFsp
Address: 0xffebfffb0
Size: 0x00009df0
Flags: ---

- *Address* is the load address of a binary module.
- *Size* is the amount of memory occupied by the module.
- *Offset* is offset of a file “section” in the actual file on the disk.

You can open a source file in the Eclipse* editor or copy the file location to a clipboard. Right-click the source file instance and select the corresponding action.

For more information on loading modules, see [Loading Debug Information](#).

View Toolbar Options

You can interact with the **Modules** view using the following buttons:

Collapse All

Collapses all modules in the module tree.

Export Modules to File

Exports information on all modules to a text file in the following format: `<address> <size>
<path>`

Remove All Modules

Clears the view and removes all modules.

Load Symbols for All Modules

Loads the symbols for all modules involved in the current session. This option does not affect the modules with already loaded symbols.

Refresh

Refreshes the view and updates the module tree.

View Menu

Opens a list of the following options:

- **Layout:** provides alternative layout options for the view.

Model-specific registers View

Menu

Window > Show View > Model-Specific Registers

Toolbar

 Model-Specific Registers

The **Model-Specific Registers** view accesses the values of 'Count' number of model-specific registers (MSRs) beginning with 'Start Index'.

Console Model-Specific Registers X

Start Index: 0x480 Count: 30

0x00000480	00DA040000000004Ù.
0x00000481	0000007F00000016
0x00000482	FFF9FFE0401E172	rá..þýùý
0x00000483	01FFFFFF00036DFF	ÿm..ÿÿ..
0x00000484	0003FFFF000011FF	ÿ...ÿÿ..
0x00000485	000000007004C1E7	çÁ.p....
0x00000486	0000000080000021	!.....
0x00000487	00000000FFFFFFFFFF	ÿÿÿ.....
0x00000488	0000000000002000
0x00000489	0000000003727FF	ÿ'7.....
0x0000048A	000000000000002E
0x0000048B	005FBCFF00000000ÿ*..
0x0000048C	00000F0106734141	AAs.....
0x0000048D	0000007F00000016
0x0000048E	FFF9FFE04006172	ra..þýùý
0x0000048F	01FFFFFF00036DFB	ûm..ÿÿ..
0x00000490	0003FFFF000011FB	û...ÿÿ..
0x00000491	0000000000000001
0x00000492	?????????????????	?????????
0x00000493	?????????????????	?????????

The view contains the three columns:

Index

A hexadecimal representation of the MSR index.

Numerical value

The table entry covers the requested MSR data and offers write access to individual registers by editing the displayed values.

Text

A text representation of the numerical values.

If you right-click any entry in the table, the following options pop up:

- **Refresh** - reload, from the target, the specified count of MSRs
- **Size** - set the word size and refresh the table
- **Radix** - set the radix of the numerical representation
- **Endian** - set the endianness
- **Code Page** - set the text coding
- **Show/Hide Text** - enable/disable the display of the text representation
- **Export** - export the MSR data to a .csv file

Note

There might be invalid MSR indices within the requested MSR list marked as question marks (????????????????).

Page Tables View

Menu

Window > Show View > Page Tables

Toolbar



The **Page Tables** view displays the tables that are used on the system for paging. Use it to analyze address translation set up by the software running on the target.

This includes the paging modes IA-32e with four or five levels of paging. Depending on the paging mode, the view displays one of the following types of tables: page table (PT), page directory (PD), page-directory-pointer table (PDPT), page map level 4 (PML4), and page map level 5 (PML5) if supported by Intel(R) architecture. The top-most table is shown when the Page Tables view opens. If a table entry points to a sub-table, you can view the sub-table by clicking on the expand icon.

A screenshot of the Page Tables view in a debugger's navigation bar. The bar includes tabs for Console, Registers, Problems, Executables, Debugger Console, Page Tables (which is selected and highlighted in blue), and Memory. Below the bar is a table with the following columns: Name, Virtual Memory Range, Physical Base Address, and Description. The table lists entries for PML4[0] through PML4[7], each with its corresponding PD entries (PD[0] through PD[7]). The table shows memory ranges from 0x0000000000000000 to 0x000000000000FFFF and physical base addresses ranging from 0x000000003DA02000 to 0x000000003DA03000. All entries have PWT=0, PCD=0, and R/W=1.

Name	Virtual Memory Range	Physical Base Address	Description
PML4[0]	0x0000000000000000 to 0x0000007FFFFFFF	0x000000003DA02000	PWT=0 PCD=0 R/W=1 ...
PD[0]	0x0000000000000000 to 0x000000003FFFFFFF	0x000000003DA03000	PWT=0 PCD=0 R/W=1 ...
PD[0]	0x0000000000000000 to 0x0000000001FFFFFF	0x0000000000000000	PWT=0 PCD=0 R/W=1 ...
PD[1]	0x000000000000200000 to 0x0000000003FFFFFF	0x000000000000200000	PWT=0 PCD=0 R/W=1 ...
PD[2]	0x0000000000400000 to 0x0000000005FFFFFF	0x0000000000400000	PWT=0 PCD=0 R/W=1 ...
PD[3]	0x0000000000600000 to 0x0000000007FFFFFF	0x0000000000600000	PWT=0 PCD=0 R/W=1 ...
PD[4]	0x0000000000800000 to 0x0000000009FFFFFF	0x0000000000800000	PWT=0 PCD=0 R/W=1 ...
PD[5]	0x000000000A000000 to 0x000000000BFFFFFF	0x000000000A000000	PWT=0 PCD=0 R/W=1 ...
PD[6]	0x000000000C000000 to 0x000000000DFFFFFF	0x000000000C000000	PWT=0 PCD=0 R/W=1 ...
PD[7]	0x000000000E000000 to 0x000000000FFFFFFF	0x000000000E000000	PWT=0 PCD=0 R/W=1 ...

The view contains the following columns:

Name

The name is the table type and index of the table entry: PT refers to page table, PD to page directory, PML4 to page map level 4, PML5 to page map level 5 (if supported), and PDPT to page-directory-pointer table. The first index is 0; the number of entries depends on the table size.

Virtual Memory Range

The memory range is covered in this table entry.

Physical Base Address

These addresses point to the next table or the physical location of the page.

Description

Depending on the table type, each table entry has various attributes.

If you right-click any entry in the table, the following options pop up:

- **Collapse All** - collapses all expanded table entries
- **Copy** - copies the selected entry to clipboard
- **Edit** - opens the Bitfield Editor window allowing you to modify a page table entry. Click **Set** to save changes.

HLAT

Hypervisor-managed linear-address translation (HLAT) is a feature that changes the way in which linear addresses are translated in Virtual Machine Extensions (VMX) non-root operation. HLAT views are enabled in the Page Table when certain conditions are met: when the CPU is in VMX non-root operation and “enable HLAT” VM-execution control is 1.

The image below is the view of two roots, HLATP and CR3. Each root contains either 5-level or 4-level translation tables. 

The image below is an expanded view of HLATP showing the 4-level translation table.



PCI Devices View

Menu

Window > Show View > PCI Devices

Toolbar



To show a list of PCI devices available during a debug session, halt the target and open the **PCI Devices** view, which displays all devices found.

Note

When the target is running, the view is empty. The list of PCI devices gets updated every time you halt the target.

To find a particular PCI device, start typing the device name in the search bar. The suggested items are displayed and the best match is highlighted blue. Alternatively, you can use the drop-down menu.

When a particular PCI device is selected, the view displays all registers belonging to the device. For certain registers, you can find meaningful descriptions in the [Registers](#) view.

PCI Bar Linking

In the PCI device browser, the Base Address Register (BAR) addresses are linked to the memory view widget to easily display these address spaces. Right-click on the desired BAR address, and then select **View Memory** as shown below:

(x) Variables Expressions Breakpoints Modules PCI Devices X

00:1f.03 Meteor Lake-P HD Audio Controller

Name	Hex	Decimal
✓ Type 0 Configuration Space Header		
Vendor ID	8086	32902
Device ID	7e28	32296
Command	0400	1024
Status	0010	16
Revision ID	00	0
Programming Interface	00	0
Subclass	03	3
Class code	04	4
Cache Line Size	00	0
Latency Timer	00	0
Header Type	00	0
BIST	00	0
BAR0	b7efc004	3085942788
BAR1	View Memory	
BAR2	bffd000	3221090304
BAR3	00000000	0
BAR4	b7f00004	3085959172
BAR5	000003ff	1023
Cardbus CIS Pointer	00000000	0
Subsystem Vendor ID	8086	32902
Subsystem ID	7270	29296
Expansion ROM base address	00000000	0
Capabilities Pointer	50	80
Interrupt Line	10	16
Interrupt PIN	01	1
Min Grant	00	0
Max latency	00	0
> Power management capability		
✓ PCI Express Capability		
PCI Express Cap ID	10	16
Next Cap Pointer	00	0
> PCI Express Capabilities Register	0091	145
> Device Capabilities	10000000	268435456
> Device Control	2000	8192
> Device Status	0010	16
> Link Capabilities	00000000	0
> Link Control	6009	24585
> Link Status	f014	61460
> Slot Capabilities	01400010	20971536
> Slot Control	0000	0
> Slot Status	0000	0
> Root Control	04a1	1185
> Root Capabilities	0001	1

Note

The 4 lower bits of the BAR addresses have a special meaning/encoding and should be ignored.

View Toolbar Options

You can interact with the PCI Devices view using the following default buttons:



Forces a new scan for PCI devices available. Click a chevron next to the button to specify the type of scan:

- **Quick scan** (default) - searches through buses as a tree structure to extract information about PCI devices. Not recommended for complex systems with multiple CPUs and start buses (use the full scan instead).
- **Full scan** - iteratively searches for all buses. Takes more time than the quick scan but enables successful scanning through complex systems.

Platform Register Dictionary View

Menu

Window > Show View > Platform Register Dictionary

Toolbar



The **Platform Register Dictionary** view allows you to explore all registers associated with the debug context selected in the **Debug** view.

To find a specific register, repeatedly expand the tree. Alternatively, you can type the register name into the search field on the top-right corner of the view. In this case, the place of the register in the tree is reflected in the **Register Hierarchy** column.

To copy information about a register, right-click the table row and select **copy**. All cell values are available in the clipboard.

View Tool Options

You can interact with the **Platform Register Dictionary** view using the following buttons:



Adds a register to be displayed in the [Platform Register Watch view](#).

Platform Register Editor View

Menu

Window > Show View > Platform Register Editor View

Toolbar

Platform Register Editor View

The **Platform Register Editor** provides a graphical representation of the register selected in one of the related views.

Platform Register Watch View

Menu

Window > Show View > Platform Register Watch

Toolbar

 Platform Register Watch

The **Platform Register Watch** view basically allows you to monitor the values of specific registers during runtime.

For each register added to the view, the following information is displayed:

- Name
- Visualized place in the register hierarchy
- Address in hexadecimal and decimal formats
- Description
- ID of the debug context associated with the register.

You can copy information about a register as follows:

- To copy all information about a particular register (or a number of registers), right-click the table row (or select and right-click several rows) and select **Copy Registers**. All cell values are available in the clipboard.
- To copy only one cell value, right-click the cell and select **Copy**.

Alternatively, press Ctrl+C when the cursor is hovered over the required cell.

View Toolbar Options

You can interact with the **Platform Register Watch** view using the following buttons:

Show Register Dict

Opens a dialog box, which duplicates the contents and functionality of the [Platform Register Dictionary view](#). This option allows you to save working space by interacting with the dialog box instead of the associated view.

Remove From Watch

Removes a selected register from the view.

Remove All From Watch

Removes all registers from the view.

Registers View

Menu

Window > Show View > Registers

Toolbar

 Registers

The **Registers** view allows you to explore and edit the contents of the target registers. The view contains two areas:

- Register tree - displays the hierarchical structure of available registers and details on each instance. If a register contains flags, they are also displayed as nodes of the register.
- Details pane - displays information on a selected register:
 - Full name of the register.
 - Register value in the following number systems: hexadecimal, decimal, binary, and octal.
 - Register size and permissions.

To copy information about a register, right-click the required table cell and select **Copy**.

Alternatively, press **Ctrl+C** when the cursor is hovered over the required cell. The cell value is available in the clipboard.

View Toolbar Options

Show Type Names

Displays type names of registers.

Show Logical Structure

Displays the logical structure of registers.

Collapse All

Collapses all currently expanded register instances in the view.

Refresh

Refreshes the view and updates the register tree.

Open New View

Opens a new **Registers** view.

Pin to Debug Context

Pins the selected register to an active debug context displayed in the **Debug** view.

View Menu

Provides the following configuration options:

- **Layout:** sets an alternative layout for the view.

ISD Shell View

Menu

Window > Show View > Other > ISD Shell

Toolbar

ISD Shell

This shell is used across Intel(R) System Debugger components (including Target Connection Assistant). Refer to the shell to check log messages and implement scripting solutions described at [Scripting Samples](#).

You can also use the shell to [import and execute custom Python*](#) scripts.

Variables View

Menu

Window > Show View > Variables

Toolbar

(x)= Variables

The **Variables** view displays an expandable tree structure of local variables associated with the stack frame selected in the **Debug** view. Alternatively, you can display the value of a particular local variable by hovering over its instance in code. You can see details on a selected variable on the bottom of the **Variables** view.

Note

If the debugger cannot display the value of a particular variable and sets it to **N/A**, refer to the corresponding case described in the [Troubleshooting](#) section.

During a debug session, you can change values of variables to check how they are handled or to go through a loop faster. To do it, right-click a variable, select **Change Value**, and type in a new value.

To disable a variable and prevent the debugger from reading it, right-click a variable and select **Disable**. It might be useful if a target is very sensitive.

View Toolbar Options

You can interact with the **Variables** view using the following buttons:

Show Logical Structure

Displays the logical structure of variables.

Collapse All

Collapses all currently expanded variable instances in the view.

Open New View

Opens a new **Variables** view.

Pin to Debug Context

Pins the variable to the associated debug context.

View Menu

Provides the following configuration options:

- **Layout:** sets an alternative layout for the view.

System Trace

The System Trace component of Intel(R) System Debugger provides the ability to capture, decode, and display traces from hardware, firmware, and software sources via Intel(R) Trace Hub. System Trace provides various features to analyze these traces, including the ability to filter and search the traces, display their event distribution, and view their state transitions over time.

This chapter covers the standard flow of working with System Trace as well as an overview of basic and advanced analysis features.

Before You Begin

1. If you are a new user, review the primary [tracing use cases](#).
2. Check [supported platforms and probes](#) for the present release.
3. Complete the [startup procedure](#): configure and connect the target.

If you plan to use a Simics(R) simulation, follow [additional setup instructions](#) after you complete the primary setup.

4. Finish [setting up System Trace](#).

If you plan to work with any targets from the list below, see [additional setup instructions](#):

- Ice Lake
- Skylake PCH-H and Skylake PCH-LP
- Broxton P
- Gordon Peak Modular Reference Board (GP MRB)

If you want to use Lauterbach* connections, refer to the [installation steps](#).

! See also

[Troubleshooting](#)

Setup System Trace

The basic scenarios of using the System Trace feature of Intel(R) System Debugger are the following:

- Capture trace for the target connected and analyze it

In this scenario, you need to connect the debugger to your physical or virtual target, set up a new Trace Project, capture traces, and use different visualization features in System Trace for analysis as described in the [corresponding chapter](#).

To follow this scenario:

1. Start [here](#) and follow instructions for physical or virtual targets.
 2. Proceed with additional instructions in the child pages of this chapter.
- Import previously captured traces and analyze it

In this scenario, you need to specify the target associated with the imported Trace Capture and select the *offline* connection mode. System Trace will use the selected target information to properly decode the capture file so that you can successfully use different visualization features for analysis as described in the [corresponding chapter](#).

To follow this scenario:

1. Start [here](#) and follow instructions for your case (use the manual connection mode).
2. Proceed with additional instructions in the child pages of this chapter.

Capture Trace

Consider the following prerequisites before starting trace capture:

- Connection to the Target was successful.

In case of issues, refer to the Target Connection Assistant section.

- A **Trace Configuration file** for the active target connection settings has been created. For instructions, see the setup section.
- Target system has been **configured** for tracing.

For target-specific details, see [Appendix A: Hardware Setup](#) and [Appendix B: Tracing with Simics\(R\) Simulation](#).

! Note

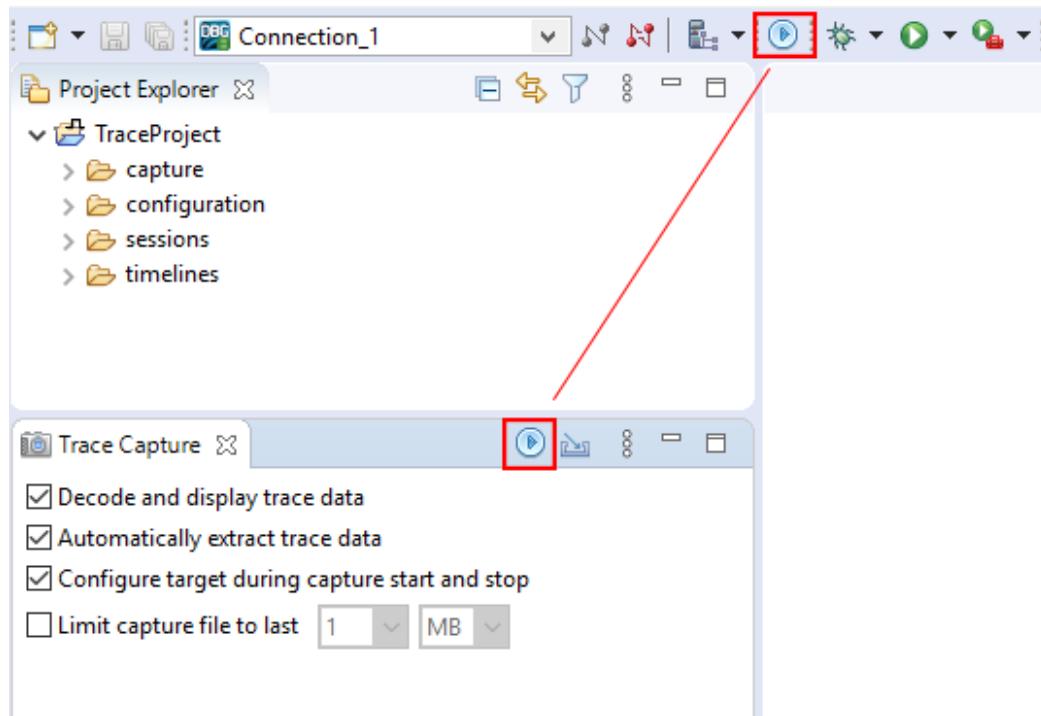
The state **Connected** means that the System Trace is connected to the selected target system. Connected does not mean that the Intel(R) Trace Hub is already configured. The configuration of the Intel(R) Trace Hub is done when the capturing starts.

The Target Connection dialog also shows the current Target Status and Core Status:

Connected: Connected
Connection: Intel(R) DCI OOB
Target Status: Ok
Cores Status: Running

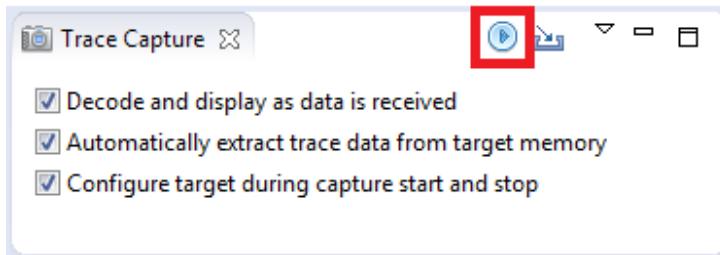
Start Trace Capture

When your target is successfully connected, **Start Trace** buttons are enabled and your workspace should look as follows:



Launch trace capture as follows:

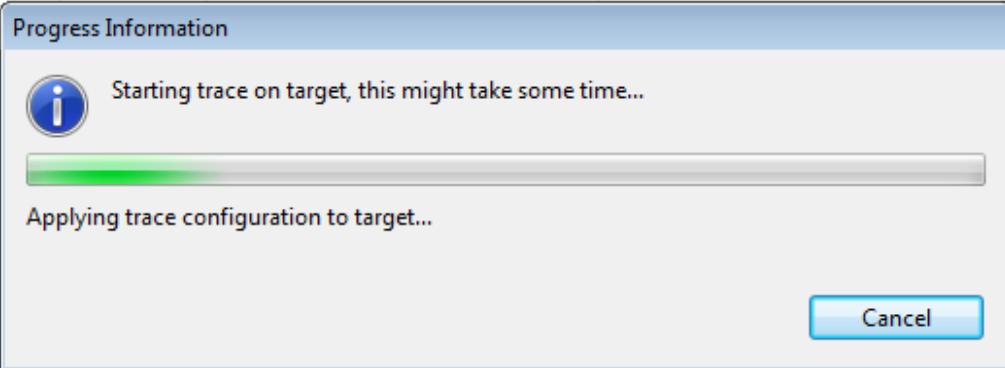
1. Click on the start trace button in Trace Capture view or in Eclipse toolbar to begin **capturing trace data from target**.



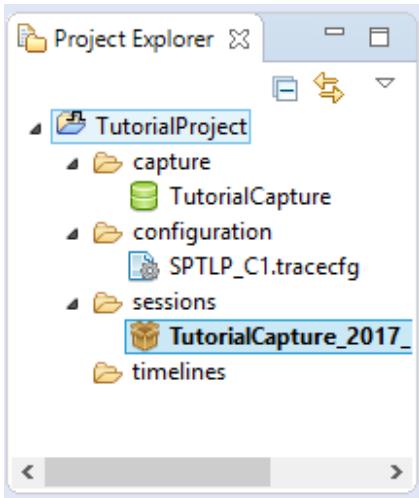
or



2. The following progress dialog appears:

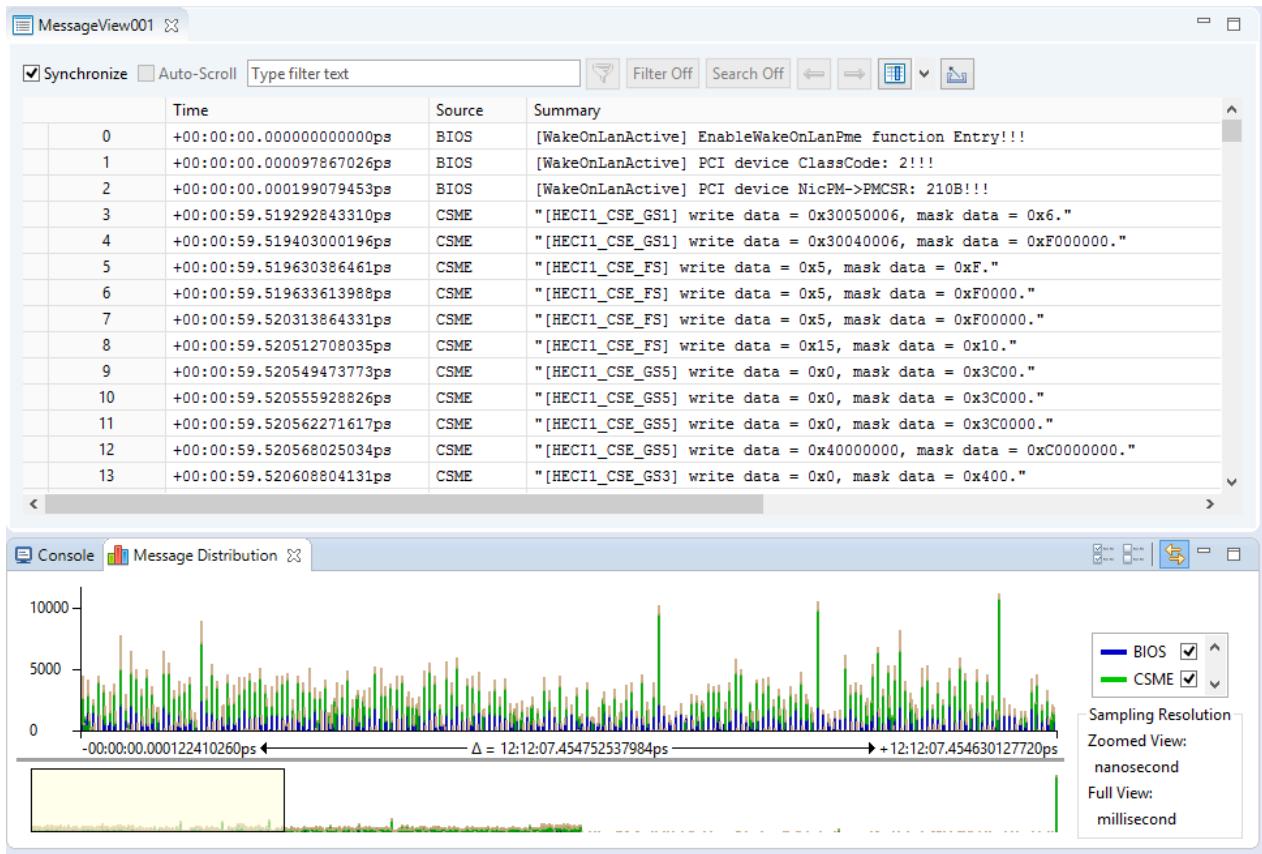


3. In the **Project Explorer** view, the **Trace Capture** file will appear in the capture folder, while the **decoded file** will appear in **sessions** folder.



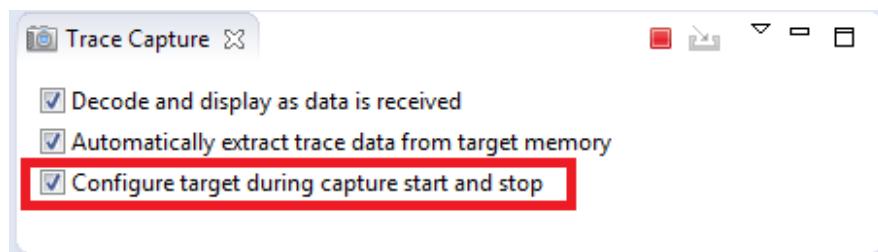
4. The **Message View** will open which will show the decoded **Trace Messages** in a table.

The link: [Message Distribution View](#) shows the distribution of different types of Trace Messages over time.



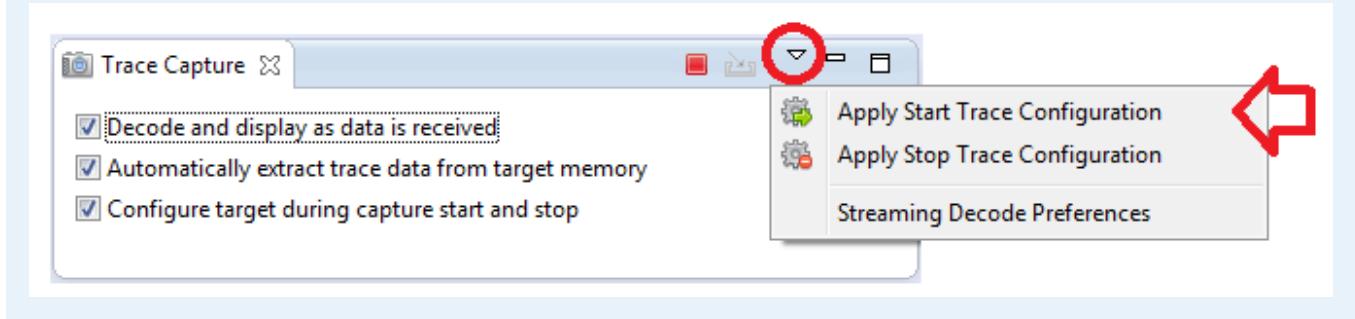
Advanced: Capture Trace without Changing Hardware Configuration

By default, the trace viewer configures the Intel(R) Trace Hub hardware for tracing and starts the capture of trace data at the same time. This overrides any existing trace configuration. To just capture data without changing the hardware configuration, uncheck this option in the **Trace Capture** view:

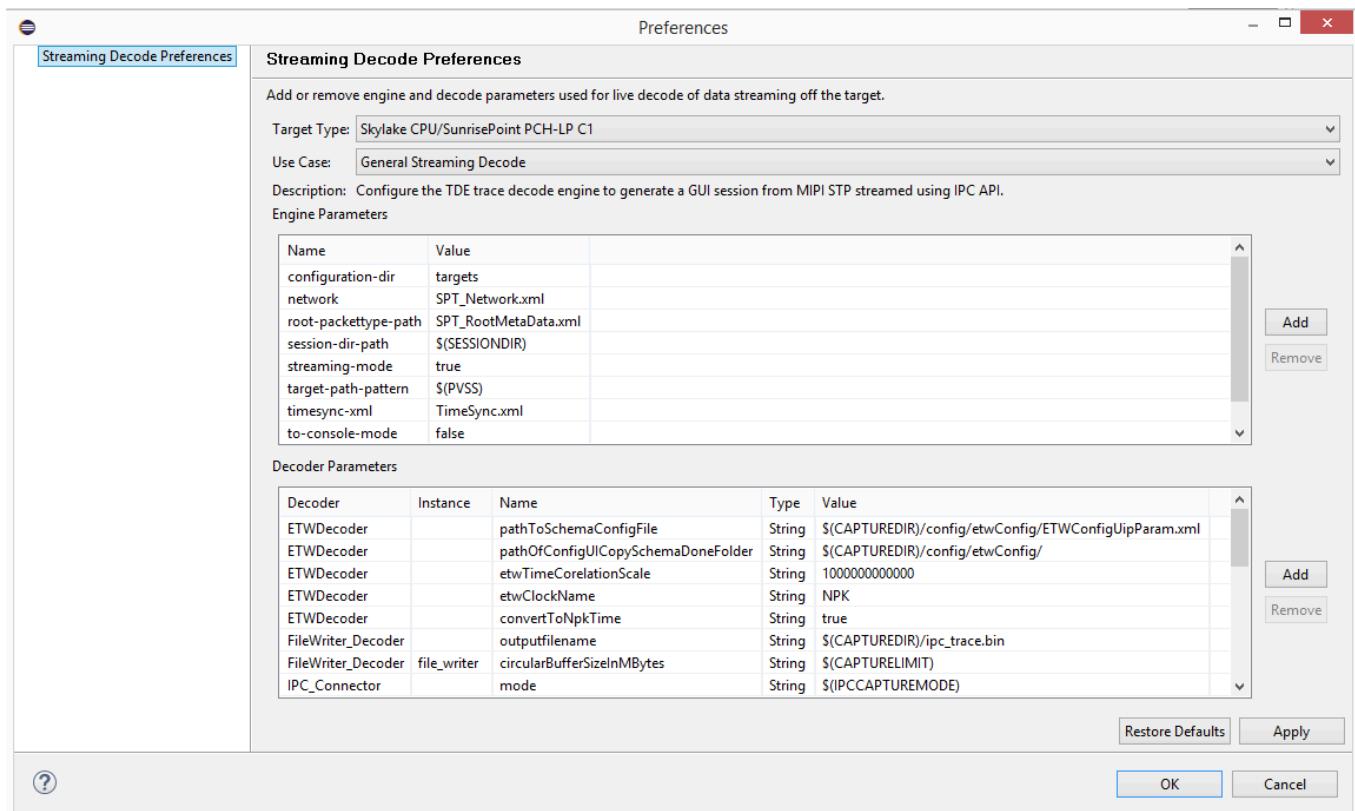
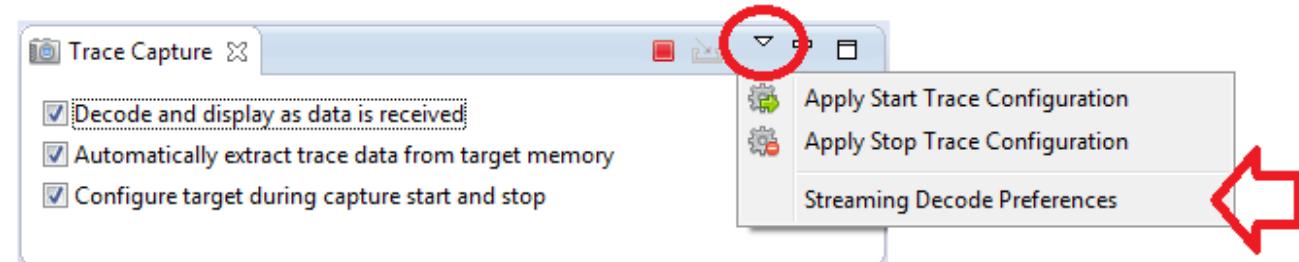


Note

You can still apply the start capture configuration settings manually during a running capture using the context menu of the **Trace Capture** view.



If you want to change the decoder parameters for streaming, open the **Streaming Decode Preferences** dialog and set your preferred options.



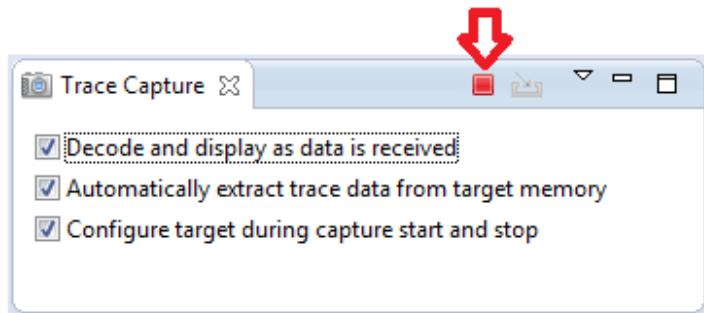
Close the dialog with **OK** and start live tracing.

! Warning

Changing streaming parameters is an expert setting and requires detailed knowledge about decoders.

Stop Capturing Trace

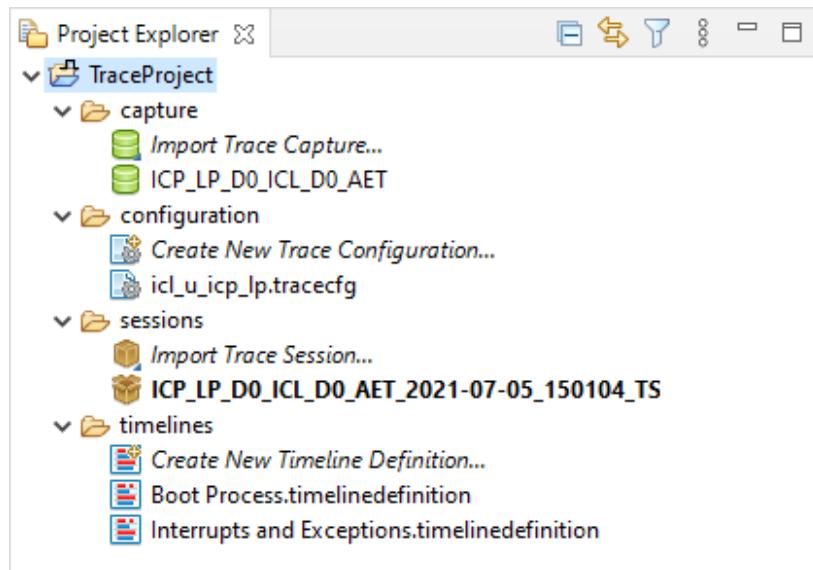
To stop the live tracing, click the **Stop Capture** button in the Trace Capture view.



The System Trace stops the live trace capture.

File Types in System Trace

System Trace stores necessary information in different file types displayed in the Project Explorer view:



See the table below to learn what each file type is used for.

File Type	File Extension	Usage
Trace Capture file	.tracecpt	Raw capture data collected during trace capture session
Trace Extension file	.traceext	Additional information that helps the tool to decode captured data
Trace Session file	.tracesession	Decoded capture data in a human-readable format (session log)
Trace Configuration file	.tracecfg	Configuration file that contains information on trace settings

File Type	File Extension	Usage
Timeline Definition file	.timelinedefinition	Visualization of trace data from the trace session. Show

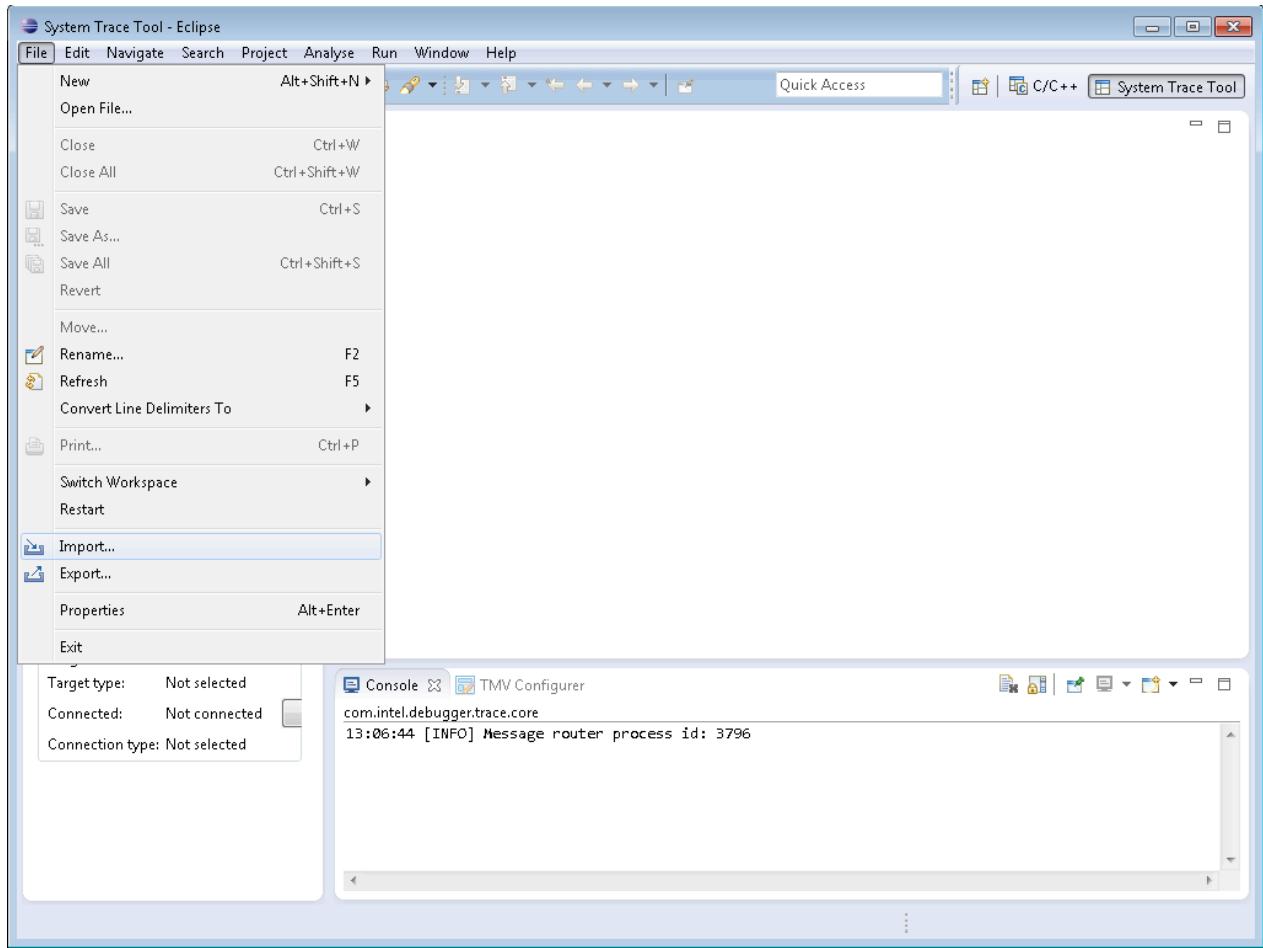
Import Pre-captured Trace File

You can import pre-captured trace file (Trace Capture Archive [.tracecpt](#) or plain [.bin](#) file) into System Trace to analyze the data.

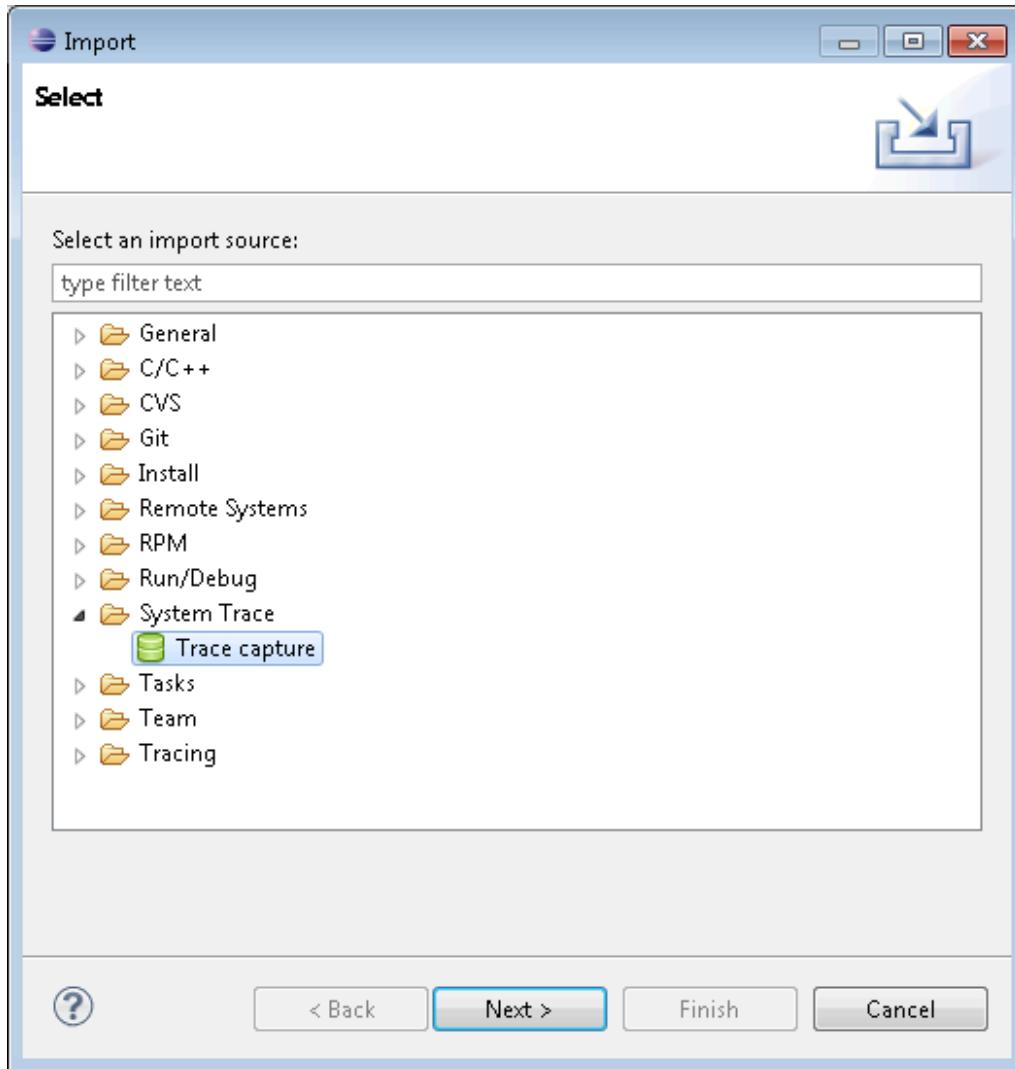
To import the file, follow the steps below.

1. Complete startup steps described [here](#) to ensure proper decode of the imported file. ▶

2. Go to **File > Import:**

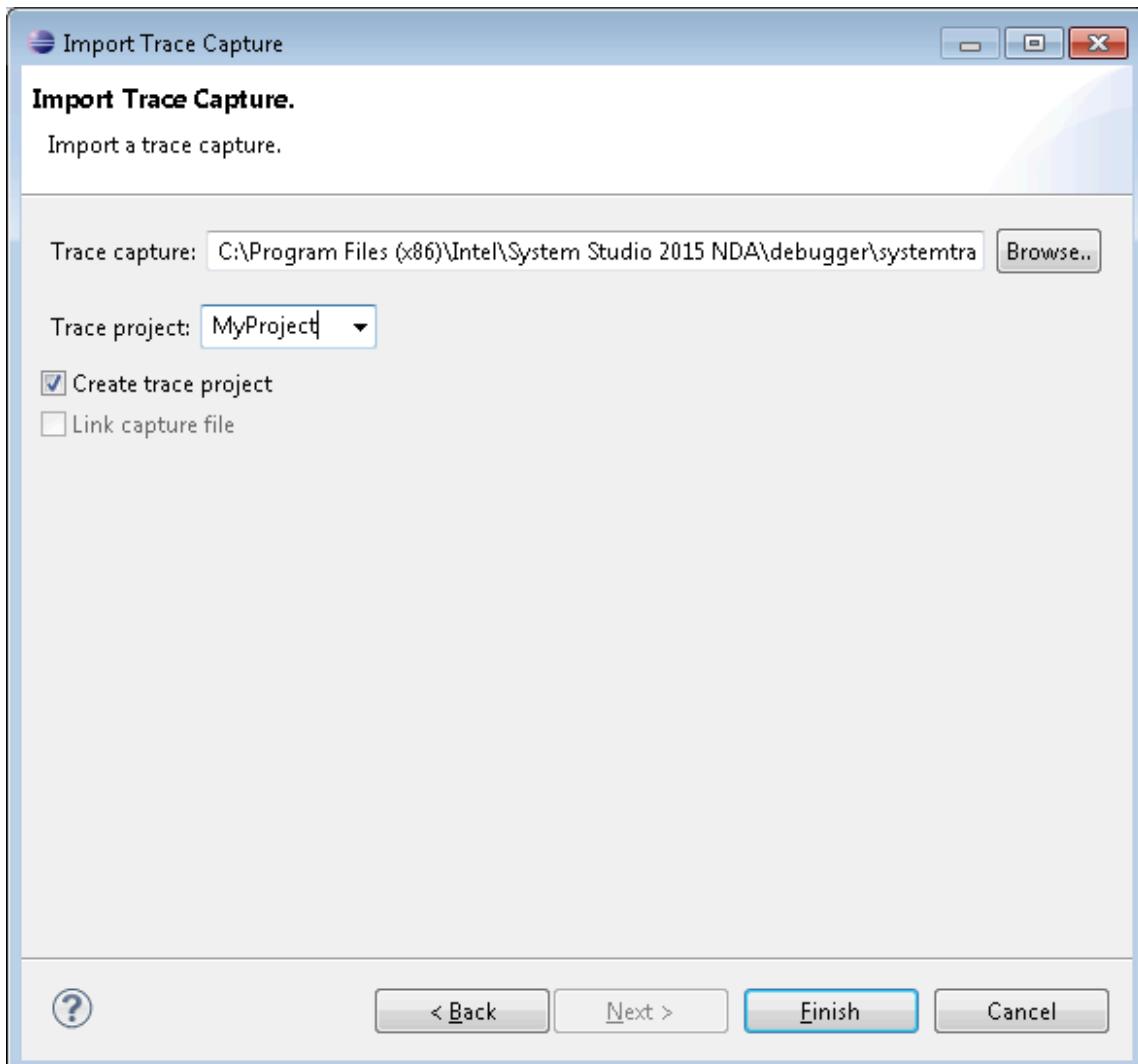


3. Expand the list and select **System Trace > Trace capture:**

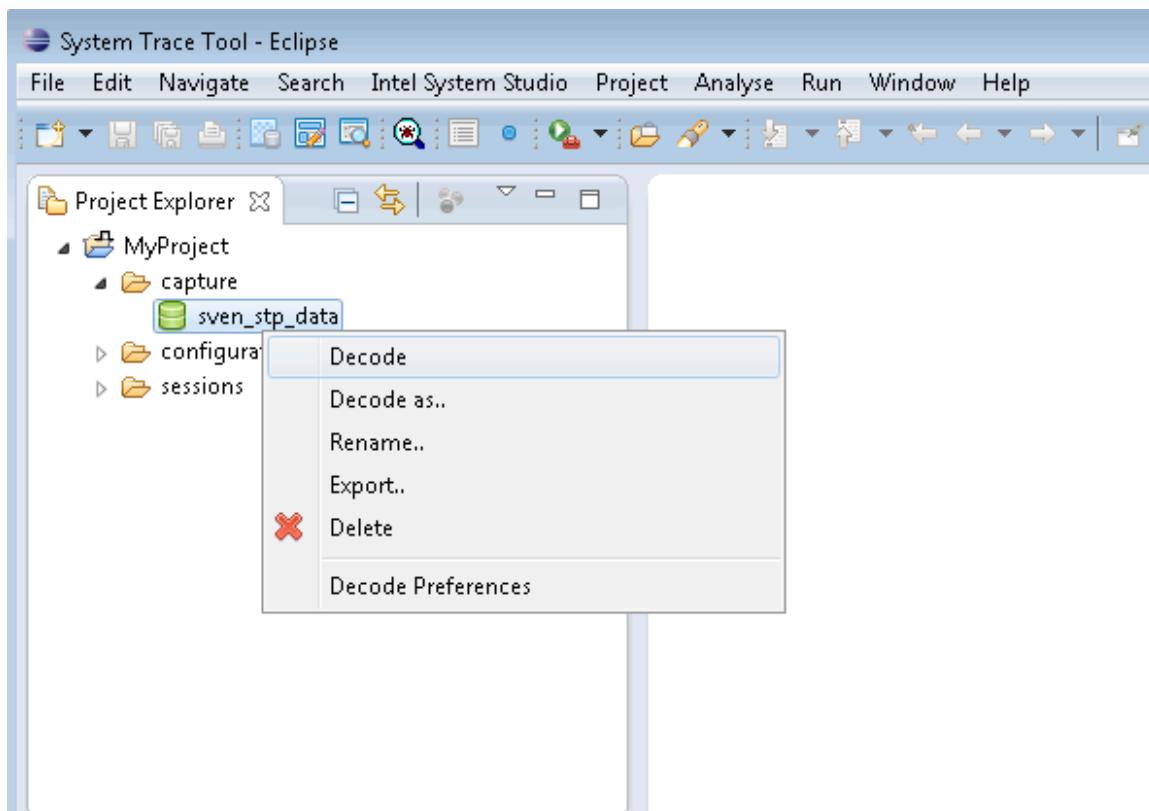


Click **Next >**

4. In the opened dialog box, click **Browse**, choose the file to import, and click **Open**.
5. Select the Trace Project you want to import the data to from the drop-down list. If you do not have a project yet, type in a name for your new project and check the **Create trace project** box.



6. Click **Finish** to close the dialog box.
7. In the **Project Explorer** view, expand the **capture** node. You should see a new entry named after the imported file.
8. Right-click the entry and select **Decode**.



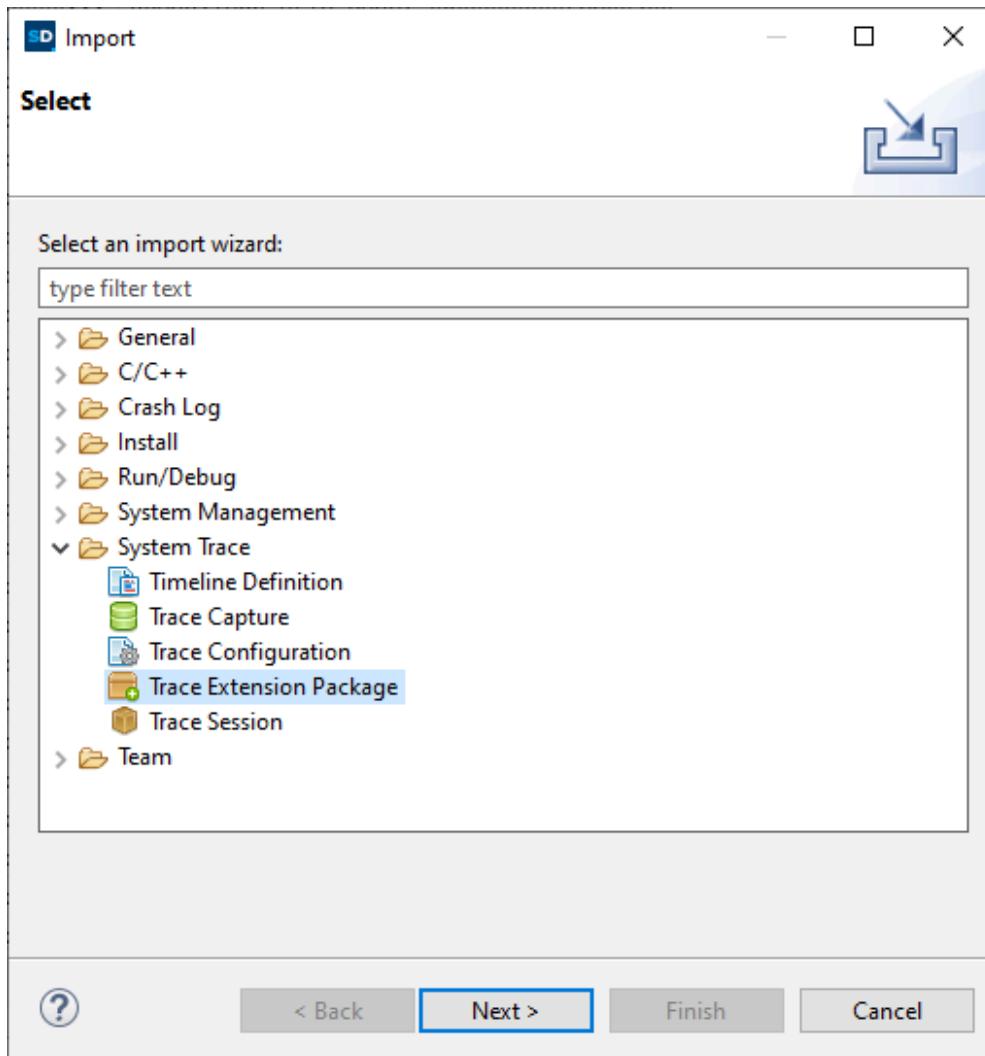
The sample trace is displayed in the trace viewer.

Index	FORMATTED GTS	_Summary
0	00:00m:00s 000.000ms	SVEN: 1:None t:os_clock fmt=epoch_microseconds 2014-Apr-16 12:00:00.000100 GMT m:
1	00:00m:00s 000.015ms	SVEN: 1:None t:debug_string s:generic Hello world! m:0A16878865C93B469F431EFE9FD
2	00:00m:00s 000.030ms	SVEN: 1:User1 t:debug_string s:generic Hello world! m:0A16878865C93B469F431EFE9FD
3	00:00m:00s 000.045ms	SVEN: 1:Normal t:debug_string s:generic Hello world! m:0A16878865C93B469F431EFE9FD
4	00:00m:00s 000.060ms	SVEN: 1:Normal t:debug_string s:generic Hello world! m:0A16878865C93B469F431EFE9FD
5	00:00m:00s 000.075ms	SVEN: 1:Normal t:debug_string s:generic Hello world! m:0A16878865C93B469F431EFE9FD
6	00:00m:00s 000.090ms	SVEN: 1:Normal t:debug_string s:func_enter main m:0A16878865C93B469F431EFE9FD
7	00:00m:00s 000.105ms	SVEN: 1:Normal t:debug_string s:func_enter main m:0A16878865C93B469F431EFE9FD
8	00:00m:00s 000.120ms	SVEN: 1:Warning t:catalog_msg s:cat64 A 64bit catalog message from svntrace/example
9	00:00m:00s 000.135ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 1 parameter - p1=1 m:0A1687886
10	00:00m:00s 000.150ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 2 parameters - p1=1 p2=0x2 m:0
11	00:00m:00s 000.165ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 3 parameters - p1=1 p2=0x2 p3=
12	00:00m:00s 000.180ms	SVEN: 1:Warning t:catalog_msg s:cat64 Undefined catalog ID 0x4 m:0A16878865C93B46
13	00:00m:00s 000.195ms	SVEN: 1:Warning t:catalog_msg s:cat32 A 32bit catalog message from svntrace/example
14	00:00m:00s 000.210ms	SVEN: 1:Warning t:catalog_msg s:cat32 Many think that 42 is the meaning of life m
15	00:00m:00s 000.225ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p2=0x2 p3=
16	00:00m:00s 000.240ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p2=0x2 p3=
17	00:00m:00s 000.255ms	SVEN: 1:Warning t:catalog_msg s:cat64 Message with 6 parameters - p1=1 p2=0x2 p3=
18	00:00m:00s 000.270ms	SVEN: 1:Error t:debug_string s:assert example.c:138 m:0A16878865C93B469F431EFE9FD

Import Trace Extension

Importing a trace extension is required for decoding particular trace messages properly (for example, CSME traces). Without an extension applied, undecoded messages contain the following text in the Summary column of the **Message** view: `Undefined message with ID ...`.

1. In the **Project Explorer**, right-click any empty space and select **Import**
2. In the opened dialog box, expand the **System Trace** node, select **Trace Extension Package**, and click **Next**.



3. Click **Browse** to find and add the `.traceext` file.
4. Click **Finish**.

Alternatively, open the **System Trace Extensions** view and click **Import System Trace Extension**.

See also

[Capture and Decode Early-boot CSME Traces in a Cold Boot Scenario](#)

Analyze Trace

Analyze Trace Messages with Message View

The main interface object for trace analysis is the **Message View**, which opens automatically when trace messages are captured and decoded.

The Message View displays a table with all decoded trace messages captured from the target. You can configure the table to display specific contents of the trace messages or filter/search to find for specific message contents.

Decode the Summary Message with Trace Extension

The Summary column of the **Message View** may show message strings that are not decoded correctly. See an example below:

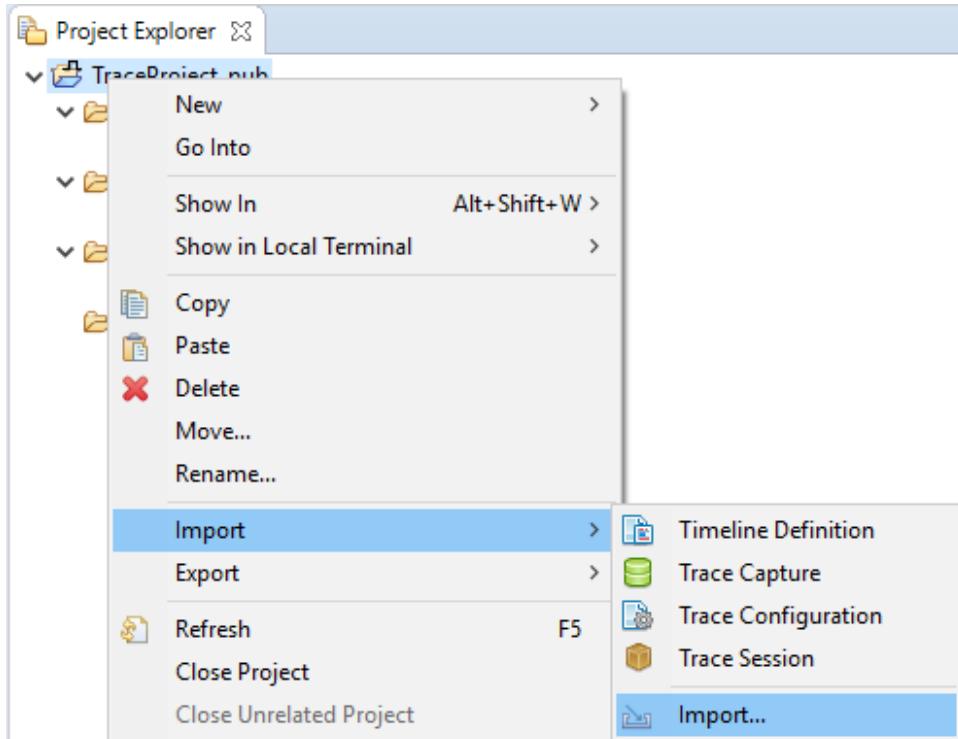
	[Intvl]Time	Summary
8	[0]+00:00:00.070390541687	Undefined message with ID 0x000000000000005257
9	[0]+00:00:00.073541315104	Undefined message with ID 0x000000000000005258
10	[0]+00:00:00.073541886393	Undefined message with ID 0x000000000000005291
11	[0]+00:00:00.073549228312	Undefined message with ID 0x000000000000005295
12	[0]+00:00:00.073549592997	Undefined message with ID 0x000000000000005296

To decode the summary message properly, use a Trace Extension - an archive package with firmware trace dictionaries used to decode trace data into human-readable text.

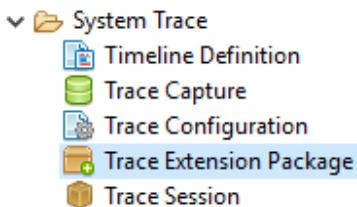
Trace Extensions are available as part of the Intel(R) Best Known Configuration Package Manager. They can be found by searching for files ending in `.traceext`.

To import a Trace Extension:

1. In the **Project Explorer** view, right-click your Trace project and select **Import** twice:



2. Under System Trace node, select **Trace Extension Package** and click **Next**.

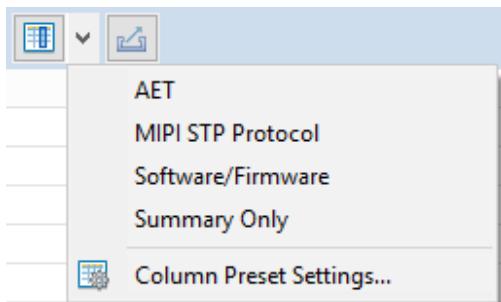


3. In the new dialog box, click **Browse** and navigate to the required **.traceext** file.
4. Click **Finish**.
5. The package is opened in the **System Trace Extensions** view.
6. To apply the extension, restart **trace capturing** or redecode **imported trace files** and check the **Message View**.

Show the Source of Traces

To verify from which trace source the decoded messages in the Message View are coming, do the following (assuming the Message View is open and displays messages):

1. In the Message View toolbar, click the small arrow near the column picker button to show the available column presets and select **MIPI STP Protocol** or **Software/Firmware** preset.



2. This column preset will show the **Source** column, which displays the origin of a trace message:

A screenshot of the Message View table. The table has columns for Intvl, Time, Source, Severity, and Summary. The 'Source' column is highlighted with a red box. The data in the table is as follows:

	[Intvl]Time	Source	Severity	Summary
21	+00:00:00.193236583252	0x0000090018		IN(0x00001808)=0x00F3C9C1
22	+00:00:00.193598333333	0x0000090018		IN(0x00001808)
23	+00:00:00.195988750000	0x0000090018		IN(0x00001808)=0x00F3C9CD
24	+00:00:00.196350333252	0x0000090018		IN(0x00001808)
25	+00:00:00.198740749919	0x0000090018		IN(0x00001808)=0x00F3C9D9

Select Trace Fields to Be Displayed

In many cases, you are interested in only a certain set of trace fields, which are shown as columns in the trace viewer. You can customize the column display of the trace viewer with the **Select Columns** dialog in **Message View**.

1. Open **Select Columns** dialog from the **Message View** toolbar :

F S	Type filter text								
	Summary	LIP	VECTOR	ADDR					
	IN(0x00001808)=0x00F3C9C1	0x000000003D40D53D		0x00001808					

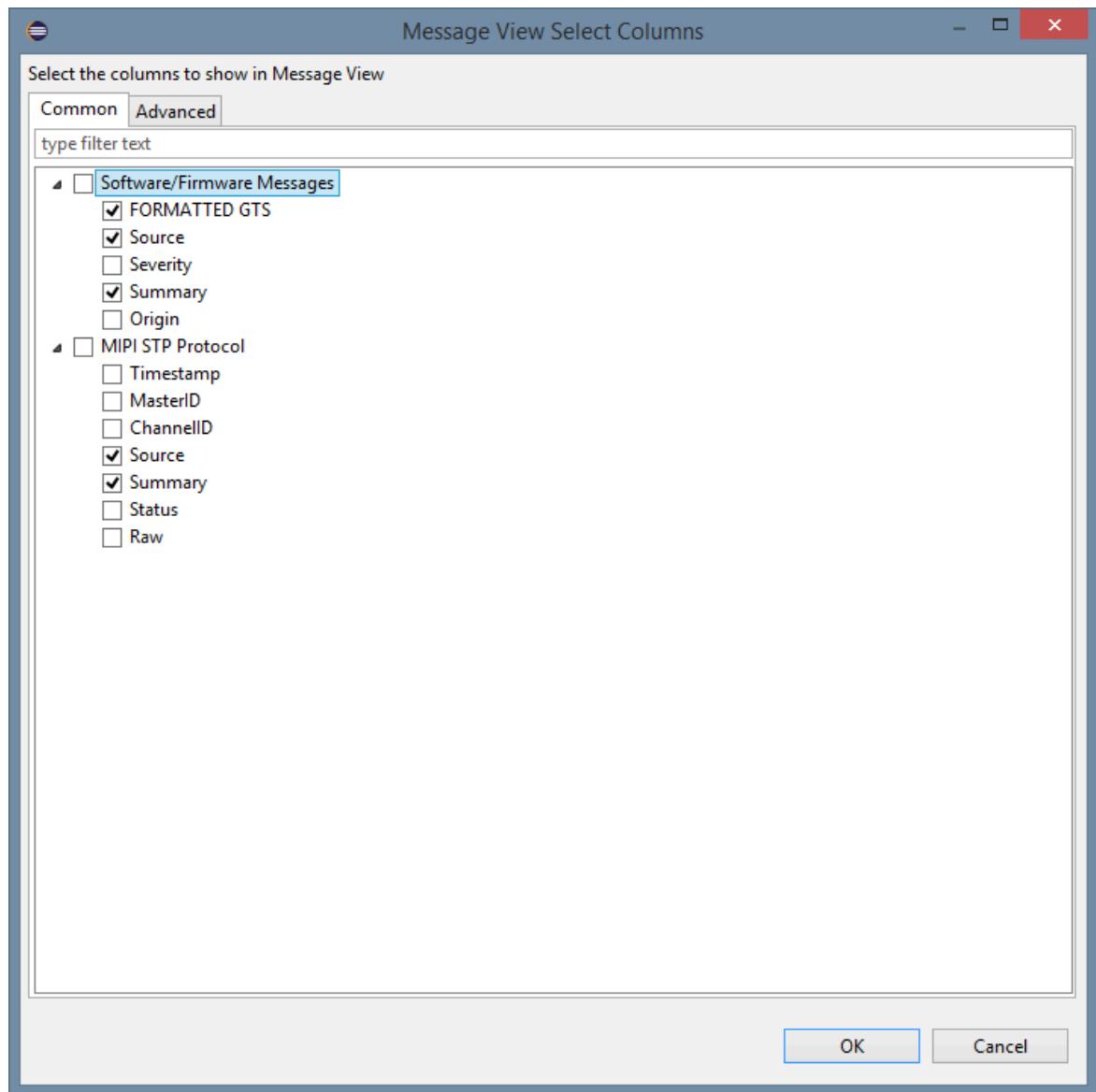
Or right click in message view and pick “Select Columns...” item from context menu.

F S	Type filter text								
	[Intvl]Time	Source	Severity	Summary					
21	+00:00:00.193236583252	0x000009...		IN(0x00001808)=0x00F3C					
22	+00:00:00.193598333333	0x0	Copy						
23	+00:00:00.195988750000	0x0	Marker						
24	+00:00:00.196350333252	0x0	Search						
25	+00:00:00.198740749919	0x0	Copy Scenario Expression						
26	+00:00:00.199102333577	0x0	Select Timestamp Format						
27	+00:00:00.201492166748	0x0	Time difference						
28	+00:00:00.201834333496	0x0							
29	+00:00:00.204223666585	0x0	Select Columns...						
30	+00:00:00.204585500081	0x0	Column Presets						

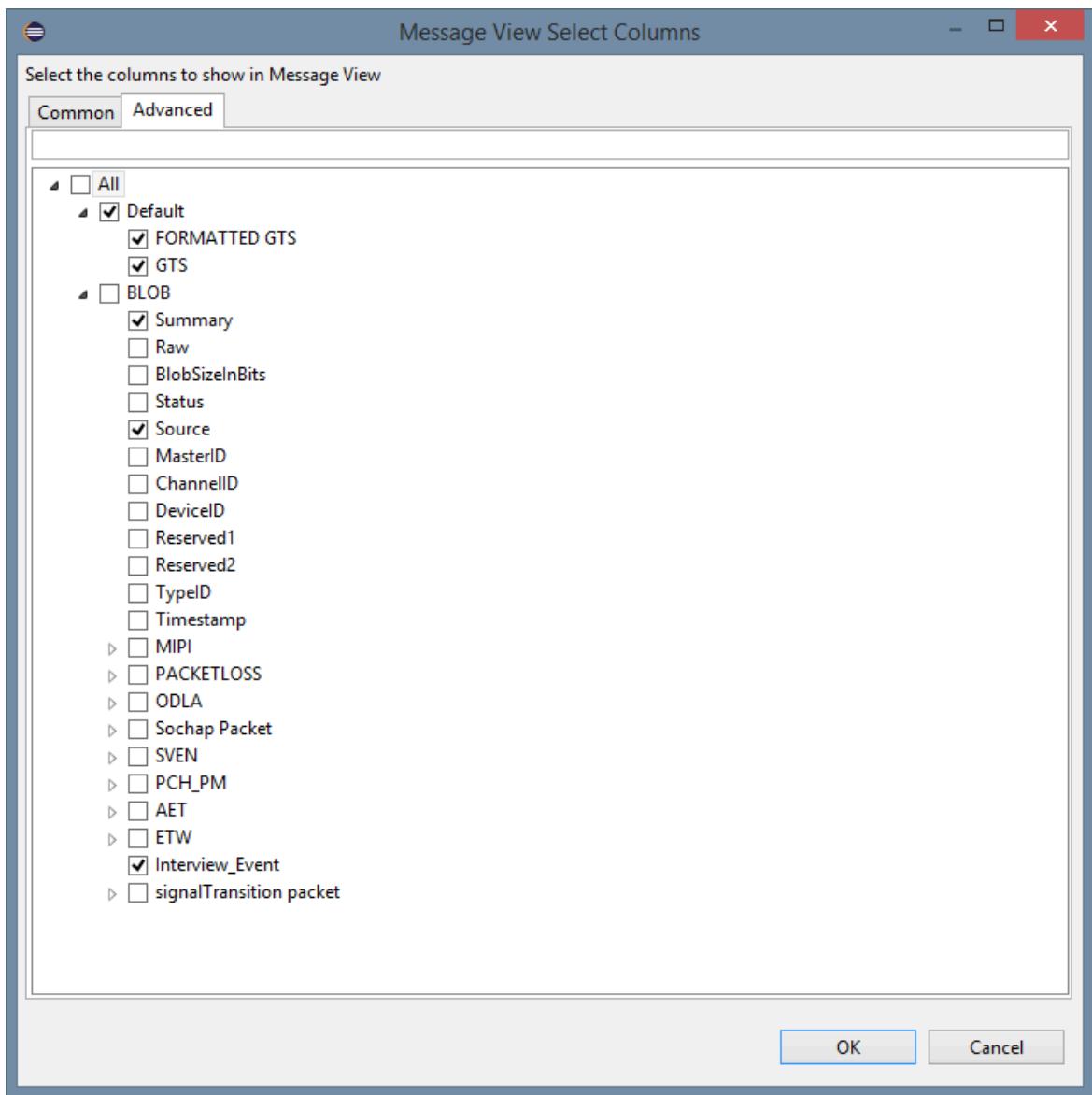
2. The Message View Select Columns dialog is opened.

The dialog contains two tabs: **Common** and **Advanced**. The selection is synchronized between the tabs.

- The **Common** tab displays commonly used fields and grouped by use cases or protocols:



- The **Advanced** tab displays every selectable field for a session, where the metadata fields retain their hierarchical structure within **BLOB** as the root node of the metadata tree. The **Default** node contains additional fields that are not part of the metadata.



! Note

The **All** node is great for clearing selection, but it is not recommended to simply enable all fields in **Advanced** tab. A maximum of 32 fields can be selected.

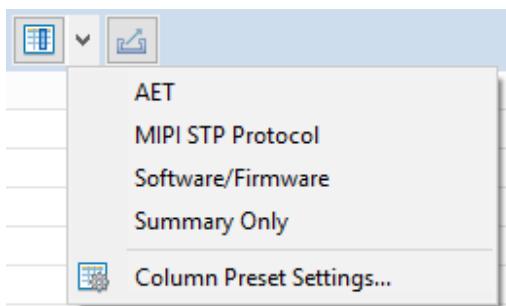
Some fields are common fields and are shared across protocols. When you enable or disable one of these fields, it will be enabled or disabled across all protocols.

3. To implement your selection, click **OK** in the Message View Select Columns dialog.

The trace viewer now displays all selected fields of the session.

Create Column Presets

The column preset feature allows you to save column settings and load them very easily. In the Message View, expand the drop down menu to show the available column presets:

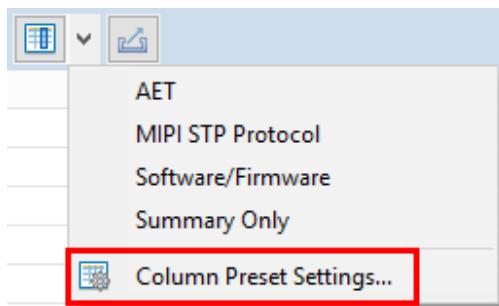


To **create** a new custom column preset, do the following:

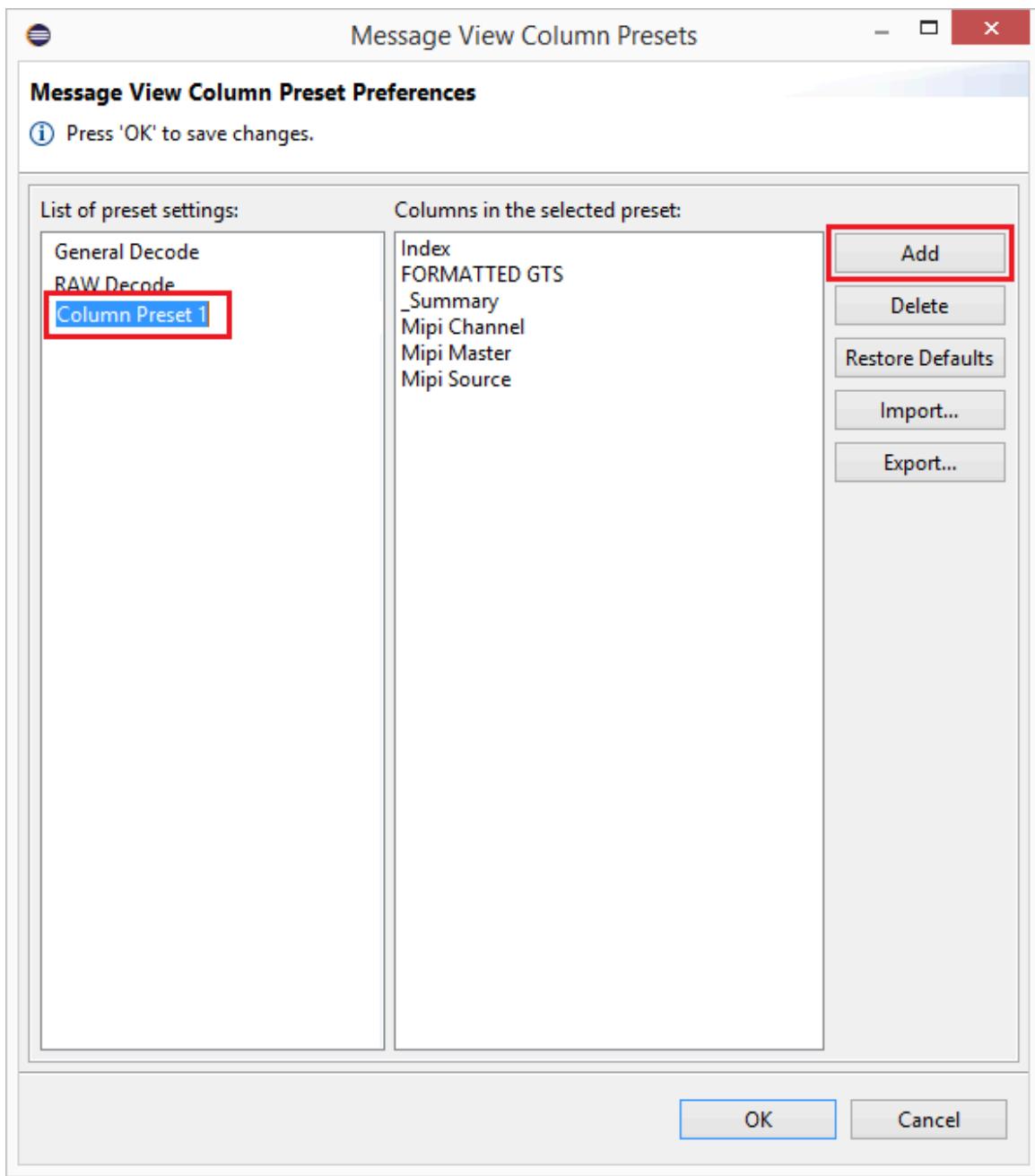
1. Enable all columns in the Message View you want to have in your new preset.

See [Select Trace Fields to Be Displayed](#).

2. Select **Column Preset Settings** from the column picker drop down menu.



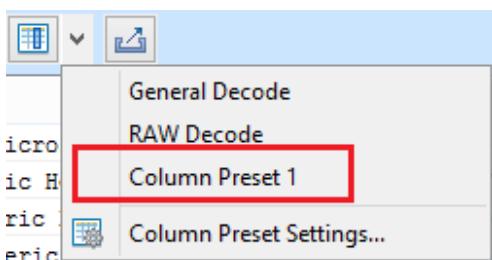
3. In the new dialog, click **Add** to create a new column preset.



In the left **List of preset settings***column, you can type in a new name for the column preset. The right column (***Columns in the selected preset**) shows the columns you have activated in the Message View. If you want to rename the new column preset, right click on the entry and choose **Rename**.

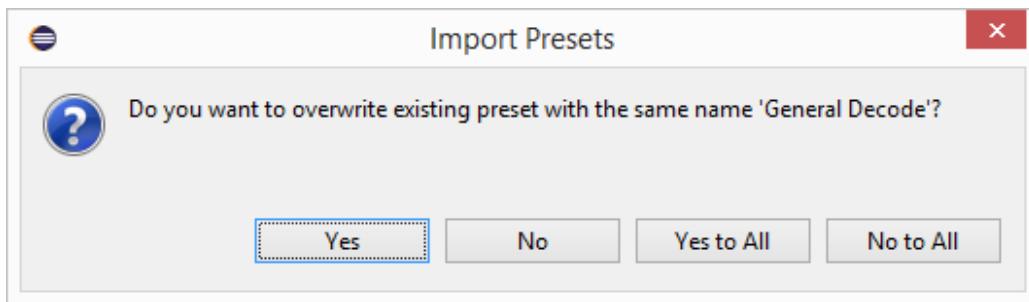
4. Click **OK** to save the preset in your workspace.

5. The new column preset will show up in the column picker drop down menu:



To **import** column presets, do the following:

1. Open the **Column Preset Settings** and click **Import** on the right.
2. Select the **.columnOrder** file you want to import and click **Open**.
3. If the **Message View Column Presets** dialog detects that you try to import existing settings, it asks if the column preset should be overwritten or not:

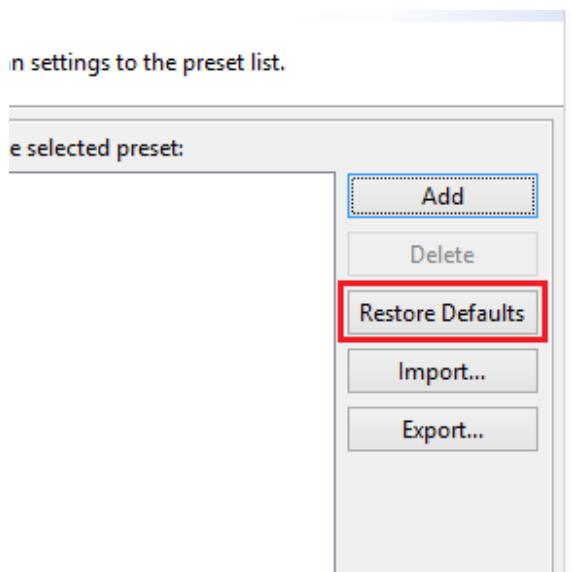


- Select **Yes** to overwrite the existing column preset in the list.
- Select **No** to skip importing the existing column preset.
- Select **Yes to All** to overwrite all existing column presets in the list from the import file.
- Select **No to All** to skip importing all existing column presets.

To **export** column presets:

1. Click **Export** in the **Column Preset Settings**.
2. Choose a location and a filename in the file picker and click **Save**.

To **restore** default presets, click **Restore Defaults** in the **Column Preset Settings**:



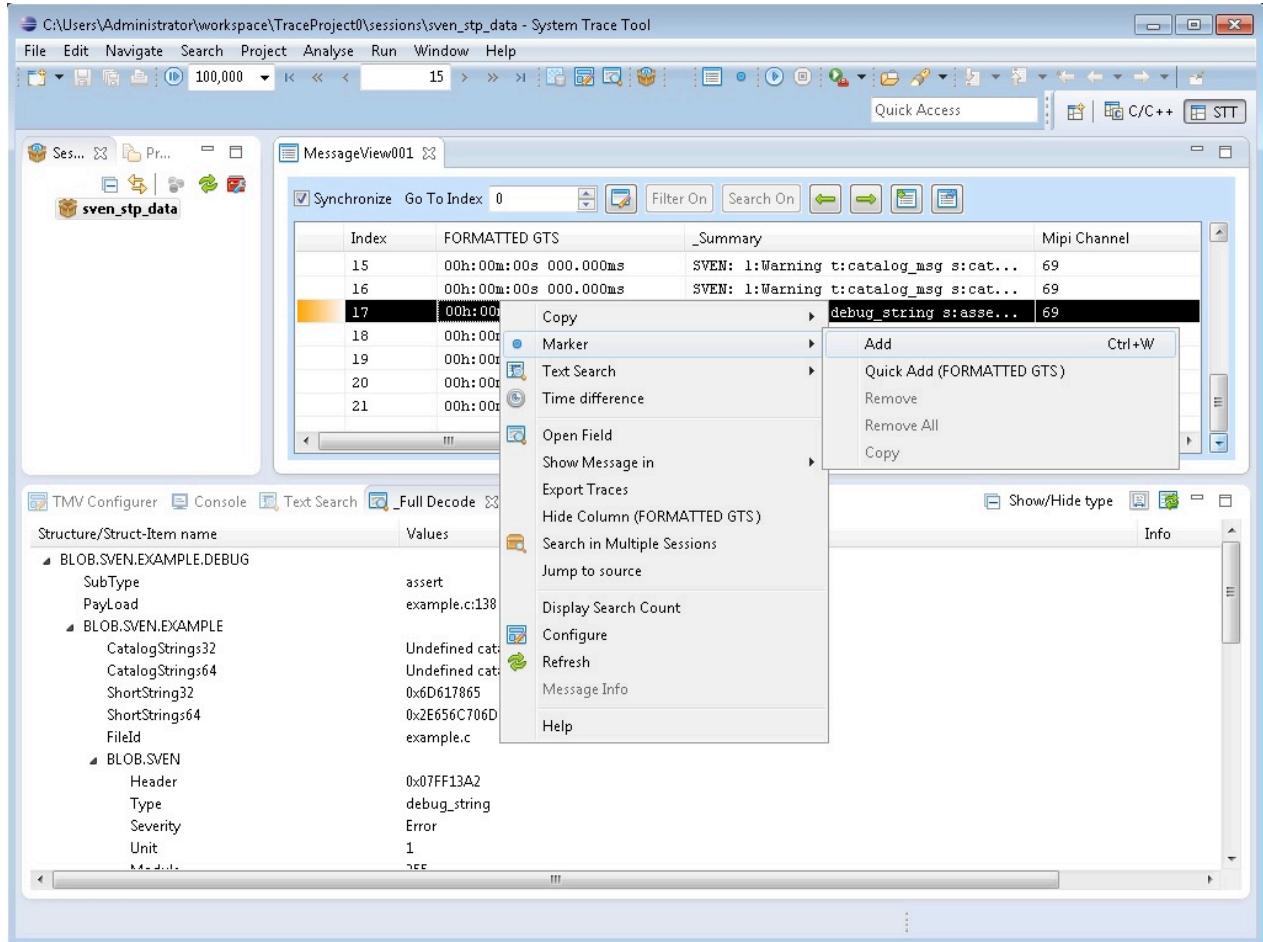
! Warning

Your custom presets will be removed.

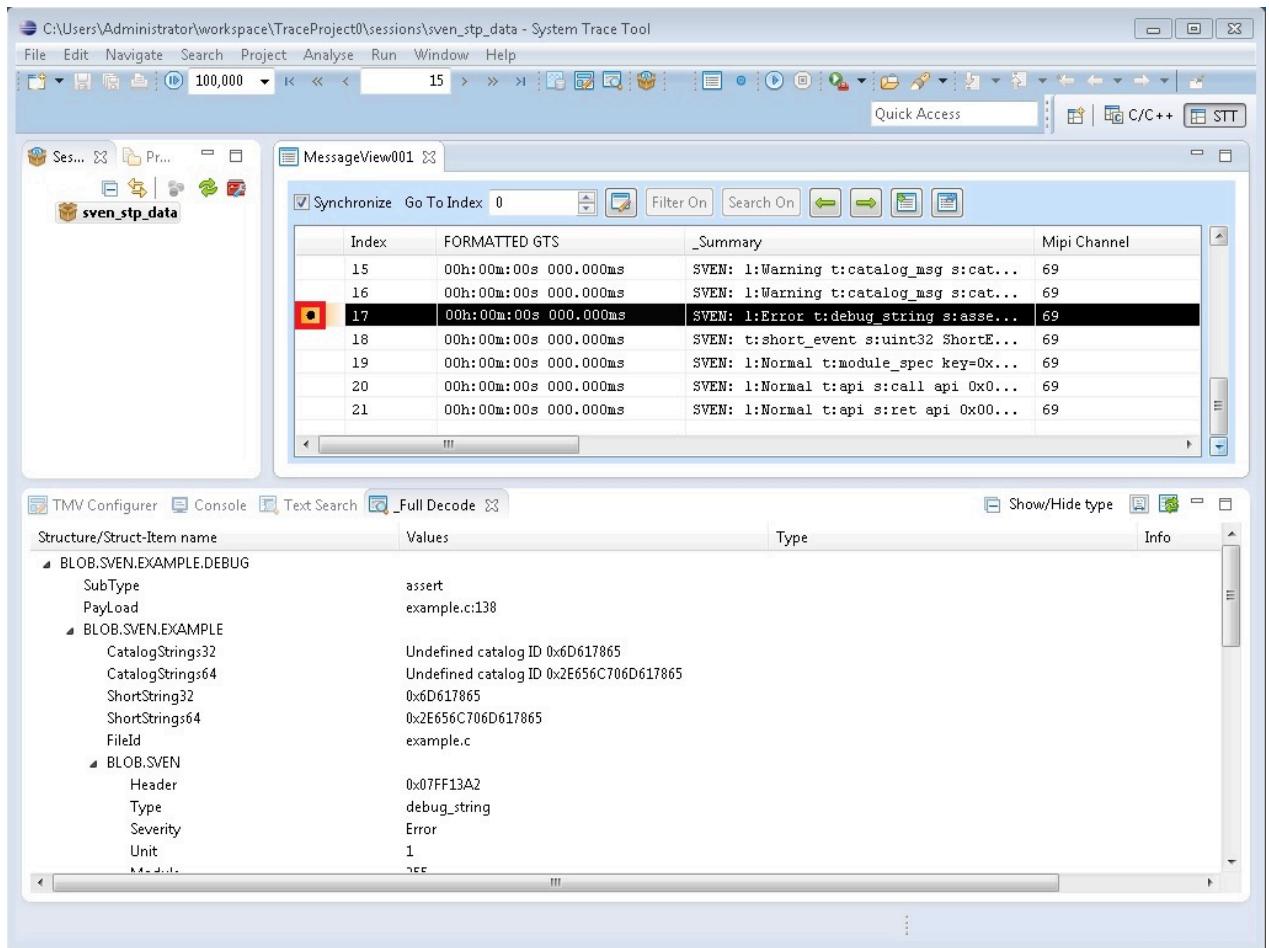
Mark Trace Messages

You can mark trace entries for later use. This is useful for recognizing them in a later stage of your project, for example if you wish to examine them in a team collaboration effort.

1. Assuming the SVEN Error trace message is selected, right-click the message. The trace entry's context menu is displayed.

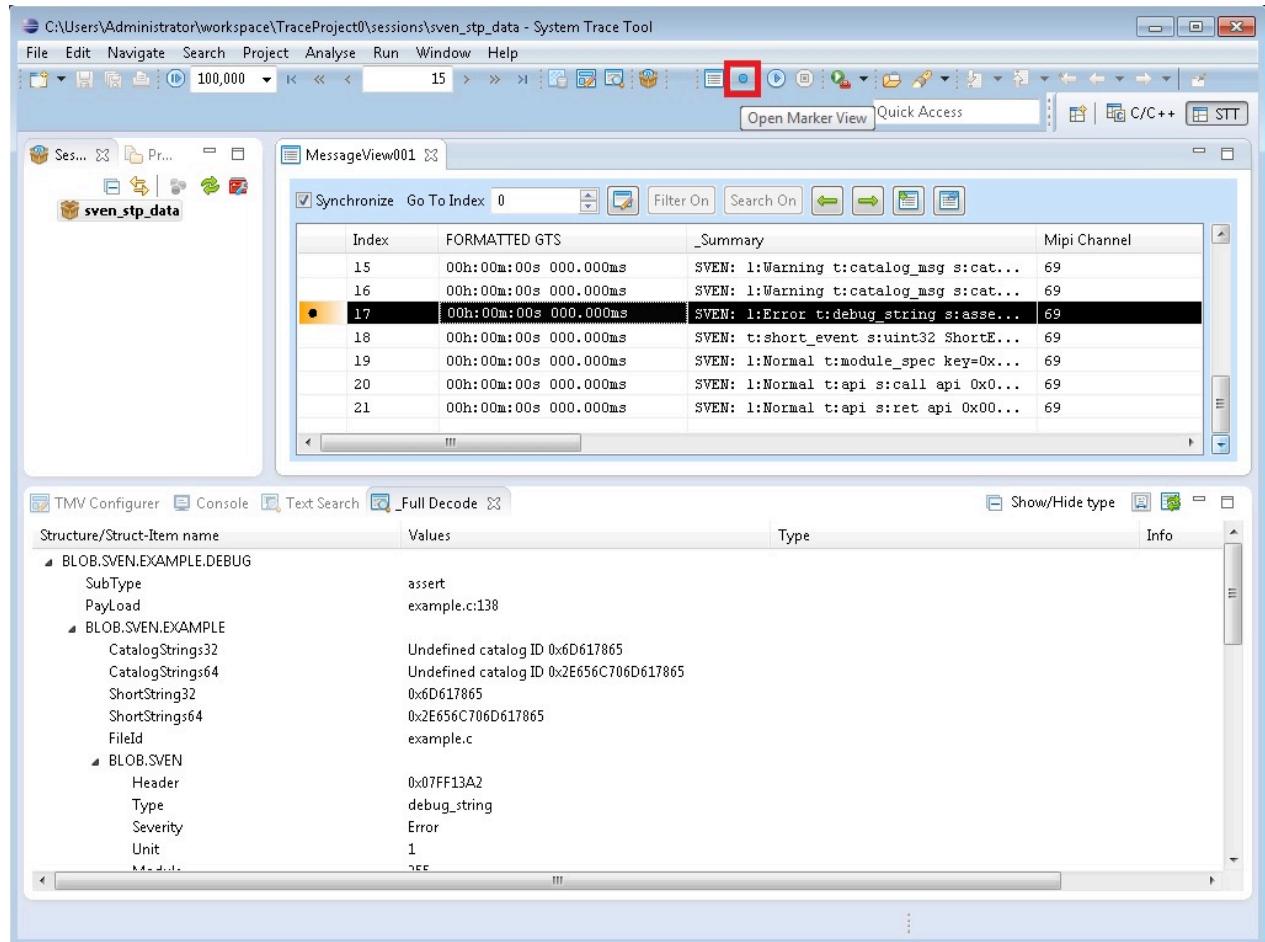


2. Select **Marker > Add** from the context menu. A marker bubble is added in the very first column of the trace view:

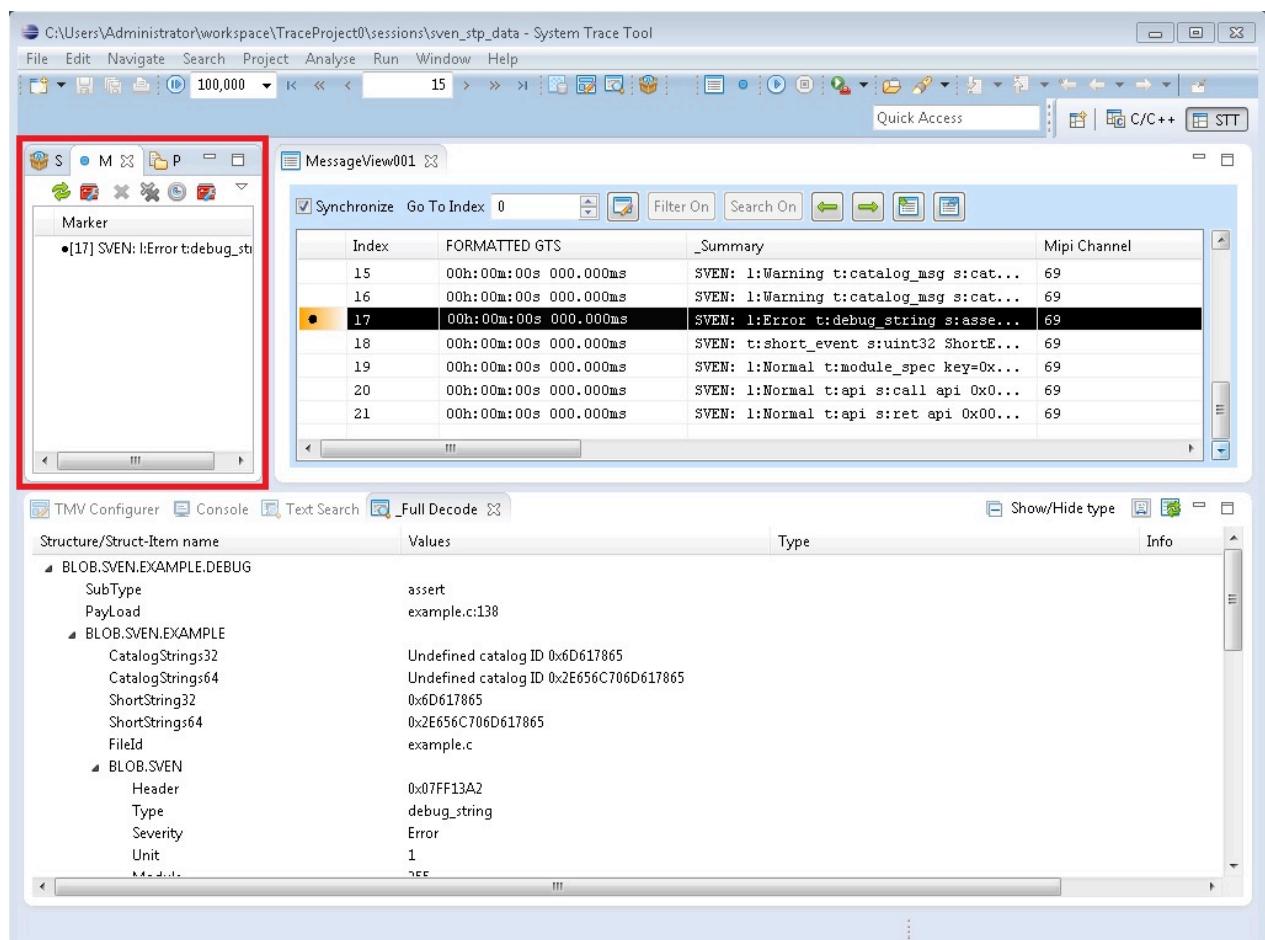


This works well for small numbers of markers in a trace. If there are more markers, a dedicated view displays all markers set in the trace.

3. To open this **Marker** view, click the **Open Marker View** icon in the toolbar.



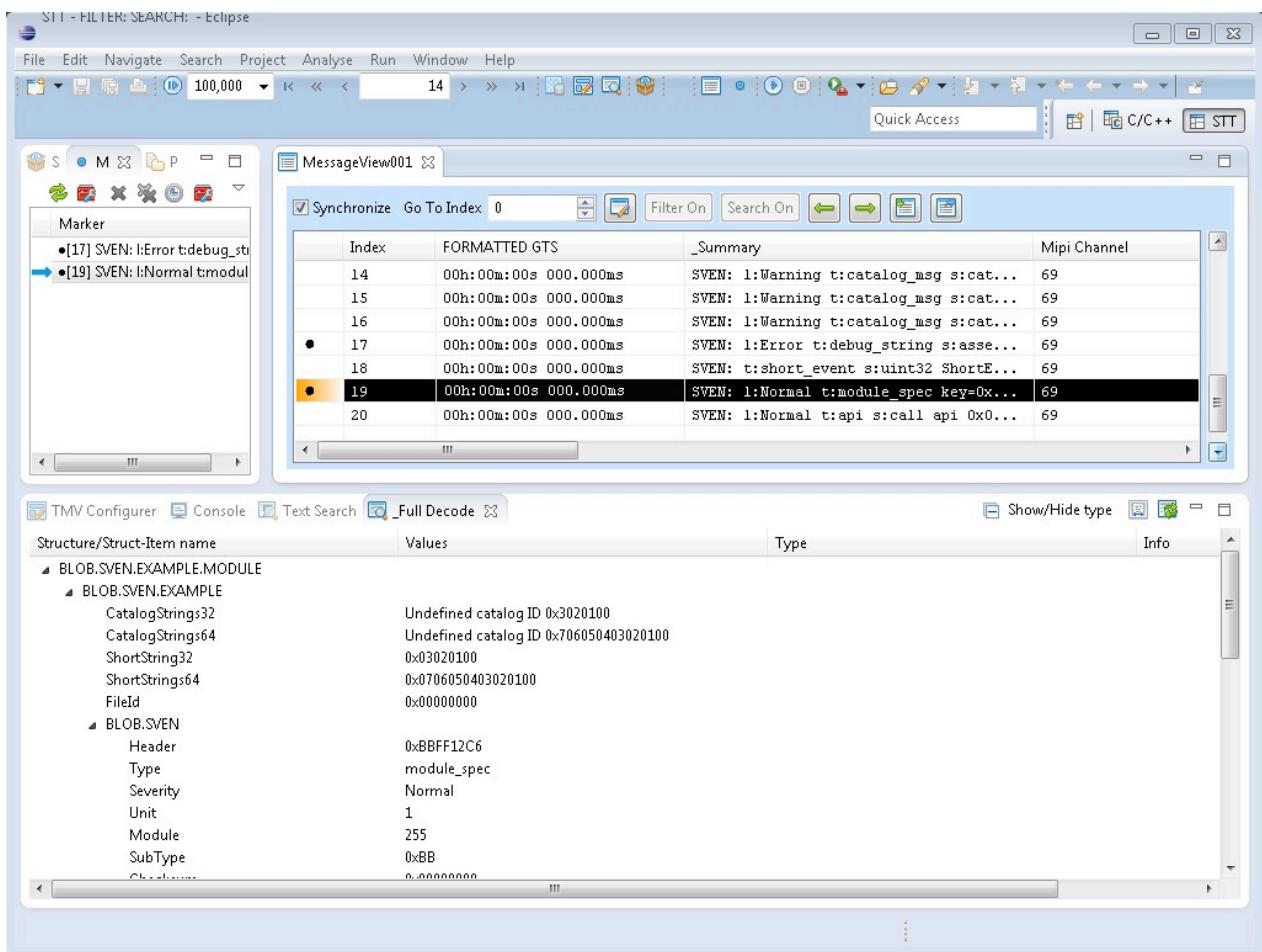
The Marker view is displayed on the top-left tabs.



The **Marker** view displays a list of all markers in a trace message view.

4. Add another marker on index 19 by selecting the respective trace entry with a left-click.

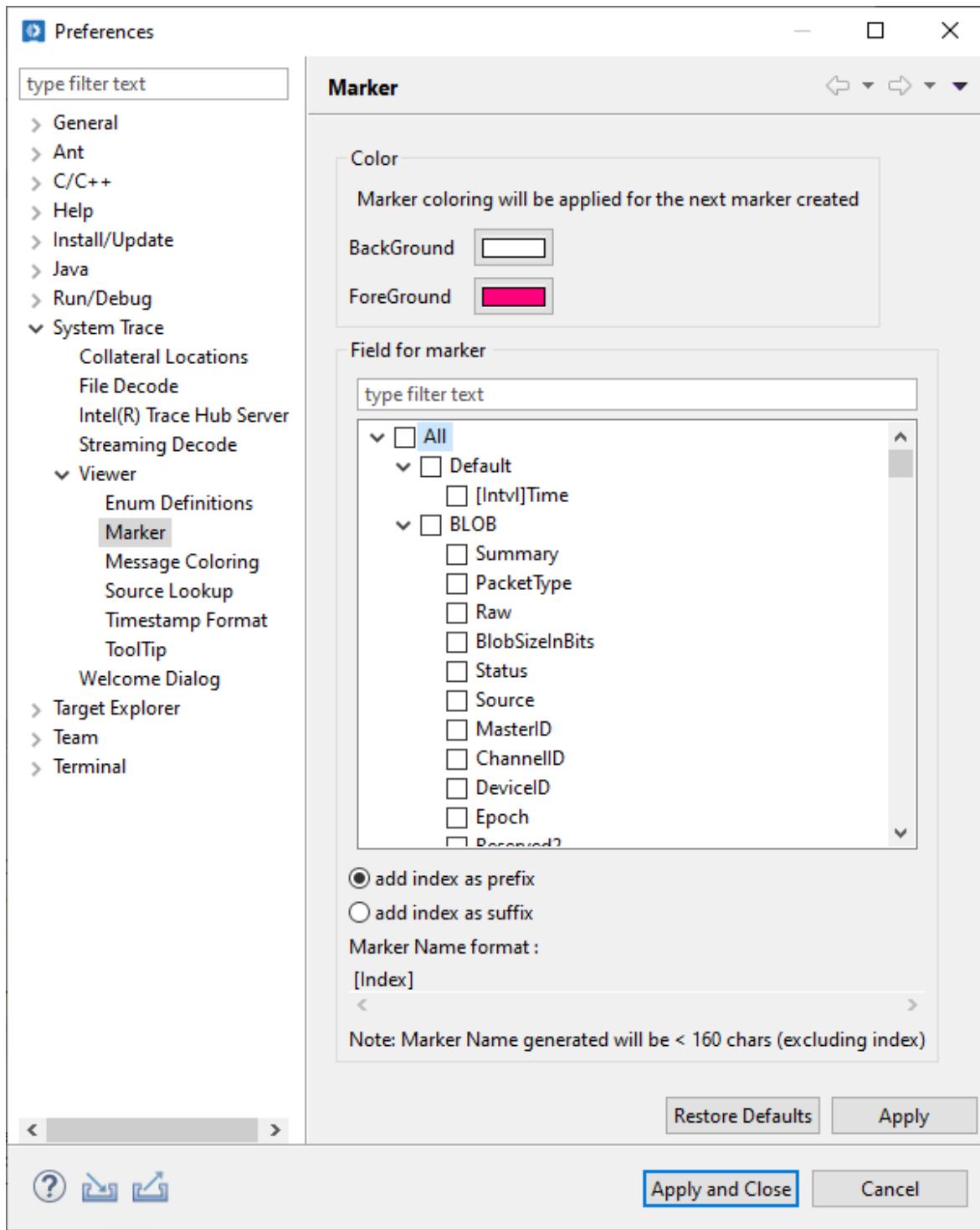
5. Open the context-menu by right-click and select **Marker > Add**. The Marker view now contains two entries, such as.



You can now use the trace marker entries in the Marker view to scroll quickly the trace to display interesting entries.

You can change the marker color and the displayed information the following way:

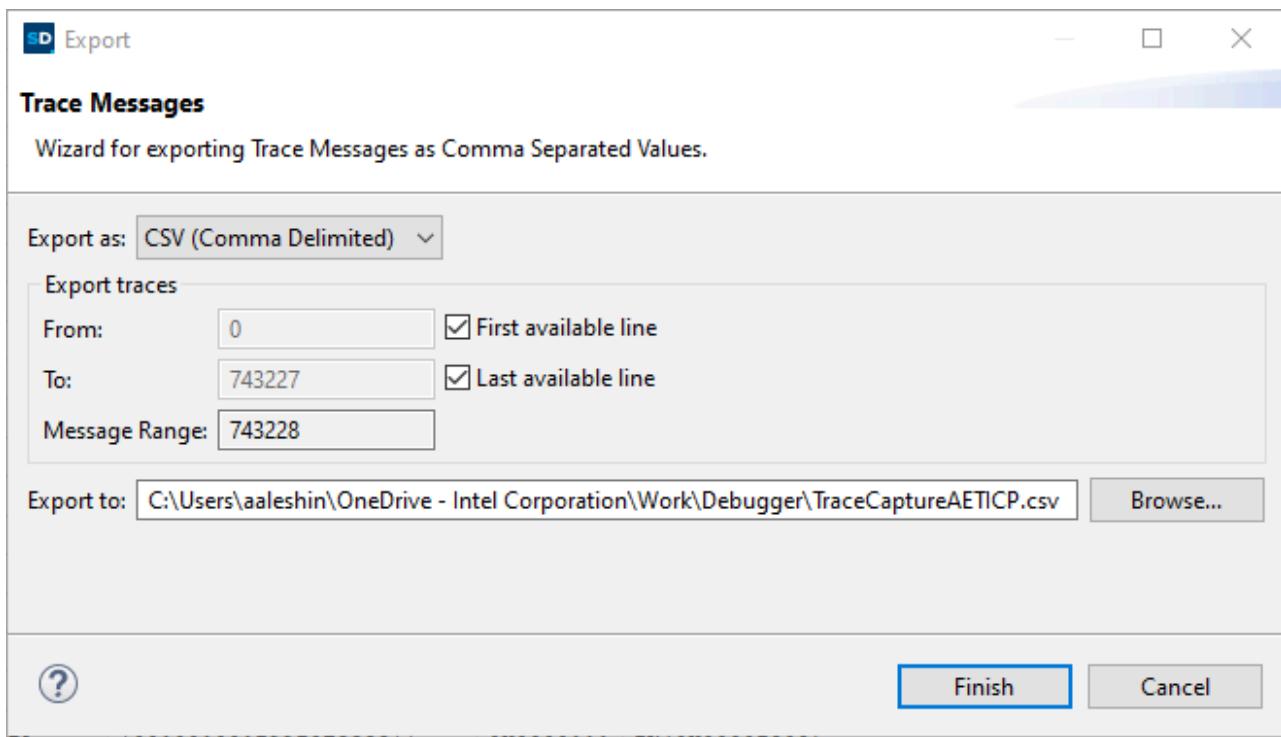
1. Select **Window > Preferences** to open the **Preferences** dialog box.
2. Navigate to **System Trace > Viewer > Marker** to view and modify the marker attributes.
3. Click **Apply** to save your changes or **Apply and Close** to save changes and close the **Preferences** dialog.



- If your trace contains timestamp information (the SVEN example does not), you can also examine, for example, the frequency/time interval in which a certain event occurs. To do so, select Time Difference from a marker's context menu and select a second entry by holding the <CTRL> key.
- Besides the basic trace analysis, there is also a more advanced way of searching, marking and filtering trace entries based on user-defined rules. The subsequent trace analysis sections in this document describe trace analysis based on rules.

Export Trace Messages to CSV

To export the decoded trace to a **.csv** or **.txt** file, click  **Export** in the Message view. In the opened dialog box, select the export type and specify the destination directory.



Alternatively, you can export decoded trace as a trace session ([.tracesession](#) file, which can be imported back to System Trace later). See [Export Captured Trace](#).

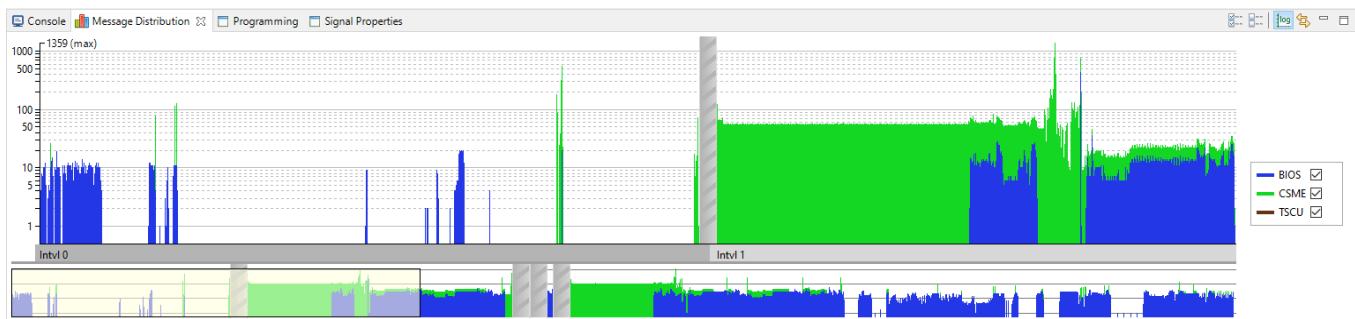
Analyze Message Distribution

The **Message Distribution** view visualizes the distribution of messages over time to provide a better overview of the trace and assist you in identifying interesting regions. This can be a good starting point for trace analysis.

To open the view:

1. Select **Window > Show View > Other...** from the menu bar.
2. Select **System Trace > Message Distribution** and click **OK**.

The **Message Distribution** view is opened.



The **Message Distribution** view contains the following panes:

- Graph pane on the left:
- Full View (the lower graph) shows the full range of the trace.

- **Zoomed View** (the upper graph) shows the selected range if set.
- **Source Selection** - the legend box on the right.

It lists all sources present in the trace with the corresponding color code in plot. Setting the checkbox shows or hides the source in the graphs accordingly.

A point or a pixel on the horizontal axis in the graph represents a time frame known as *Slice* and the value on the vertical axis is the total number of events within that *Slice*. More information of the *Slice* is shown in the top right corner of the view, including the timestamp and the index of the first message in the *Slice*.

The **Zooming Range Selection** (the semi-transparent yellow box on the **Full View**) represents the portion of the full trace that the **Zoomed View** is showing.

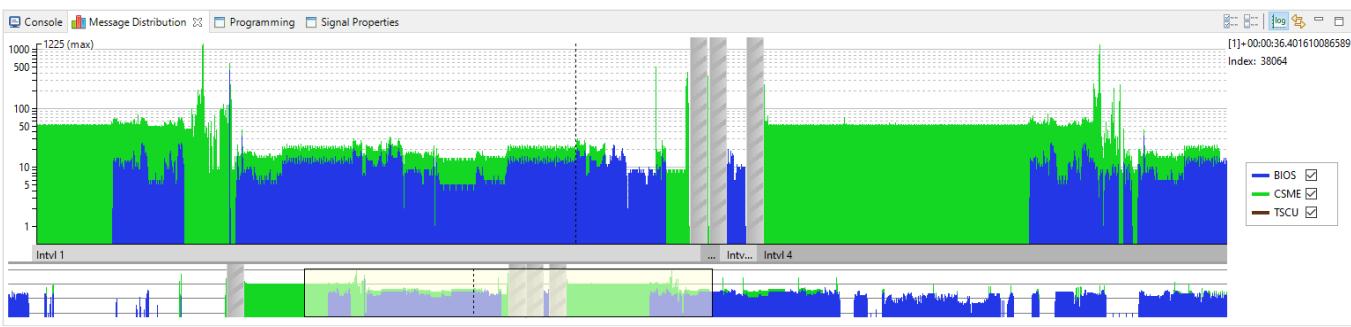
To select a region of the trace to zoom into, left click on the intended starting point in the **Full View** and drag to the intended end point.



You can update the **Zooming Range Selection** using the following controls:

Select Zooming Range	left mouse click on the Full View and drag to left or right
Resize Zooming Range	left mouse click on the left or right edge of the Zooming Range Selection and drag
Zoom In	scroll wheel up up arrow key
Zoom Out	scroll wheel down down arrow key
Pan Left	left mouse click in the Zooming Range Selection and drag to left middle mouse click on the Full View and drag to left left arrow key
Pan Right	left mouse click in the Zooming Range Selection and drag to right middle mouse click on the Full View and drag to right right arrow key

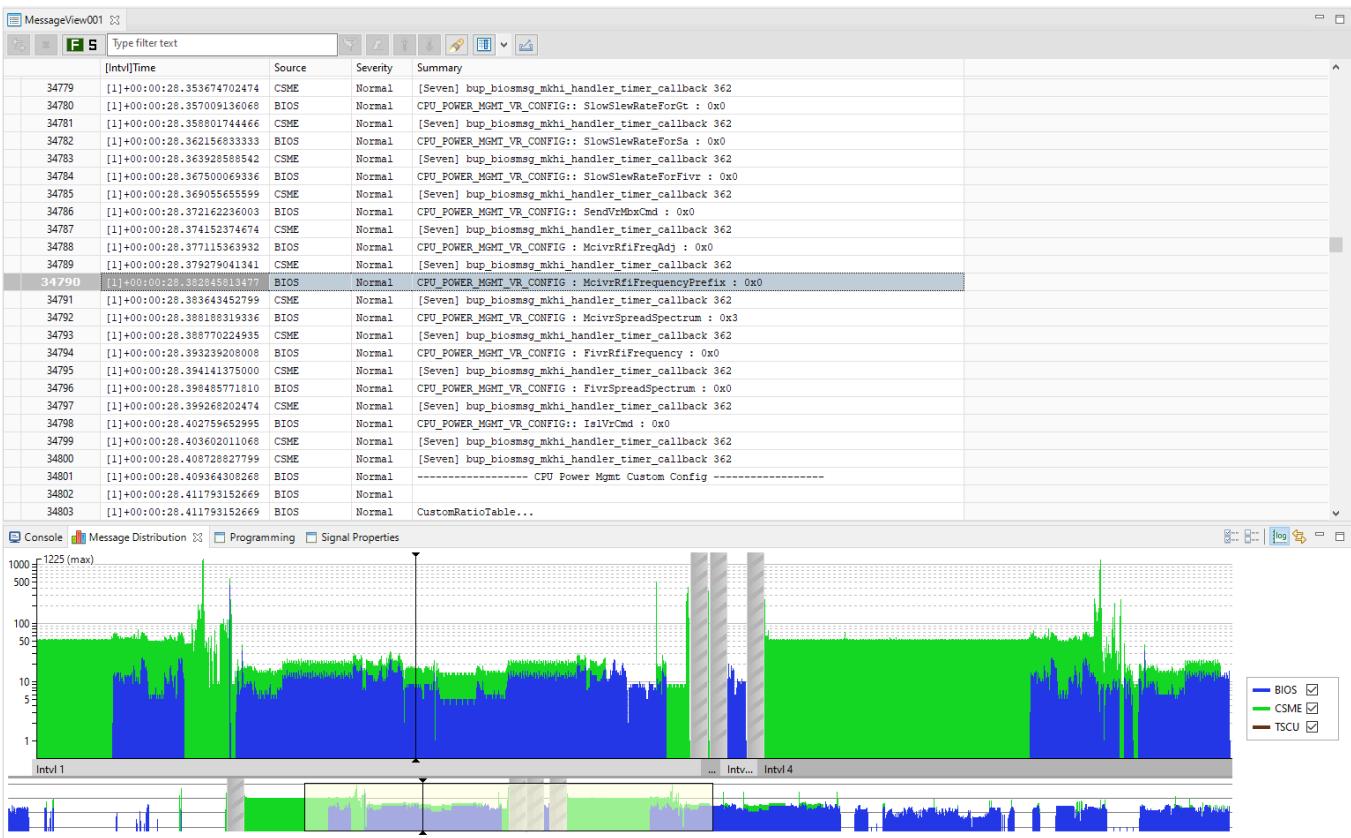
To see more information for a particular *Slice*, hover the mouse pointer over the **Zoomed View** or the **Full View**.



The **Cursor** (the dotted line) is displayed at where the mouse pointer is hovering.

- If the mouse pointer is hovering in the **Zoomed View**, a **Cursor** is always shown at the corresponding *Slice* in the **Full View**.
- If the mouse pointer is hovering in the **Full View**, a **Cursor** is shown at the corresponding *Slice* in the **Zoomed View** only if it is within the selected range.

To select the first message of a *Slice* in the **Trace Message** view, double click on the *Slice* in the **Zoomed View** or the **Full View**.

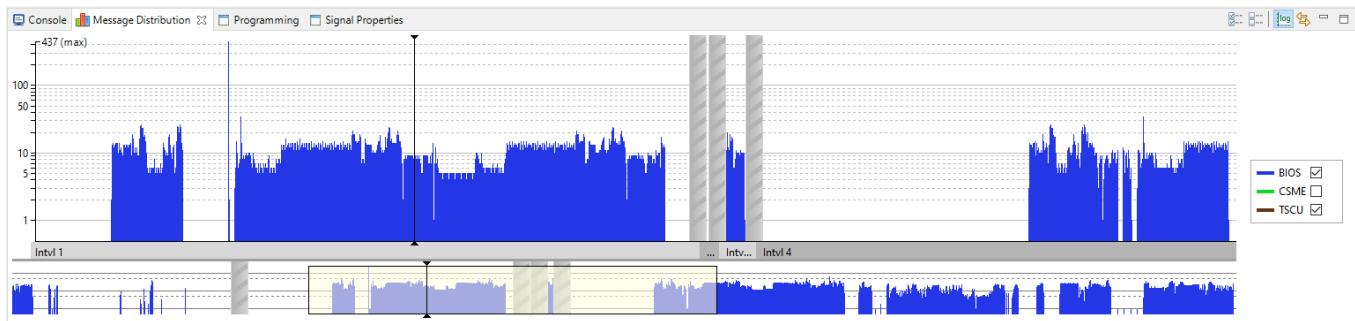


The **Marker** is displayed in the **Full View**, and the **Zoomed View** if applicable.

The next nearest message from the point marked in the graphs is selected in the **Trace Message** view.

On the other hand, selecting a message in the **Trace Message** view positions the **Marker** in the **Message Distribution** view on the **Slice** that contains the selected message.

To show or hide a source in the graphs, set or clear the checkbox for the source in the **Source Selection**.



In this example, the source that is overwhelming the distribution data of the other sources, is hidden in the graphs. This allows you to study the other sources more effectively.

The **Cursor** and the **Marker** ignore the hidden sources. For instance, setting the **Marker** in the **Message Distribution** view selects the next nearest message of visible sources only in the **Trace Message** view.

In the toolbar, you can find four buttons for settings:

- **Select All Sources**: Show all available sources in the graphs.
- **Deselect All Sources**: Hide all available sources in the graphs.
- **Show Logarithmic Scale**: Display the graphs in logarithmic or linear scale.
- **Disable/Enable Sync**: Disable or enable synchronization between the **Message Distribution** view and other **Timeline Viewers**.

Filter or Search

See this chapter for basic and advanced instructions for filtering trace data.

Quick Filter Or Search



To quickly find the events you are interested in, type a text matching criteria into the Quick Filter/Search text box (see above) in the Message View toolbar and press Enter.

The Message View will display only the trace messages that contain the filter text. See an example below:

MessageView001

	[Intvl]Time	Summary
8	+00:00:00.000134714154	A 64bit catalog message from sventx/example.c
9	+00:00:00.000151553423	Message with 1 parameter - p1=1
10	+00:00:00.000168392692	Message with 2 parameters - p1=1 p2=0x2
11	+00:00:00.000185231961	Message with 3 parameters - p1=1 p2=0x2 p3=0x3
12	+00:00:00.000202071230	Undefined message with ID 0x4
13	+00:00:00.000218910499	A 32bit catalog message from sventx/example.c
15	+00:00:00.000252589038	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6
16	+00:00:00.000269428307	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6
17	+00:00:00.000286267576	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6

To see the matching events highlighted in the full list of the trace messages, press the **F S** Filter|Search button and activate the Search view.

	[Epoch]Time	Summary
7	+00:00:00.000117874884	main
8	+00:00:00.000134714154	A 64bit catalog message from sventx/example.c
9	+00:00:00.000151553423	Message with 1 parameter - p1=1
10	+00:00:00.000168392692	Message with 2 parameters - p1=1 p2=0x2
11	+00:00:00.000185231961	Message with 3 parameters - p1=1 p2=0x2 p3=0x3
12	+00:00:00.000202071230	Undefined message with ID 0x4
13	+00:00:00.000218910499	A 32bit catalog message from sventx/example.c
14	+00:00:00.000235749769	Many think that 42 is the meaning of life
15	+00:00:00.000252589038	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6
16	+00:00:00.000269428307	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6
17	+00:00:00.000286267576	Message with 6 parameters - p1=1 p2=0x2 p3=0x3 p4=0x4 p5=0x5 p6=0x6
18	+00:00:00.000303106815	example.c:138

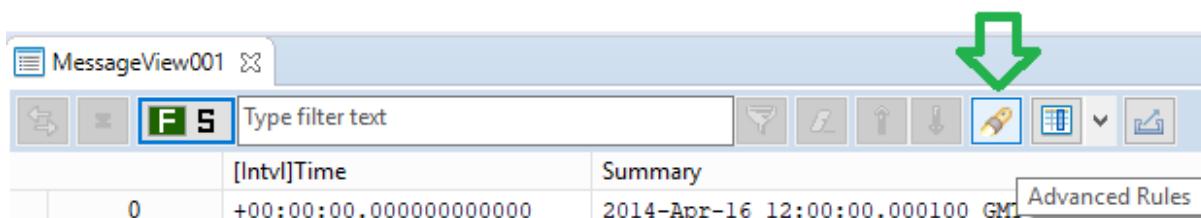
Use the Up and Down buttons to scroll search hits.

Press to Clear Filter or Search results.

Rules View

With Rules view, you can set multiple rules for highlighting messages in the Message View.

To open the Rules view, click the search icon in the Message View toolbar:



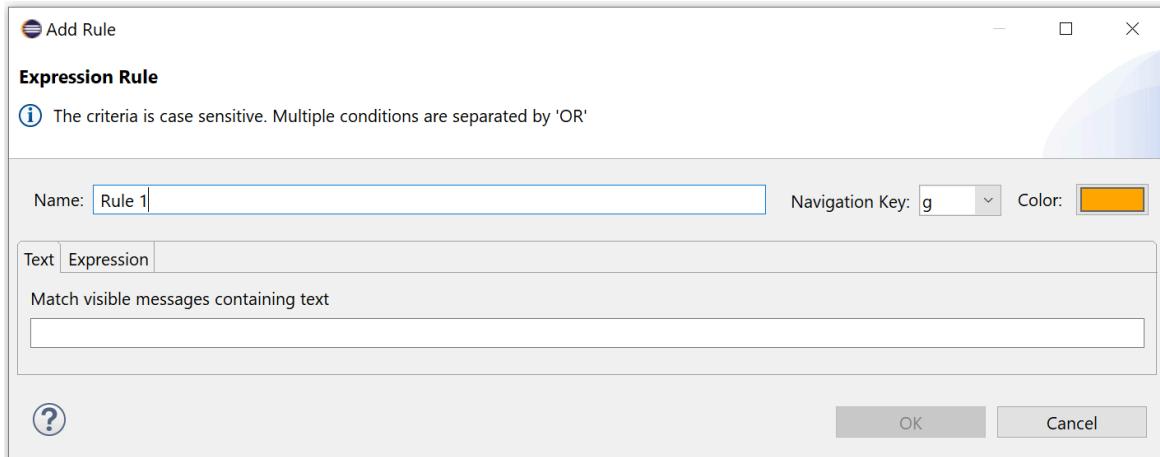
Create Rule

1. Click in the Rules view to create a new rule:

Navigation	All	Matches	Color	Name	Criteria
	<input checked="" type="checkbox"/>	-		Lost Packets	*.PacketType contains "BLOB.PACKETLOSS"
	<input checked="" type="checkbox"/>	-		Corrupted Traces	*, Status contains "ILLEGAL" Status contains "INVALID_DATA" Status contains "INVALID_TIMESTAMP" Status contains "MERGE_INCOMPLETE" Status contains "NO_MATCH" Status contains "RETRY_FAILED" Status contains "TIMEOUT"
	<input checked="" type="checkbox"/>	-		FW Error	*, Severity in ["Error", "Fatal"]
	<input checked="" type="checkbox"/>	-		FW Warning	*, Severity == "Warning"
	<input checked="" type="checkbox"/>	-		Out Of Order Packets	*, Status contains "OUT_OF_ORDER"
	<input checked="" type="checkbox"/>	-		Version	*, Summary contains "Version:"

2. In the opened dialog box, you can use two tabs to define the rule criteria:

- **Text tab** allows you to create a simple string expression. When the rule is applied, messages matching the given string are highlighted.



- **Expression tab** allows you to set more complex criteria. See how to [Set Advanced Criteria](#).

3. When criteria are defined, press **OK** to create the rule.

The rule appears in the Rules view.

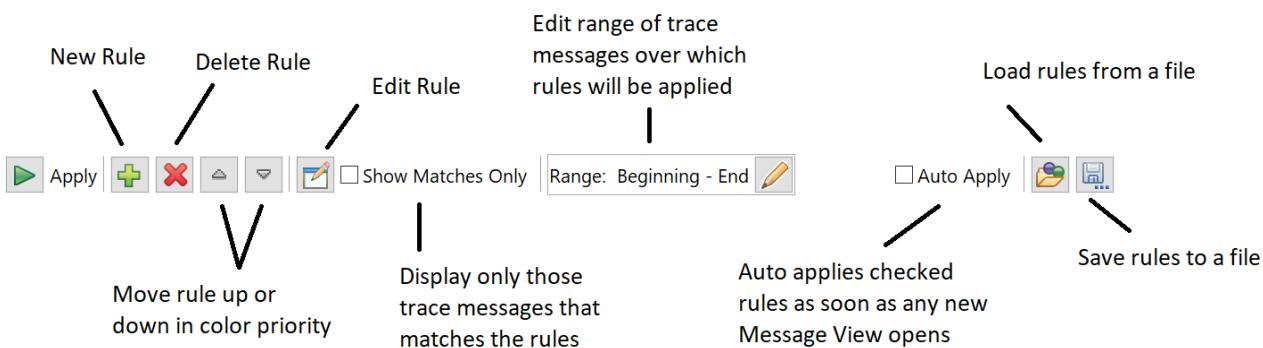
Navigation	All	Matches	Color	Name	Criteria
	<input checked="" type="checkbox"/>	-		Rule 1	* Source == "BIOS"

4. In the Rules view, you can view and edit the following columns:

- **Navigation** - Shortcut key to navigate the matches of each applied rule.
- **Checkbox** - Select the rule to apply it by clicking **Apply**.
- **Matches** - Show matches per rule when rules are applied.
- **Color** - Change the color settings of the rule.
- **Name** - Give a custom name to the rule.
- **Criteria** - The text expression of the criteria defined for this rule. To edit the rule criteria, double-click this entry.

Work with Rules View

Toolbar



Default Rules

You can start with five default rules that can help you highlight messages with Corrupted Traces, Errors, Warnings, and other common issues. These rules are shown in the Rules view when you open it. The image below illustrates the default rules with their assigned colors.

Rules					
Navigation	All	Matches	Color	Name	Criteria
↳ a	<input checked="" type="checkbox"/>	-	Red	Lost Packets	*.PacketType contains "BLOB.PACKETLOSS"
↳ b	<input checked="" type="checkbox"/>	-	Dark Gray	Corrupted Traces	*.Status contains "ILLEGAL" Status contains "INVALID_DATA" Status contains "INVALID_TIMESTAMP" Status contains "MERGE_INCOMPLETE" Status contains "NO_MEMORY"
↳ c	<input checked="" type="checkbox"/>	-	Red	FW Error	*.Severity in {"Error", "Fatal"}
↳ d	<input checked="" type="checkbox"/>	-	Orange	FW Warning	*.Severity == "Warning"
↳ e	<input checked="" type="checkbox"/>	-	Light Gray	Out Of Order Packets	*.Status contains "OUT_OF_ORDER"
↳ f	<input checked="" type="checkbox"/>	-	Blue	Version	*.Summary contains "Version."

Apart from the existing default rules, you can create multiple rules with different settings. When ready, select the required rules (check boxes on the left) and click **Apply** to apply the rules to the Message view.

The screenshot shows the Intel(R) Trace Hub HW Tester interface with two main panes:

- Message View (Top):** Displays a list of trace messages. The messages are colored according to the rules defined in the Rules view. For example, messages 1-4 are pink (Lost Packets), messages 5-7 are orange (Corrupted Traces), message 10 is purple (FW Error), message 11 is dark purple (FW Warning), and messages 15-17 are dark blue (Out Of Order Packets).
- Rules View (Bottom):** Shows the current rules configuration. The table lists the following rules:

Navigation	All	Matches	Color	Name	Criteria
↳ c	<input checked="" type="checkbox"/>	8	Orange	Rule 3	*.Severity == "Normal"
↳ b	<input checked="" type="checkbox"/>	5	Dark Purple	Rule 2	Match messages containing "parameters"
↳ a	<input checked="" type="checkbox"/>	5	Pink	Rule 1	Match messages containing "Hello world"



Note

If rules in the Rules View are changed so that they are different from what has already been applied on the Message View, an **asterik** sign will appear on title to indicate the rules are out of sync and need to be re-applied.



Color Priority

You can change the color priority of the rule using the up and down arrows in the toolbar. If several rules match the same result, the trace message is highlighted with the color of the rule which is higher in the table.

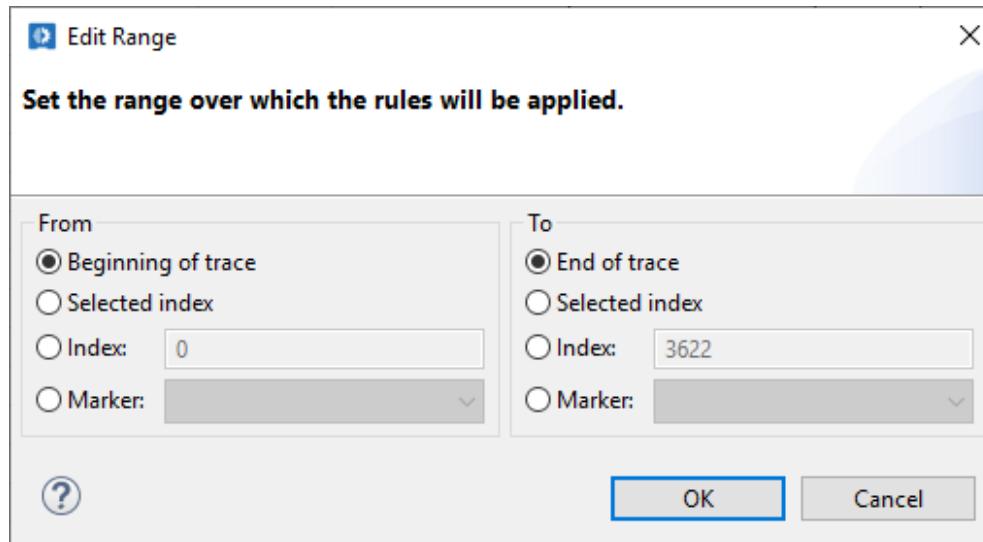
Show Matches Only

If this checkbox is selected when applying the rules, the Message View displays only those trace messages that match any of the selected rules and hides the rest of the messages.

Range

You can select the range of trace messages that the rules will be applied to. It is useful when you want to analyze a small segment of a large trace. Additionally, short ranges are filtered and displayed faster.

To limit the search to a certain window, click **Edit** icon and choose the upper bound and lower-bound using the “From” or “To” options.



If you have defined markers, they are displayed in the Marker drop-down lists and you can use them as boundaries.

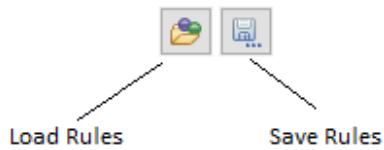
Auto Apply

You can check the Auto Apply checkbox to apply the default rules automatically (without explicitly clicking on the **Apply** button) when a trace file is decoded.

Auto Apply |

Save and Load Rules

You can use the toolbar buttons to save rules to a file or load them into the Rules view.



Set Advanced Criteria

In the **Criteria** dialog, you can use the **Expression** tab to create more complex rules with specific message fields.

Add Rule

Expression Rule

The criteria is case sensitive. Multiple conditions are separated by 'OR'

Name: Rule 7 Navigation Key: g Color: Orange

Text Expression

Remove Condition

OR

Add Condition Clear All

OK Cancel

Constraints

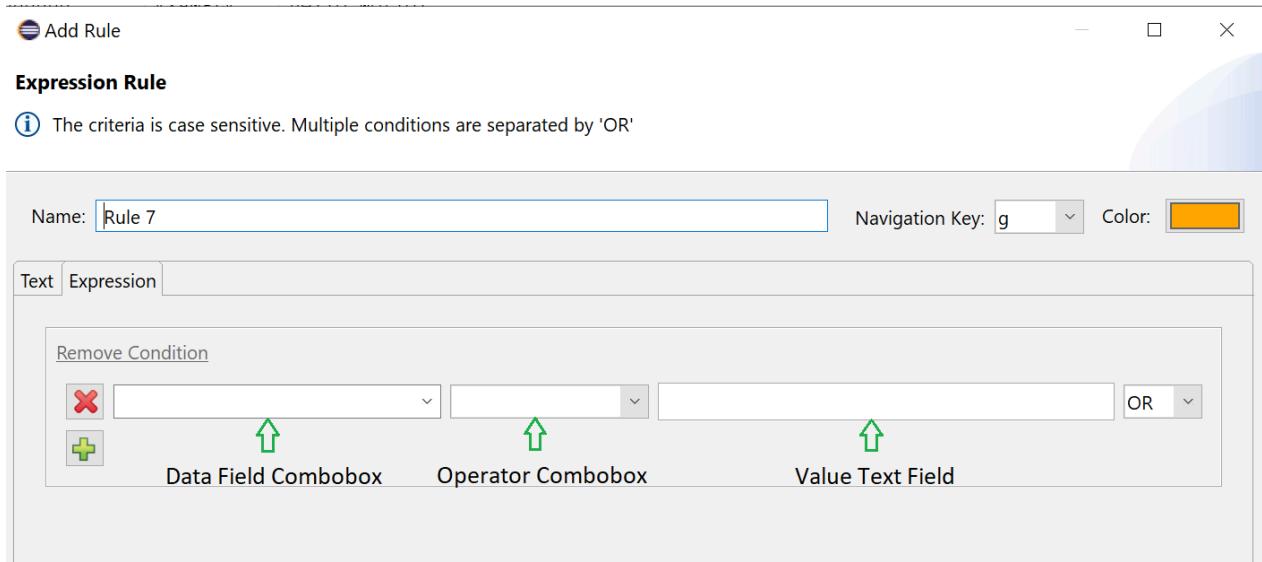
Packet Type Constraints

To select a packet type constraint:

- Click the data field drop-down and select **PacketType**.
- Click the operator drop-down and select an operator.

Commonly, the == operator is used for packet types but any other operator is also valid.

- Click in the text field to select the desired packet string.



To select multiple packet types, use logical **OR** to combine them. Logical **AND** is not supported for this purpose.

To denote all packets, do not specify any packet type constraints.

Field Constraints

- a. Click the field constraints drop-down menu and select the desired field.

Fields that belong to at least one of the selected packets are shown.

- b. Set constraints to the selected fields.

- To add a field constraint, click
- To remove a field constraint, click

If multiple field constraints are set, you can combine them by using logical **AND** or **OR**.

To add multiple conditions to your search, click **Add Condition**. An **OR** logic is applied on multiple conditions. See the last example below.

Examples

- Search for messages of the type “BLOB,” with Severity of Fatal or Error or Warning:

The screenshot shows a search condition dialog with the title "Remove Condition". It contains two conditions separated by an "AND" operator. The first condition is "PacketType == BLOB.*" and the second is "Severity IN Error, Fatal, Warning". There is also a "Remove Condition" link at the top left.

- Search for messages of all packet types, coming from the BIOS **AND** POSTCODE appears in the Summary:

The screenshot shows a search condition dialog with the title "Remove Condition". It contains two conditions separated by an "AND" operator. The first condition is "Source == BIOS" and the second is "Summary =~ ./.*POSTCODE=.*". There is also a "Remove Condition" link at the top left.

- Multiple search criteria – search for one of the following options:

- BLOB.SVEN messages where Summary contains “Destination” **AND** coming from BIOS
- BLOB.AET messages where Summary contains “dst” **OR** coming from the AET_SOC00TO

The screenshot shows two search condition dialogs. The top one is for "BLOB.SVEN" and has conditions "Summary CONTAINS Destination" and "Source == BIOS". The bottom one is for "BLOB.AET" and has conditions "Summary CONTAINS dst" and "Source == AET_SOC00TO". Both dialogs have an "OR" dropdown at the end of each row.

Visualize Trace via Timeline Viewer

In many ways, visualizing trace on a timeline can be very useful for analysis. With the trace visualization framework, it provides a configurable way to present trace data on the time axis.

Overview

The trace visualization framework consists of two main components:

- **Timeline Definition** for configuring the structure and content of the timeline. The configuration is stored in a file with the extension “.timelinedefinition”.

-  **Timeline Viewer** for displaying the timeline using the trace data from the active session with the configuration stored in the **Timeline Definition** file.

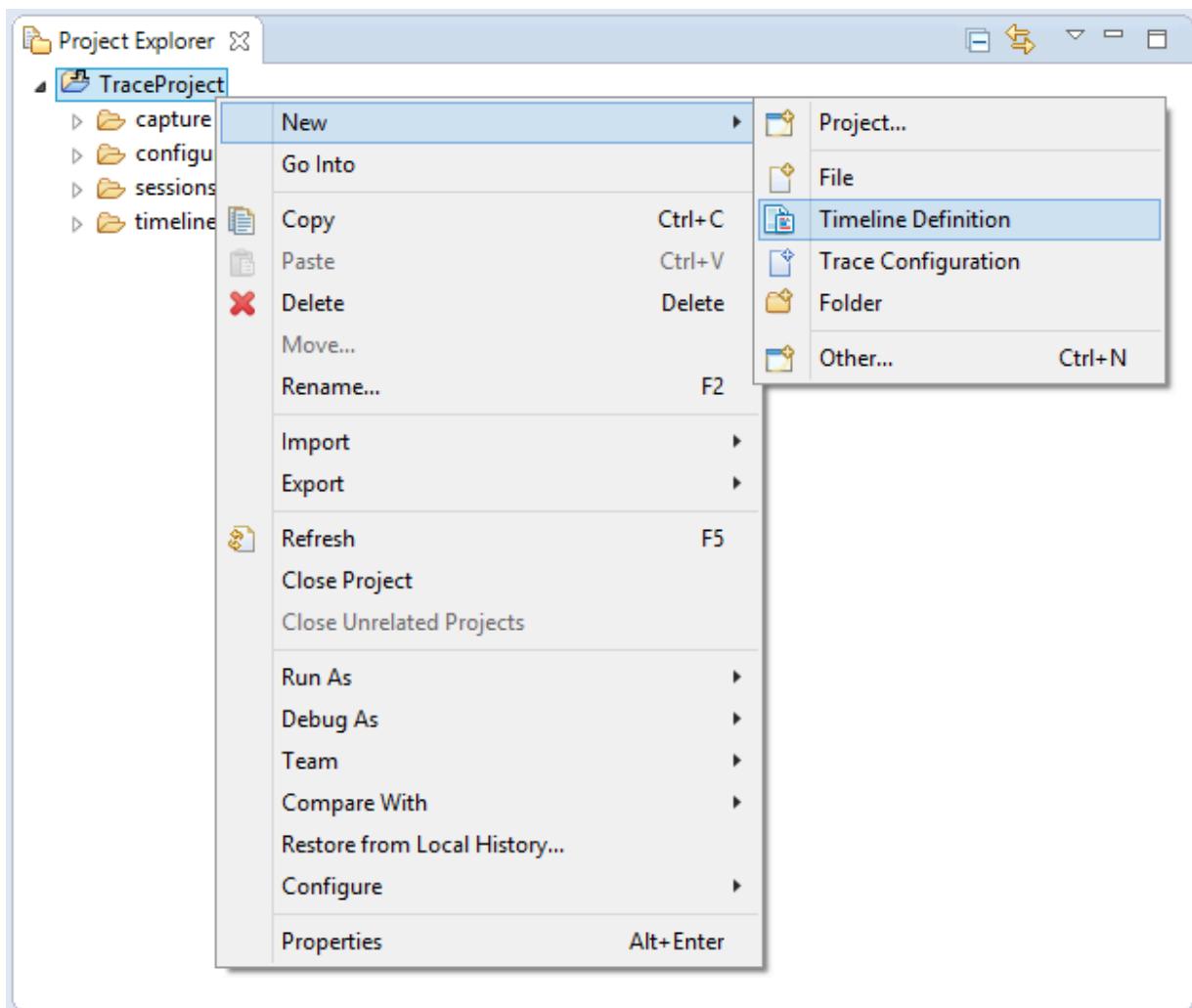
Currently, four types of visualization are supported:

-  **Message**: A simple one-to-one representation of the matching messages on the timeline.
-  **Plot**: A series of data points for a selected field plotted over time.
-  **Phase**: A visual representation of a range between two messages on the timeline, where the first message is a message that is defined as the trigger, and the second message is another message that is defined as the end of the phase.
-  **VCD (Value Change Dump)**: A visual representation of a compact waveform format. Only scalar (single bit) and vector (multi bit) signals are supported.

Create a Timeline Definition File

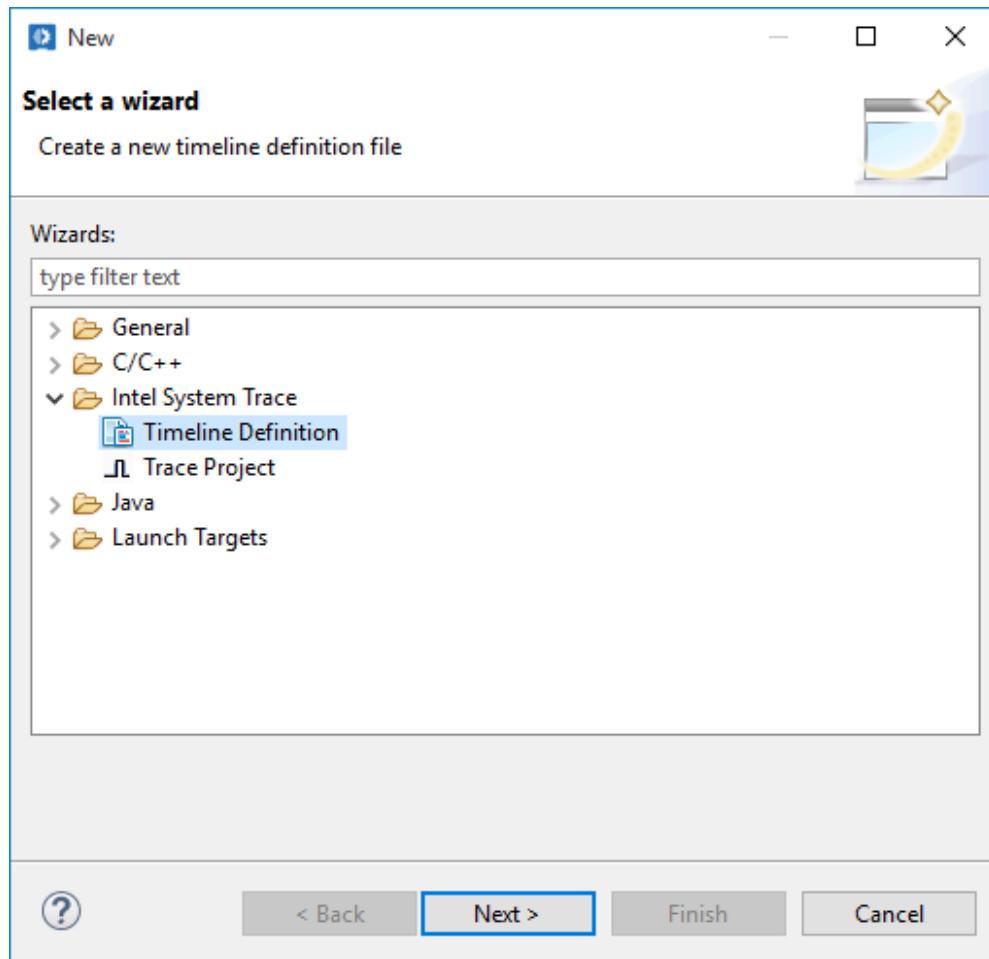
You can create an empty **Timeline Definition** file or start from a template.

1. Right click on a **Trace Project** and select **New > Timeline Definition**.

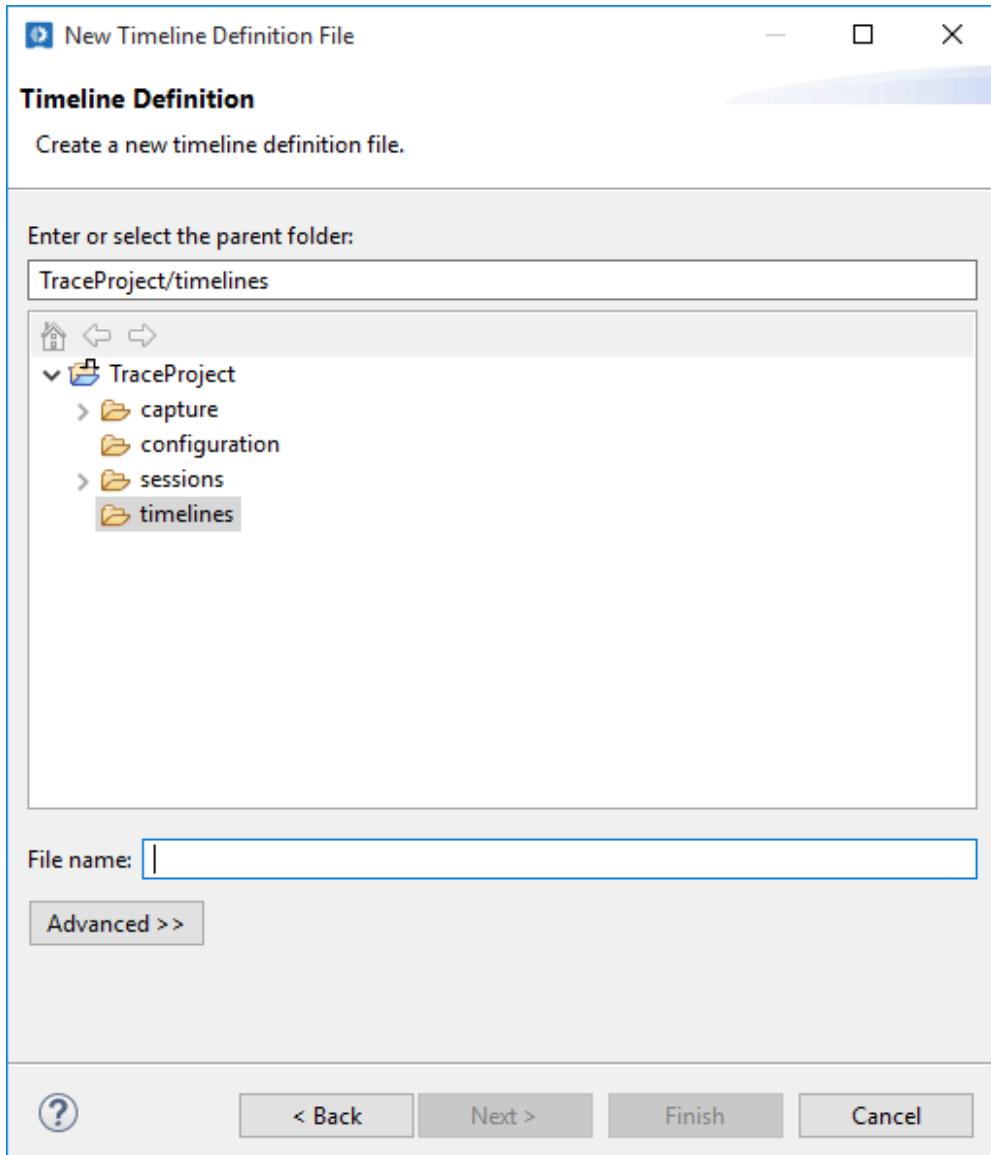


Alternatively, select **File > New > Other...** from the menu bar.

- a. The **New** wizard dialog is opened.



- b. Select **System Trace > Timeline Definition** and click **Next >**.
2. The **New Timeline Definition File** wizard dialog is opened.

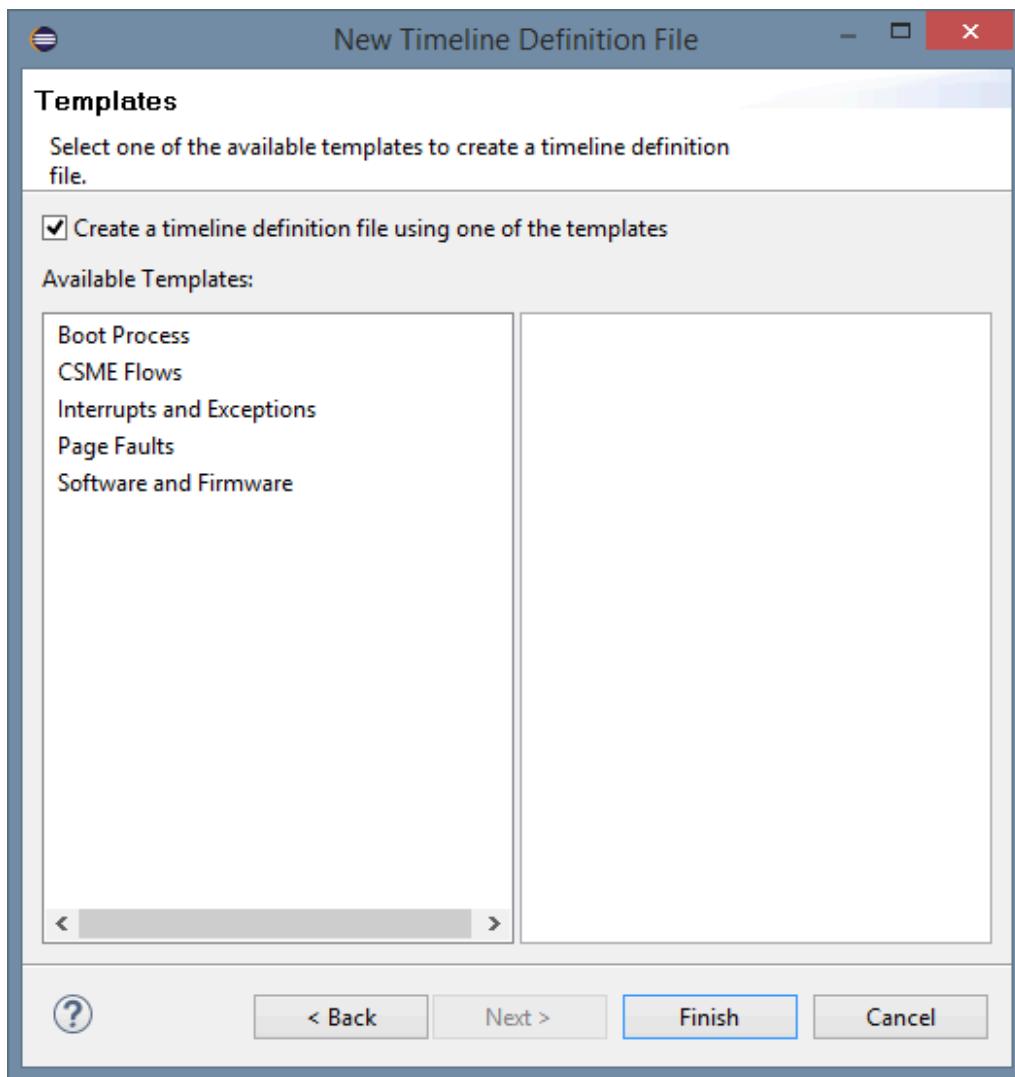


By default, the “timelines” folder is selected as the parent folder, if a **Trace Project** was selected previously. This folder is also the recommended folder for storing **Timeline Definition** files.

- a. Enter a unique file name.
- b. If you only want to create an empty **Timeline Definition** file, click **Finish** and skip Step 3.

If you want to create a **Timeline Definition** file from a template, click **Next >** and continue.

3. The template selection page is displayed.

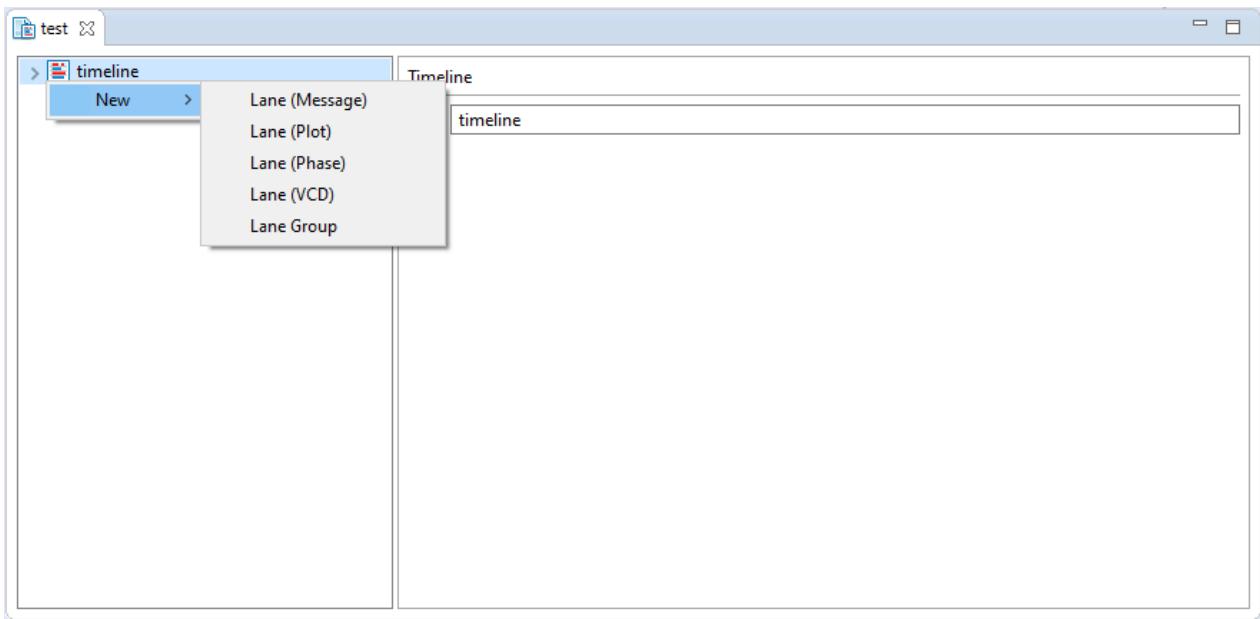


- a. Select a template from the list to see its description on the right.
- b. Select the template that you want to create the new **Timeline Definition** file with, or, uncheck the checkbox to create an empty **Timeline Definition** file.
- c. Click **Finish**.

A new **Timeline Definition** file is created and opened in a **Timeline Definition Editor**.

Edit a Timeline Definition File

1. Right click on a Timeline Definition file and select **Open With > Timeline Definition Editor**.
(Timeline Viewer is selected by default)
2. The Timeline Definition file is now opened in a Timeline Definition Editor.



A **Timeline Definition** always has the **Timeline** as the root node.

You can provide a title for the timeline.

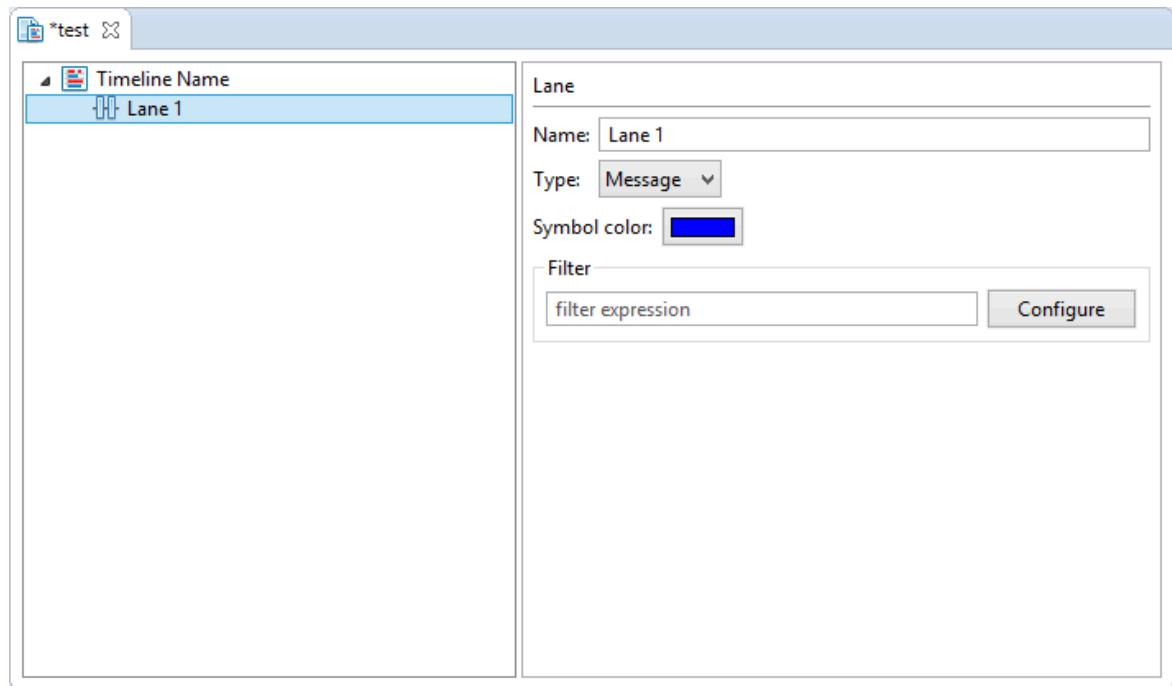
You can select **Lanes** to be added to the **Timeline** by right-clicking the **Timeline** node and selecting **New**. You can also choose **Lane Group**, which contains other **Lanes** and **Lane Groups**.

To remove the **Lane** from the Timeline, right-click the **Lane** and select **Delete**.

3. Follow the bullet points below to edit each of the **Lanes**.

- Right click the **Timeline** node and select **New > Lane (Message)**.

A new **Lane** of type **Message** is created.



The name and the symbol color can be edited.

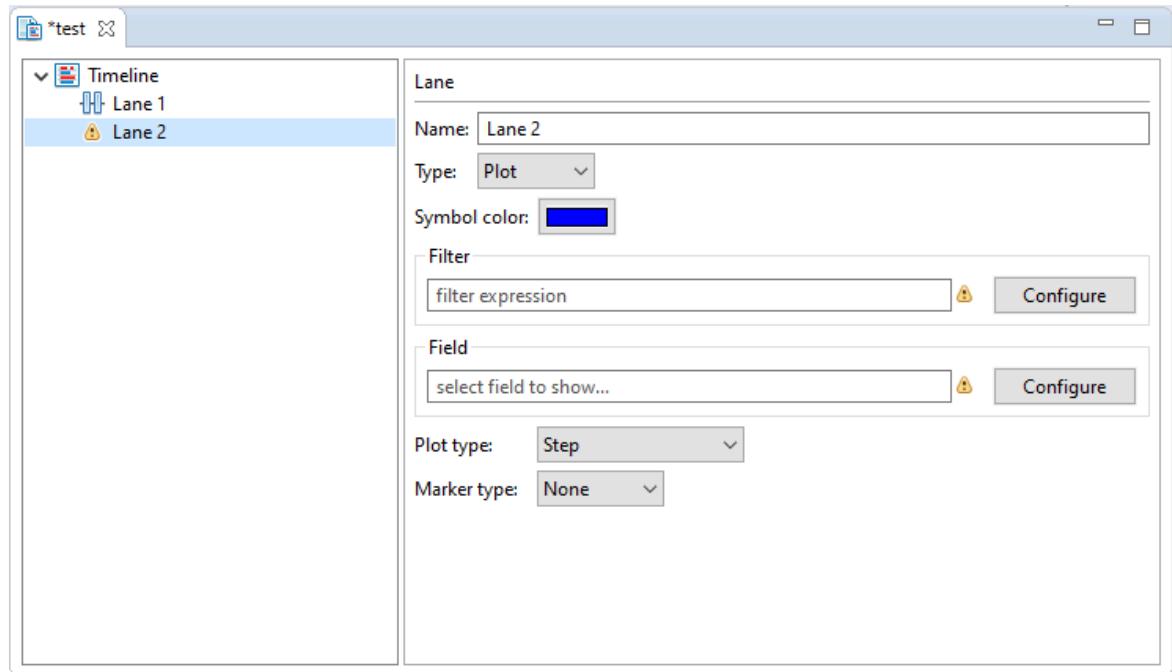
You can switch the **Lane** type to **Phase** and vice versa, which updates the editor accordingly. However, the progress for the previous **Lane** type will be lost once another **Lane** is selected.

You can click the **Configure** button to use a graphical interface for configuring the filter expression without learning the TRAM language. To use this configuration interface, the metadata tree is required. Thus, open any session with the necessary metadata before clicking the button.

Alternatively, if you are familiar with the TRAM language, you can edit the expression in the text box directly.

- Right click the **Timeline** node and select **New > Lane (Plot)**.

A new **Lane** of type **Plot** is created.



In the **Filter** section, click **Configure** and modify the filter expression in the opened dialog box. Corresponding session metadata is required.

In the **Field** section, click **Configure** and select a trace field to be displayed. Corresponding session metadata is required.

In the plot lane, the data points can be connected in one of the four different types:

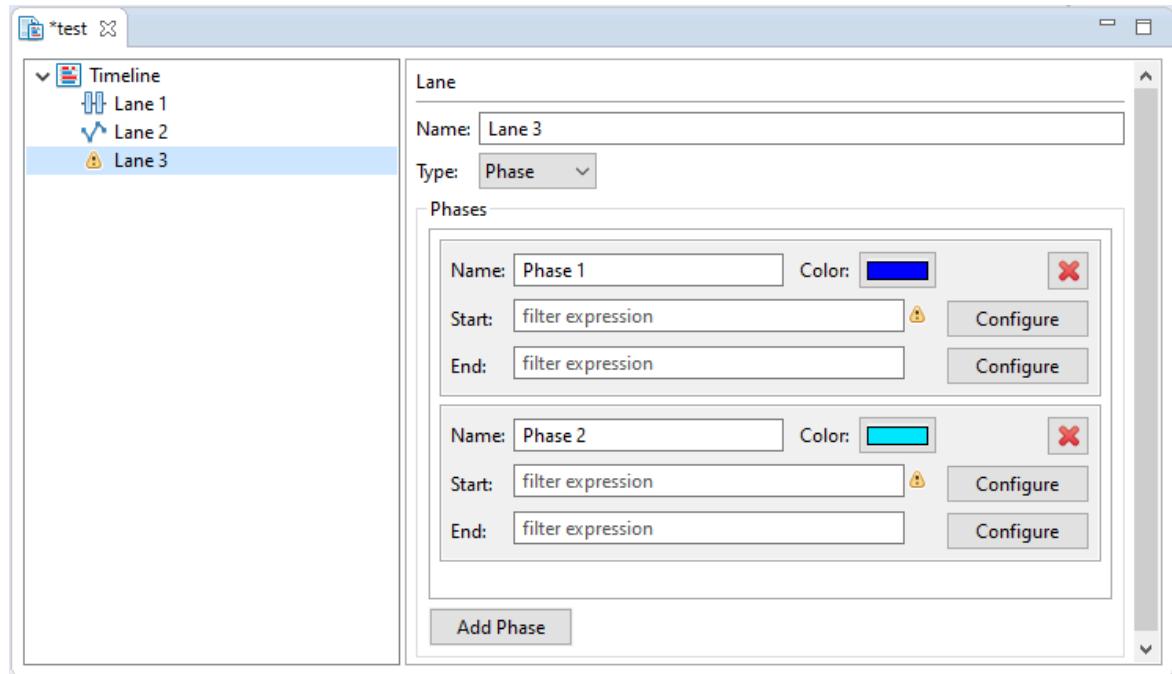
- none
- step
- linear interpolation
- spline interpolation

Optionally, select a marker type:

- none
- circle
- rectangle
- dot
- cross

- Right click the **Timeline** node and select **New > Lane (Phase)**.

A new **Lane** of type **Phase** is created.



Click **Add Phase** to add new **Phases**.

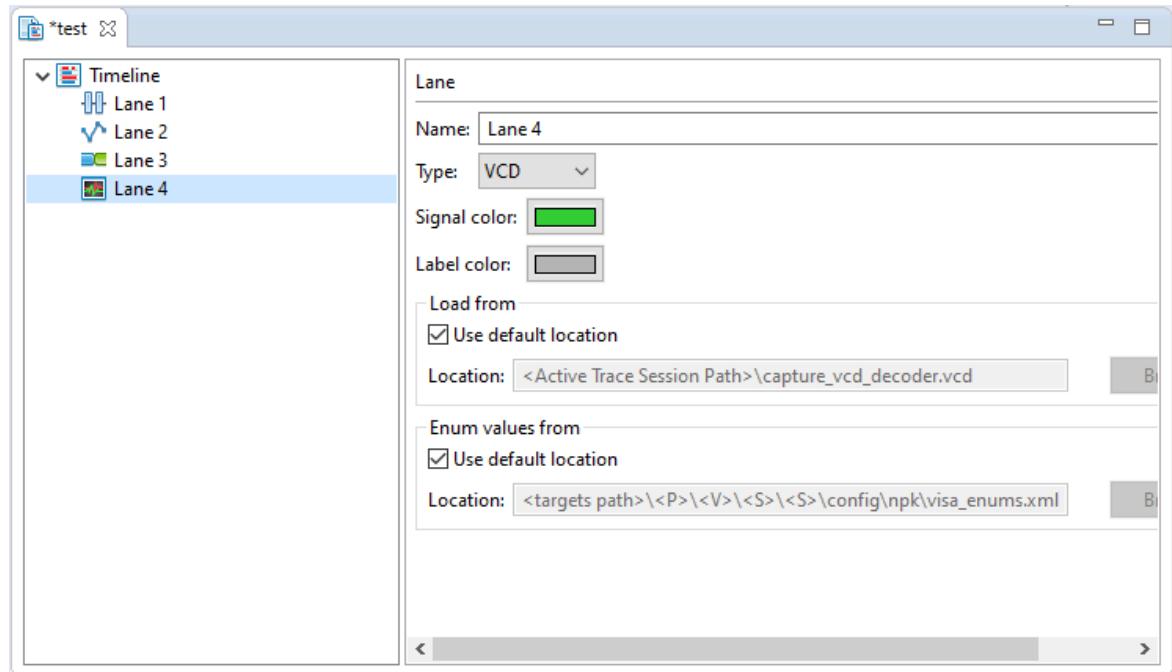
As indicated by the warning icon, the start filter is required for each **Phase**.

The end filter is optional: if none is defined, the **Phase** continues endlessly until another **Phase** is found.

Like for **Message** and **Plot Lane**, the **Configure** button opens a graphical interface dialog for configuring the filter expression, without typing into the textbox directly. The corresponding session metadata is required for the configuration interface to function.

- Right click the **Timeline** node and select **New > Lane (VCD)**.

A new **Lane** of type **VCD** is created.



By default, the VCD file is loaded from the **active trace session** path with the file name **capture_vcd_decoder.vcd**. Uncheck the **Use default location** box to select a custom VCD file using the **Browse** button.

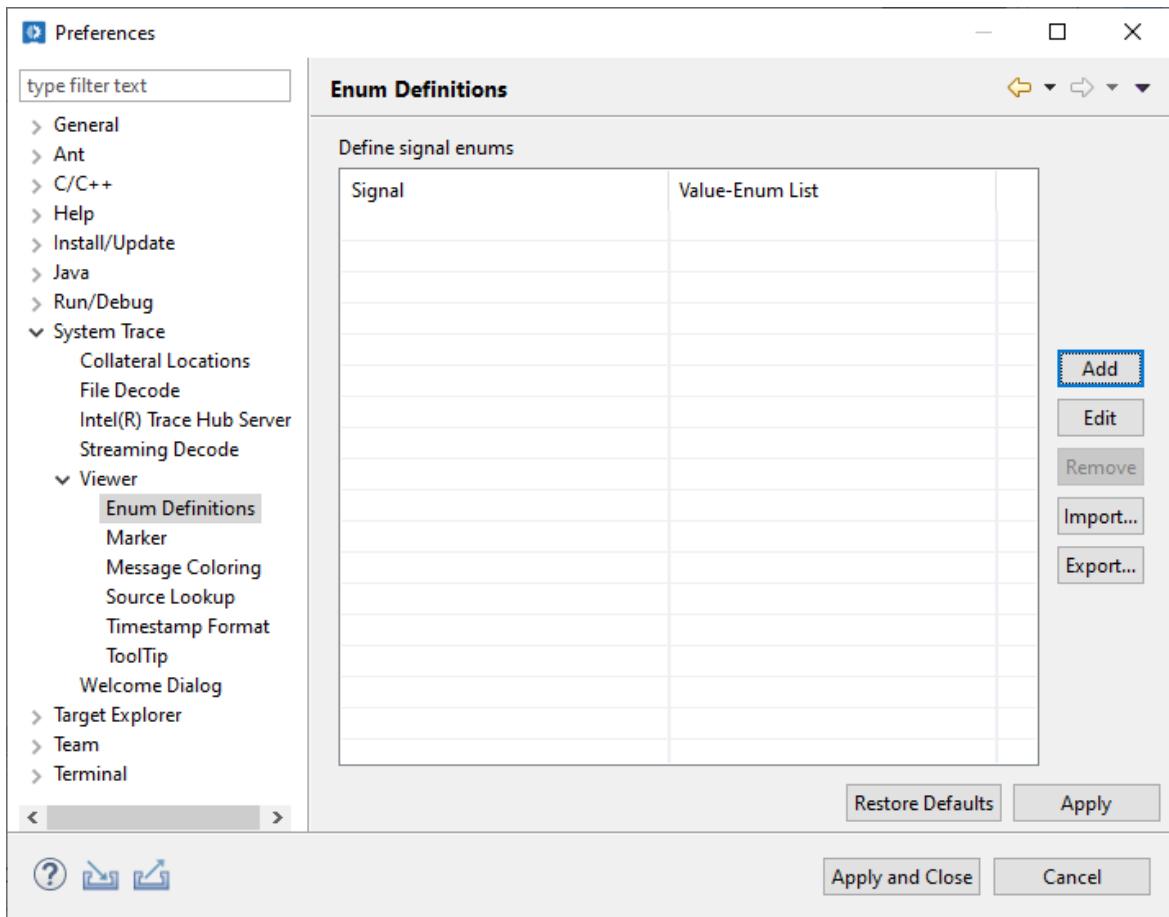
Enum value file contains a human-readable name for the corresponding hexadecimal value of each signal state in XML format. The default file is located in the collateral folders of the **active target**. Like for the VCD file, uncheck the **Use default location** box to select another XML file.

You can also define the **Enum value** for the whole workspace:

- Select **Window > Preferences** to open the **Preferences** dialog box.
- Navigate to **System Trace > Viewer > Enum Definitions**.
- **Define the enum values. You can import an XML file into the list.**

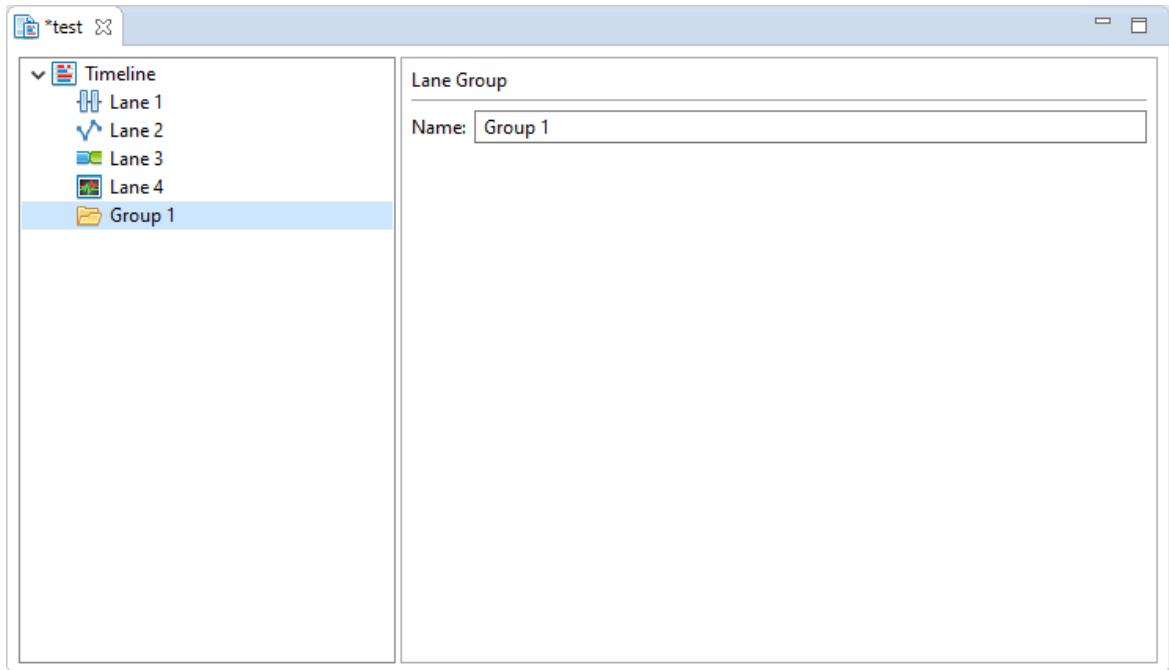
You can export the current list into an XML file for further use.

- Click **Apply** to save your changes or **Apply and Close** to save changes and close the dialog.



- Right click **Timeline** node and select **New > Lane Group**.

A new **Lane Group** is created.



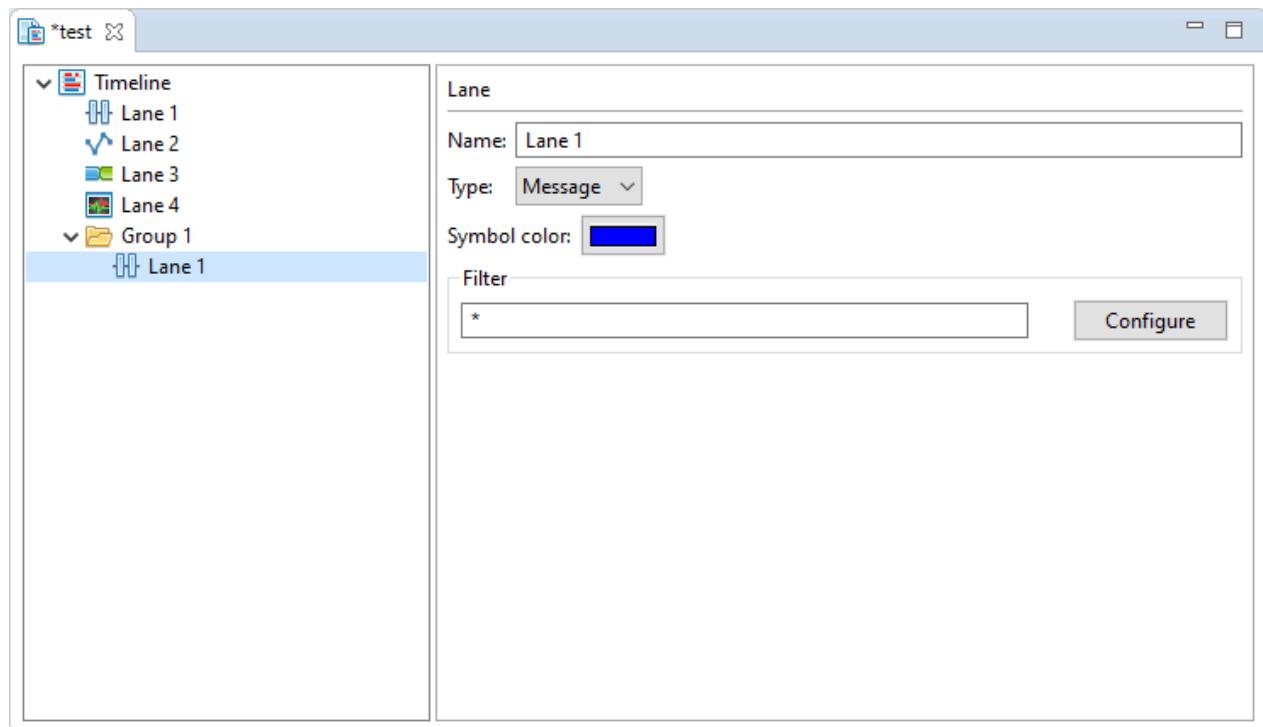
The name of the **Lane Group** can be edited.

Like the **Timeline** node, any **Lanes** and **Lane Groups** can be created in it. The difference is that the **Timeline** node cannot be removed, while a **Lane Group** can be removed by right clicking and selecting **Delete**.

4. Drag and drop is also supported for convenience.

For example, drag **Lane 1** node to **Group 1** node to move it.

If **CTRL** key is pressed while dragging, a copy operation is executed instead, which produces the following result.



Timeline Viewer

Double click on a **Timeline Definition** file to open it in a **Timeline Viewer**, or, if the default editor has changed, right click the file and select **Open With > Timeline Viewer**.

As the **Timeline Viewer** works in conjunction with a trace session, the message "No Session Opened" is displayed if no session is opened.

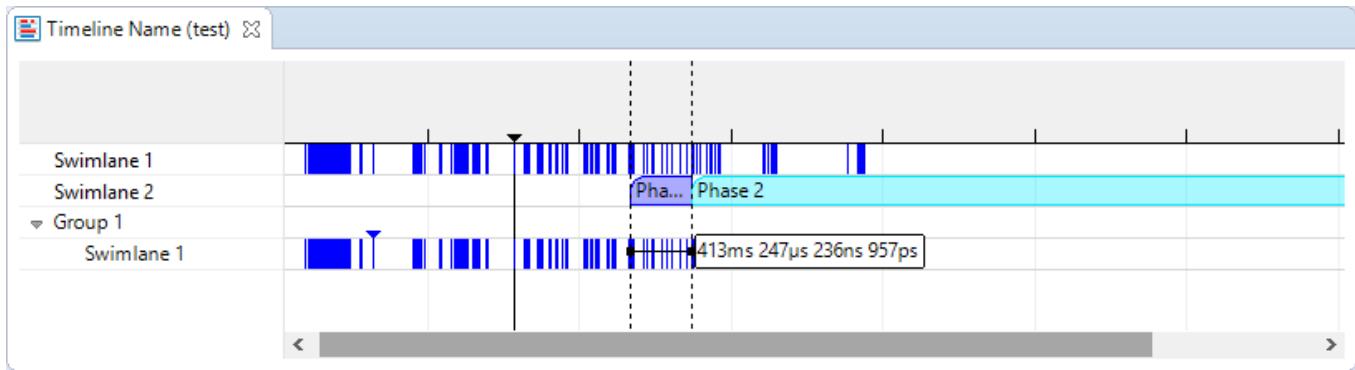
Open a session to view the corresponding visualization.



Use the mouse wheel to zoom in or out.

To pan the timeline horizontally, use the horizontal scroll bar or press the **SHIFT** key and scroll. To pan the timeline vertically, use the vertical scroll bar or press the **CTRL** key and scroll.

Additionally, the zoom range is in sync by default with the **Message Distribution** view (see [Analyzing Message Distribution](#)) and other **Timeline Viewers**. That is, updating the range selection in one view will update it for the others. To disable this behavior, open the **context menu** on Timeline Viewer and toggle the **Disable/Enable Sync** button or go to the **Message Distribution** view toolbar and toggle the **Sync Range with Timeline Viewer** button.



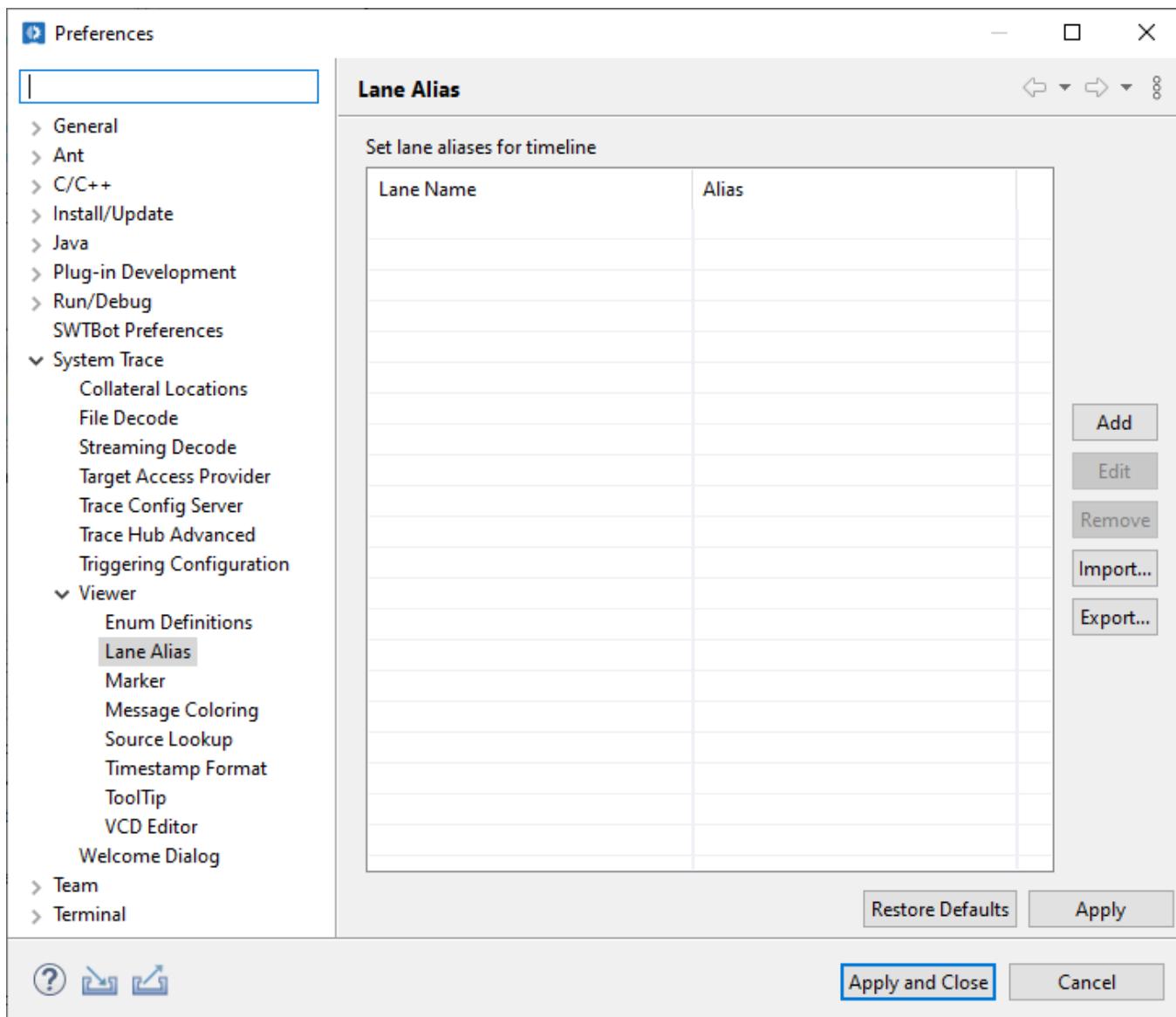
Double clicking anywhere in the timeline area activates a marker that indicates the current selection. This action also selects the closest message in the **MessageView** and set the corresponding marker in the **Message Distribution** view.

Dragging the mouse cursor anywhere in the timeline area activates the **quick measurement** tool that displays the time difference between two points on the time axis. To remove the quick measurement, right click any place within the timeline area again.

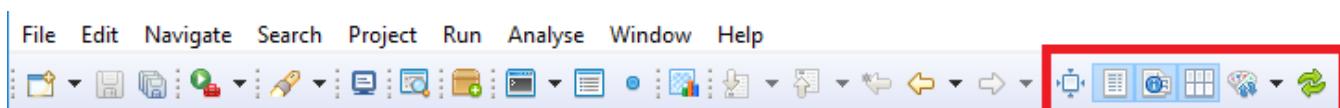
In addition to this, when a mouse is hovered over a message in the timeline, an arrow indicator with the same color is displayed. Working in conjunction with the marker and quick measurement tool described above, the indicator indicates that the timestamp of the first hovered message is used, instead of the extrapolated timestamp of the hovered pixel. This is useful for selecting and measuring timestamp with the exact precision.

You can assign a new name for a specific lane of a VCD or Timeline Definition File. Set an alias to a lane by right-clicking the specific lane and selecting **Set Alias**. You can also set the alias by the following steps below:

1. Select **Window > Preferences** to open the **Preferences** dialog box.
2. Navigate to **System Trace > Viewer > Lane Alias**.
3. Define the alias for the lane. You can import a JSON file into the list and export the current list into a JSON file for further use.
4. Click **Apply** to save your changes or **Apply and Close** to save changes and close the dialog.



In the editor toolbar, you can find six buttons for quick settings. The functionalities of these buttons are explained below.



To open the context menu, right-click the **Timeline Viewer** with the following options:

- **Scale To Fit:** Scale the resolution to fit the whole timeline range within the visible timeline area. This button is also available in the editor toolbar.
- **Show/Hide Value:** Show the value of each lane, if applicable, at the timeline marker. This button is also available in the editor toolbar.
- **Show/Hide Details:** Show the details box with timestamp details when the mouse is hovered over a message in a message lane. The details box contains information about timestamp, source, and catalog string. This button is also available in the editor toolbar.
- **Suspend/Resume Details:** “Suspend Details” to keep the selected details information shown. “Resume Details” to continue updating details box with current mouse position.

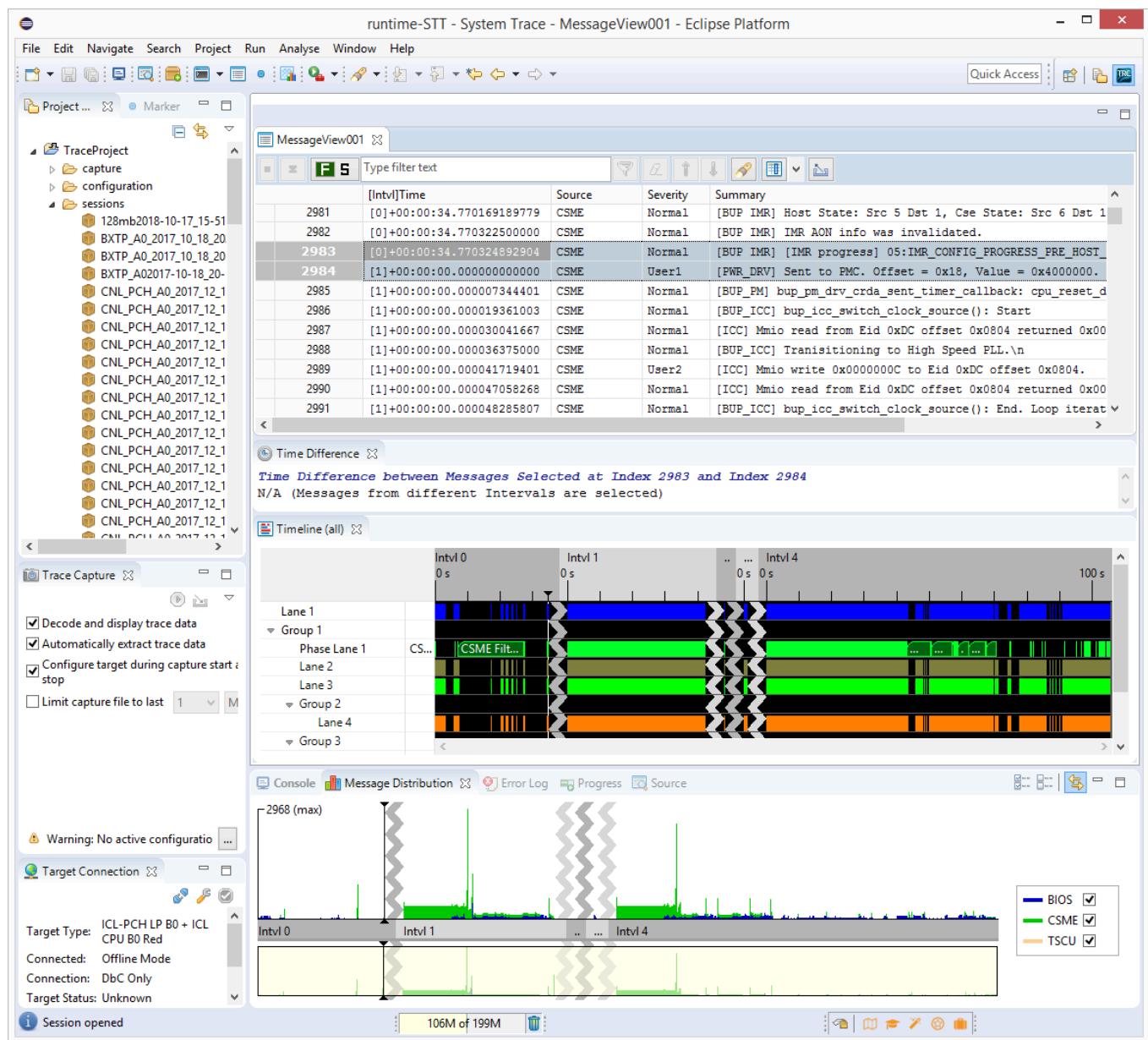
-  **Show/Hide Grid:** Show or hide grid lines. This button is also available in the editor toolbar.
-  **Change Theme:** Choose between difference themes. The main difference is the background of the timeline area having a default or darker color. This button is also available in the editor toolbar.
-  **Change Color:** Open a dialog box to modify the color of a lane.
-  **Set Alias:** Open a dialog box to modify the displayed name of a lane.
-  **Hide Lane:** Hide or show lanes
 - **Hide This:** Hide the lane that the mouse is hovered over.
 - **Unhide All:** Show all hidden lanes.
-  **Reload:** Recompute the timeline data. If the content of the Timeline Definition file has changed, clicking this button will show the timeline with the updated configuration. This button is also available in the editor toolbar.
-  **Search:** Search for a certain parameter in the selected lane(s).
 - **Select:** Select the lane to search for.
 - **Customize:** Set the search parameters for each selected lane.
 - **Select All:** Add all lanes to the search selection.
 - **Deselect All:** Clear the search selection.
 - **Find Next Match:** Find the next match in the selected lane(s).
 - **Find Previous Match:** Find the previous match in the selected lane(s).
-  **Go To:** Show timeline marker at the specified timestamp, message index, next trigger, available marker, beginning or end of interval:
 - **Start:** Go to the beginning of the interval until it reaches the beginning of timeline.
 - **End:** Go to the end of the interval until it reaches the end of timeline.
 - **Timestamp:** Open a graphical interface to go to the specified timestamp.
 - **Message Index:** Open a graphical interface to go to the specified message index.
 - **Trigger:** Go to the next available trigger.
 - **Marker:** Open a graphical interface to go to the available marker.
 - **Mouse Cursor:** Go to the current position of the mouse cursor.
-  **Measure Time Difference:** Open a graphical interface dialog box to measure the time difference between two points of interest, including: timestamp, message index, trigger, marker, beginning or end of interval.
-  **Marker:** Mark the timestamp of trace entries for later use. (See 6.9 Basic Trace Analysis: Marking Trace Entries)
 - **Add:** Open a graphical interface dialog box to specify marker parameters such as name and color. Add the timestamp that the mouse is hovered over to the marker list.
 - **Quick Add:** Quickly add timestamp to the marker list without specifying the parameters.
 - **Go To:** Open a graphical interface to go to the available marker.
 - **Marker Preferences:** Open a graphical interface to set advanced marker parameters.

- **Disable/Enable Sync:** Disable or enable synchronization between the **Message Distribution** view and other **Timeline Viewers**.
- **Zoom In:** Zoom in the timeline. You can also use the mouse wheel.
- **Zoom Out:** Zoom out the timeline. You can also use the mouse wheel.
- **Zoom Into:** Zoom into the selected range shown within the quick measurement tool.

Analyze Discontinuous Trace

Trace is organized chronologically by nature. However, hard reset of the system might distort the timestamp order. The correct chronological order of the trace is indicated by the Interval (Intvl) prefix.

The idea is simple: each time a timestamp is reset, the subsequent messages are grouped in a detached section time-wise. This section is identified as an “Interval” and labeled with an incremental number starting from zero. Therefore, the Interval can be seen as an extension of the timestamps and an integral part of the System Trace.



Visual cues:

- When a trace session contains several Intervals, timestamps are prefixed with the Interval number.
- The Message Distribution view and the Timeline Viewer show Intervals in the chronological order.
- The Message Distribution view displays the duration of each Interval.
- The Time Difference view shows the time distance between selected timestamps. As the time distance between Intervals is undetermined, you can only measure time difference between timestamps belonging to one Interval.

Advanced Trace Analysis

The features described in this section are considered advanced. If you need more help for working with them, contact Intel support.

All basic features are described in previous subsections of this chapter.

Master/Channel Filtering

Traces from the specific masters and channels can be filtered by setting a decoder parameter in **File Decode Preferences** dialog and/or in the **Streaming Decode Preferences** dialog. Filtering channels might be important if a trace source has noisy traffic, where not all information is relevant for the user.

The filtering can be enabled by adding a decoder parameter to the MIPI decoder. The decoder parameter is named **masterChannelFilter** and has the following syntax:

```
<master>:<channels-to-filter>
```

The following expressions for <master> and <channels-to-filter> are valid:

- Single number
- Range using the “-“ operator: <number-1>-<number-n>

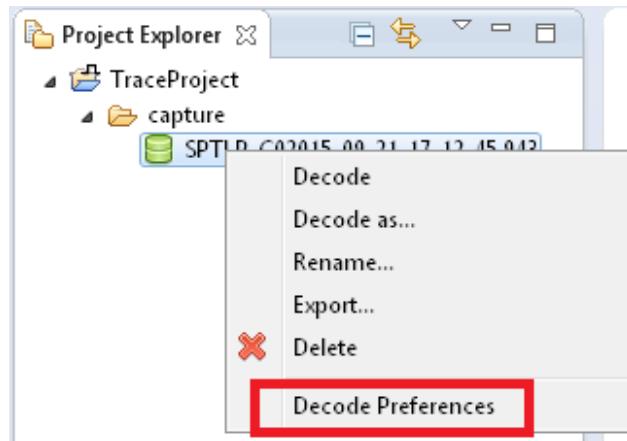
It is also allowed to combine multiple expressions by separating them with a comma, for example:

```
<master>:<channel>,<master>:<channel-1>-<channel-n>,<master-1>-<master-n>:<channel>,<master-1>-<master-n>:<channel-1>-<channel-n>
```

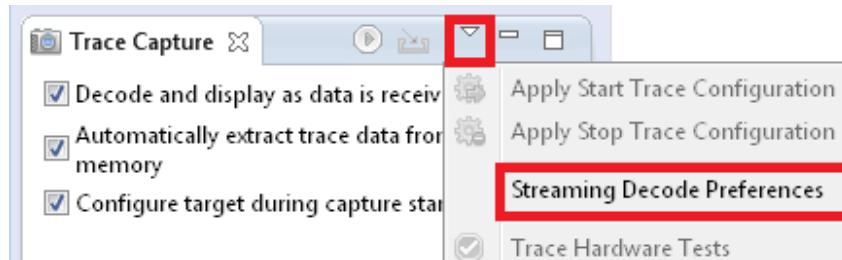
Example on master 3, which uses channels 1 to 10, filter out channels 5 to 8. To do this is to add a new **MIPI_Decoder masterChannelFilter** parameter using following steps:

1. Open File Decode Preferences or Streaming Decode Preferences depending on the streaming mode:

- **Offline decode:** Right-click on the capture file and then click on Decode Preferences



- **Live decode:** Right-click on view menu “triangle” and then click on **Streaming Decode Preferences**



2. Press the **Add** button to add a new decoder parameter

Decoder Parameters			
Decoder	Instance	Name	Value
ETWDecoder		pathToSchemaConfigFile	\$(CAPTUREDIR)/config/etwConfig/ETWConfigUpParam.xml
ETWDecoder		pathOfConfigUICopySchemaDoneFolder	\$(CAPTUREDIR)/config/etwConfig/
ETWDecoder		etwTimeCorelationScale	100000000000
ETWDecoder		etwClockName	NPK
ETWDecoder		convertToNpkTime	true

Add a following line: MIPI_Decoder | filter | 3:5-8 as shown in the picture below

Decoder Parameters			
Decoder	Instance	Name	Value
MIPI_Decoder		masterChannelFilter	3:5-8
ETWDecoder		etwTimeCorelationScale	100000000000

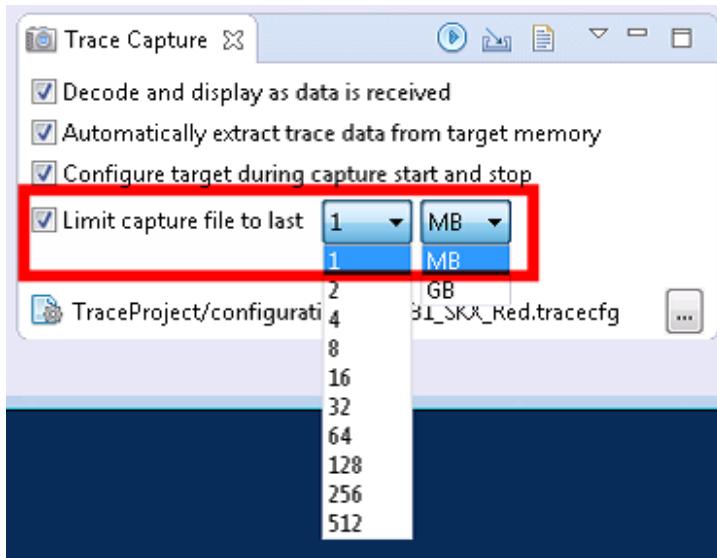
The value of the filter parameter for the MIPI decoder is 3:5-8, meaning that for master 3 channels 5, 6, 7, and 8 are filtered out and there will be no traces from these channels visible. More complex example would be 2-4:5-8, which filters on master 2 channels 5 to 8, on master 3 channels 5 to 8 and on master 4 channels 5 to 8.

! Note

Filtering on channel 0 is not supported.

Trace Capture File Size Limit

Captured file size during live decode could be limited to a specific size. This is especially important for long running live traces where gigabytes of data are received. You can choose to store on the hard drive only the last X Mbytes of the received data. Before starting capture live trace, select the desired capture file size in the **Trace Capture** view. When using a streaming destination, capture file size limit is implemented using a circular file buffer. When tracing to memory, only the specified amount of trace data is actually extracted, greatly improving extraction speed when running long traces.



Decode Part of a File

When a trace is captured, it can be afterwards additionally decoded by doing right-click on a file and then pressing "Decode" or "Decode As". This will decode the file entirely. For use cases when only a part of the decode is necessary user can use partial decode option. For example, decode only the last 20% of the file. This is useful for files of huge sizes.

Decoding only a part of the file is controlled by the FileReader parameters. There is a possibility to set the starting and the ending percentage of the file decode. For example, to skip the first 10% of the file and to decode till the last 80% of the file, you should add following decoding parameters

```
FileReader_Decoder\| \| startPositionPercentage \| 10 FileReader_Decoder\| \| endPositionPercentage  
 \| 80
```

In case of 300MBytes file, first 30MByte would be skipped and the reading would stop at MByte 240. The first 10% which is 30Mbyte and the last 20% which is 60 MByte are ignored.

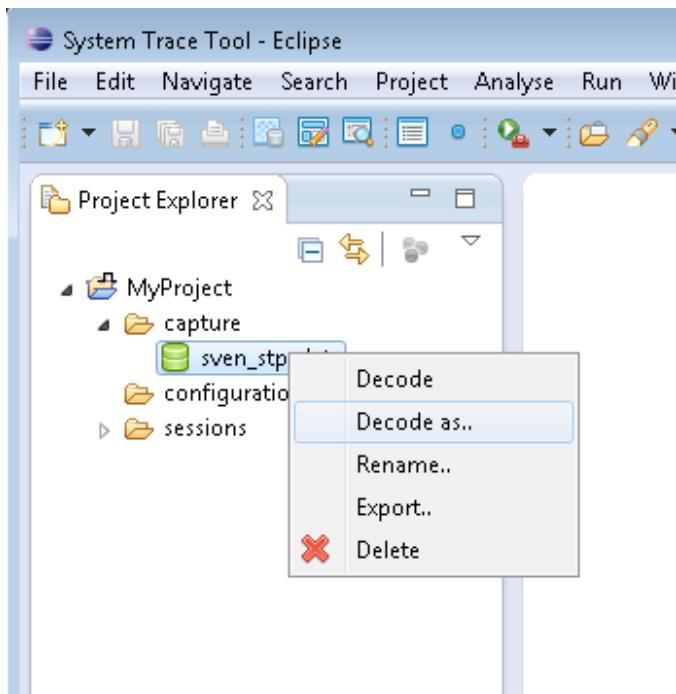
Both parameters are optional and it is possible to specify only one parameter. If for example `endPositionPercentage` is 30 and `startPositionPercentage` is not set, then only the last 30% of the file will be decoded.

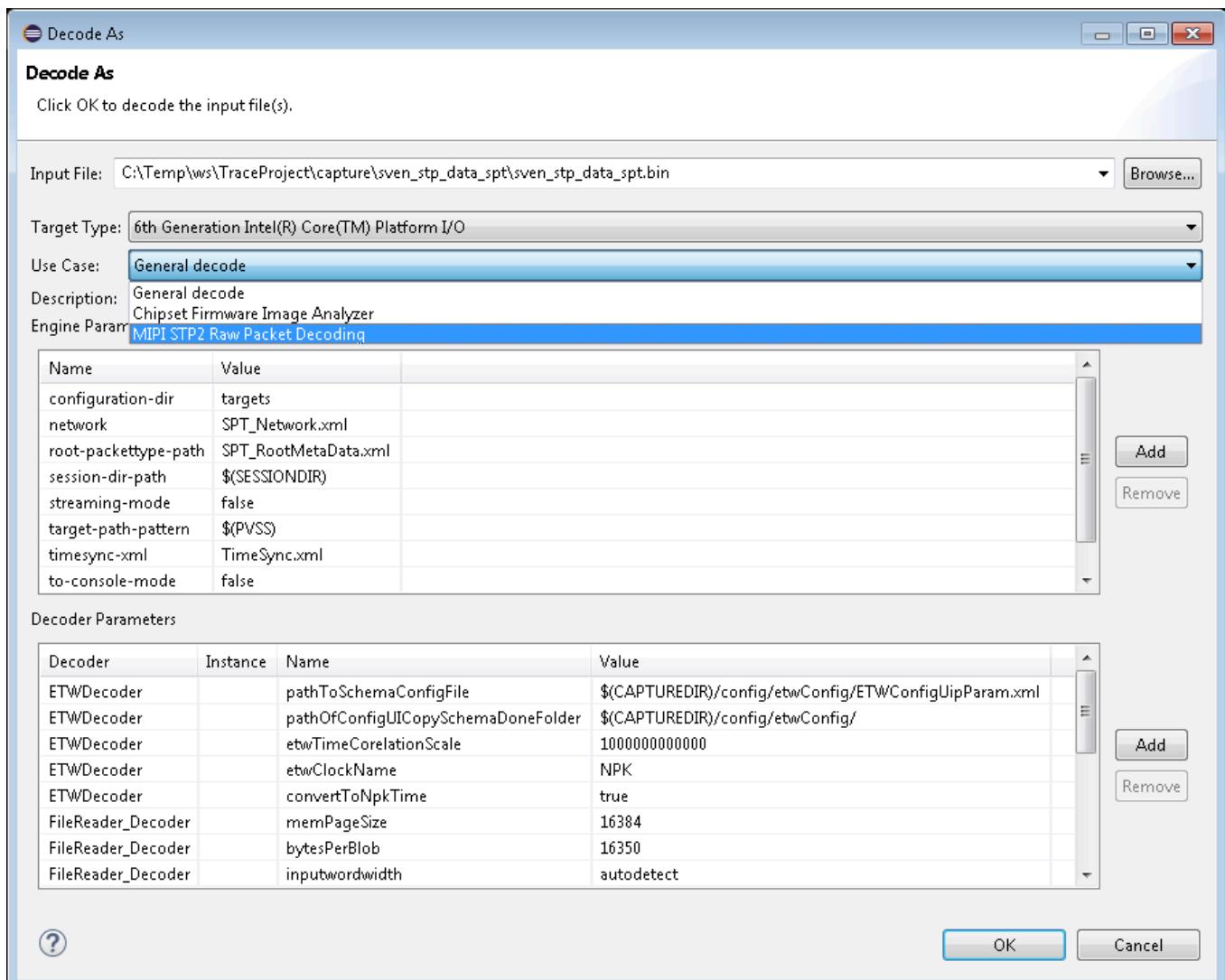
Similarly, if only `startPositionPercentage` is set, for example to 5, then the first 5% of the file are ignored.

Decode Trace with a Different Decode Network

To decode your trace with a different decode network, e.g. to decode just the raw MIPI packages, do the following:

1. In the **Project Explorer**, right-click on the captured trace file and select **Decode as...**.
2. In the next dialog, select the use-case (network) you want to use and click **OK**.





Enable Low Voltage Margining Overrides

During low power S0iX states, logic voltage is lowered (physical low power state). This causes limitations of the debug capabilities and requires a different debug interface than if the system stayed in the S0 power state. When the debug feature is using signal or power rail (which must be off for low power states), hardware blocks low power assertion.

To maintain the debug connection and allow the target to enter low power state, LVM (logical low power states) are introduced. Low Voltage Margining (LVM) overrides is a setting that disables lower voltage requirement and allows the debugger to achieve simulated (logical low power) states and keep debug connection running and active.

Note

If a platform does not have LVM setting, this configuration is not supported.

To change LVM overrides, go to the **Platform Configuration** menu and check the **Apply low power overrides** box. Trace profiles that are applicable for low power have this setting enabled by default.

The screenshot shows the System Trace configuration interface. In the 'Profiles' section, 'Intel(R) DCI USB DBC' is selected. The 'Trace Sources' section contains a list of event types: AET, CSE, BIOS, ISH, PMC, and OS and App SW. Most are checked except for AET. The 'Parameters: AET' section lists various events (Retain AET settings across CPU resets, HW Interrupt event, IRET event, Exception event, MSR event, IO event, Secure Enclave event, Write Back Invalidate event) each with LBR EN checkboxes. The 'Platform Configurations' section has a checked checkbox for 'Apply low power overrides'.

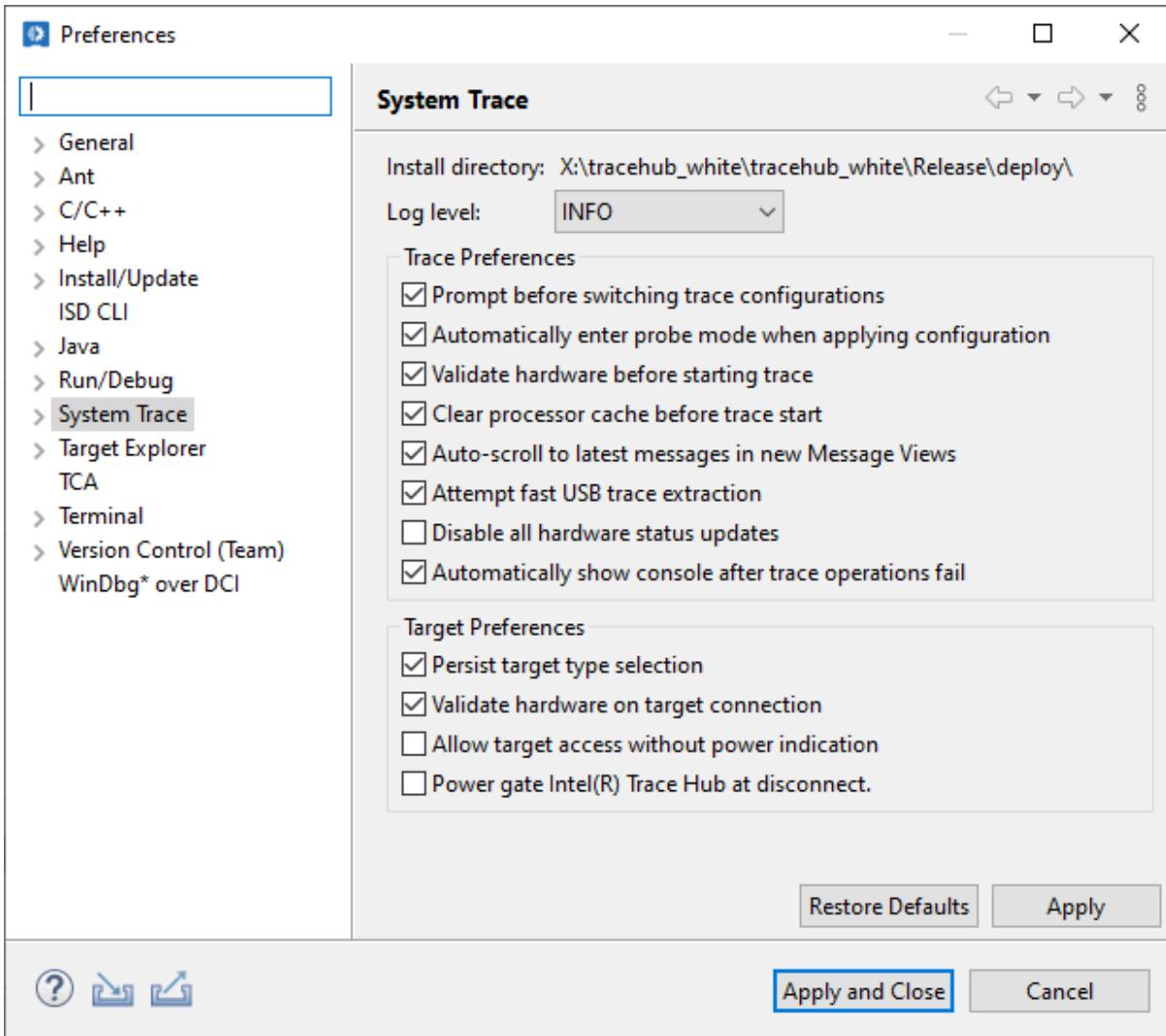
Controlling Power Management for Intel(R) Trace Hub (Intel(R) TH)

Client platforms (starting from Tiger Lake) have capability to control Intel(R) Trace Hub (Intel(R) TH) power management.

During connection, System Trace tries to detect Intel TH hardware. If the detection fails, System Trace attempts to execute Power Gate Exist request for Intel TH and repeats the hardware check.

You can set Intel TH to power gate after the tool disconnect as follows:

1. Go to the **Window** menu and select **Preferences**.
2. In the opened dialog box, navigate to System Trace and check the **Power gate Intel(R) Trace Hub at disconnect** box under Target Preferences.



3. Click **Apply** to save your changes or **Apply and Close** to save changes and close the dialog.

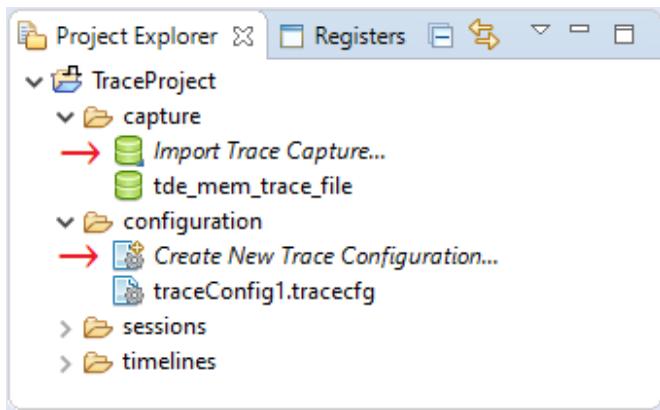
Trace Analysis: Hints

Copying and Pasting Trace Entries

After filtering your trace, you may be interested in copying and pasting the resulting trace entry set into another application. To do so, select one or more trace entries, right click on the trace entries, select **Copy** and pick the method that best matches your use-case.

Drag and Drop to Import Files

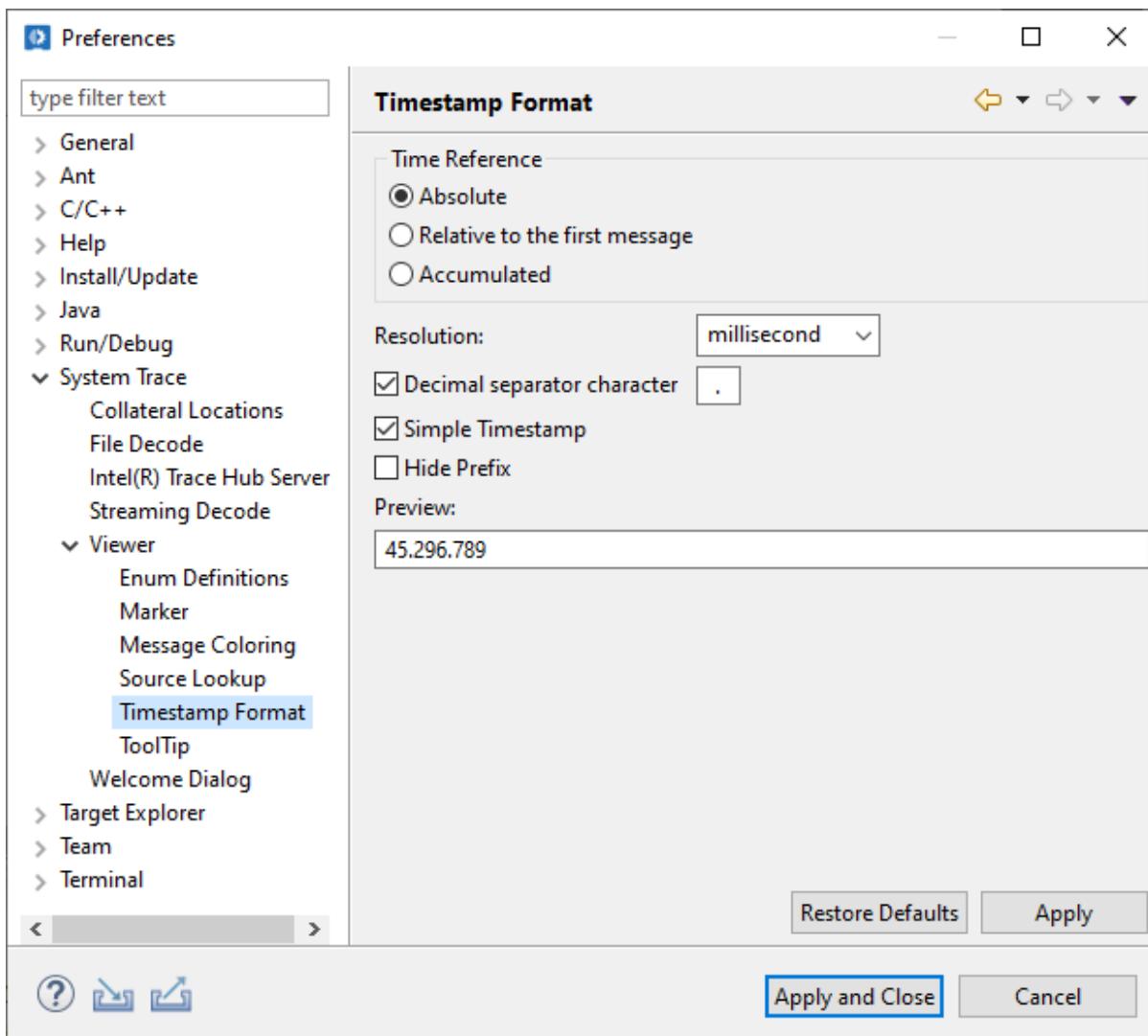
You can import the suitable file(s) to the **capture**, **configure**, **sessions**, or **timelines** folder by drag-and-dropping them into the action node below the folder you wish to import to.



Changing the Timestamp Format

You can change the format of the message timestamp shown in the Message View, Message Distribution (see section 5.4), Marker View (see section 5.9), and Timeline Viewer (see section 5.16.4). To change the format of the message timestamp:

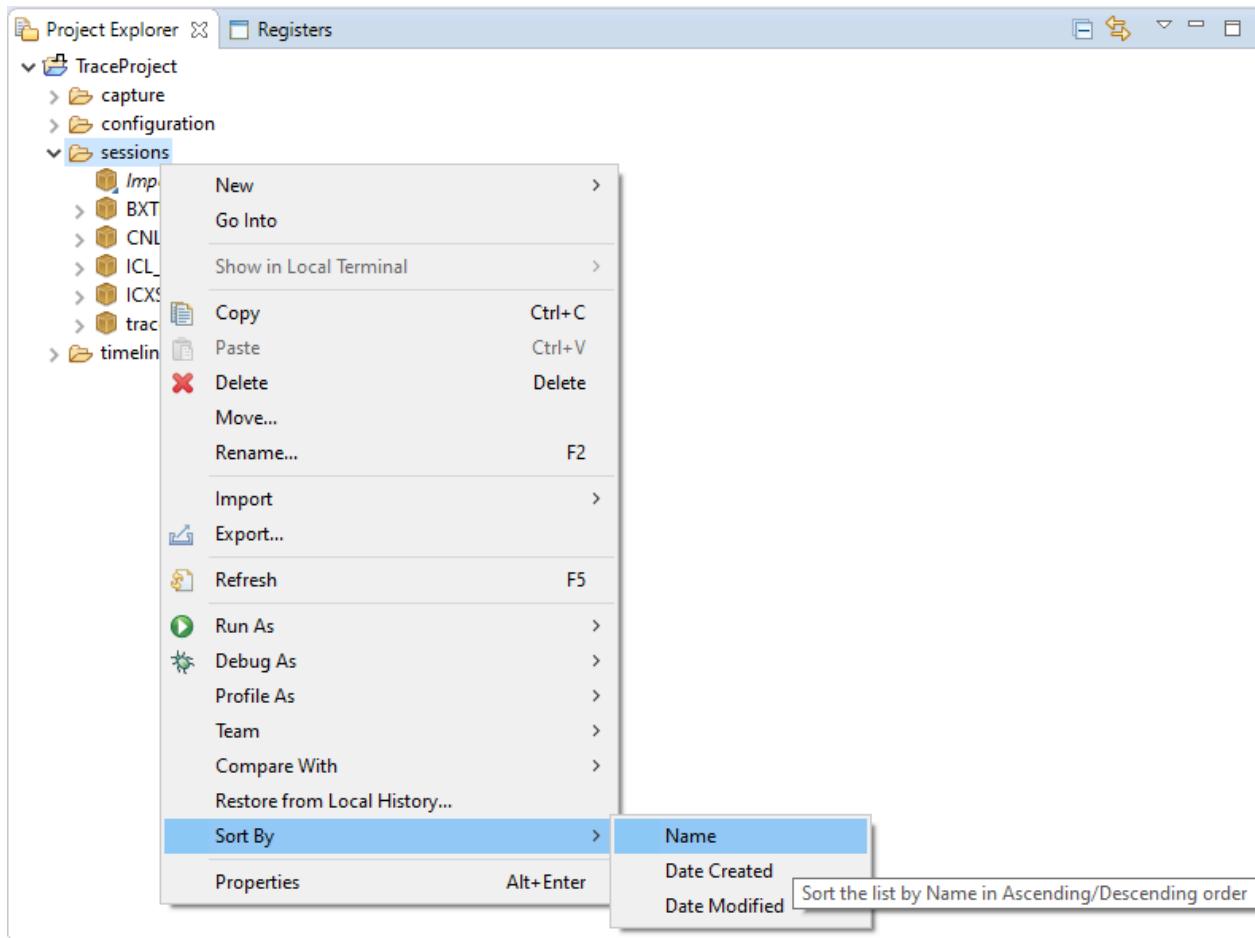
1. Select **Window > Preferences** to open the Preferences dialog box.
2. Navigate to **System Trace > Viewer > Timestamp Format** to change the setting.
3. Click **Apply** to save your changes or **Apply and Close** to save changes and close the dialog.



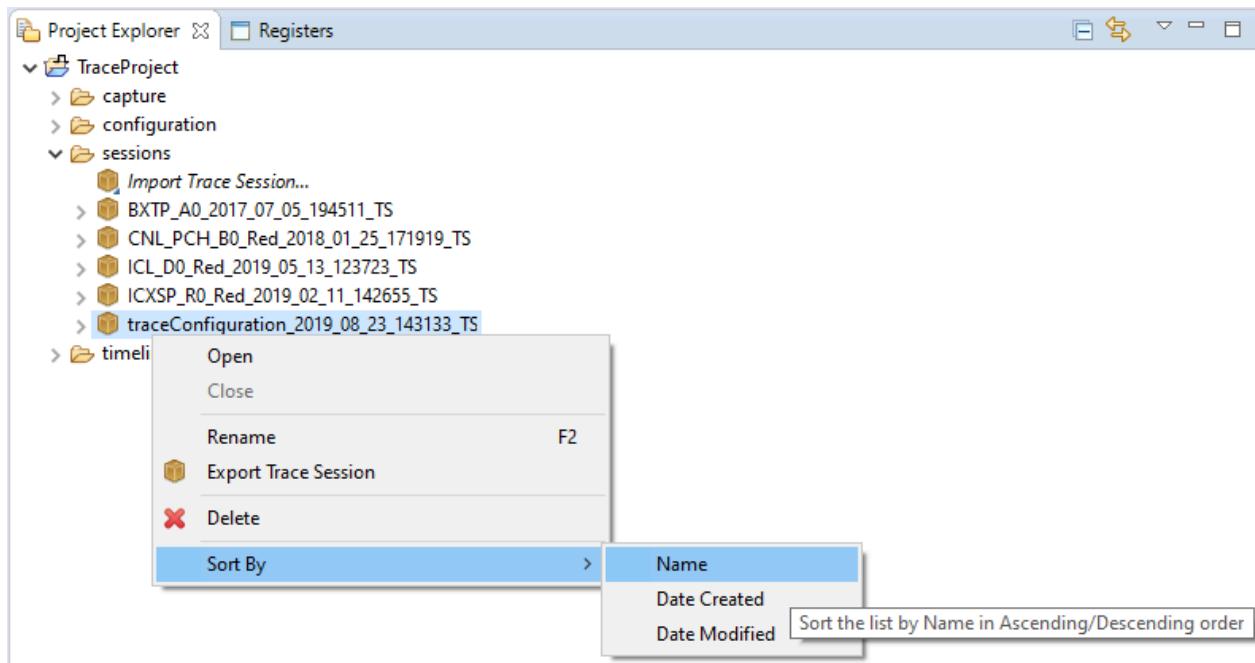
Sorting Files in the Project Explorer

Depending on the file system in your operating system, you can sort the files in the four main folders in the Project Explorer based on their name, date created, or date modified in ascending or descending order. For example, to sort files in the **sessions** folder by name in ascending order, do any of the following:

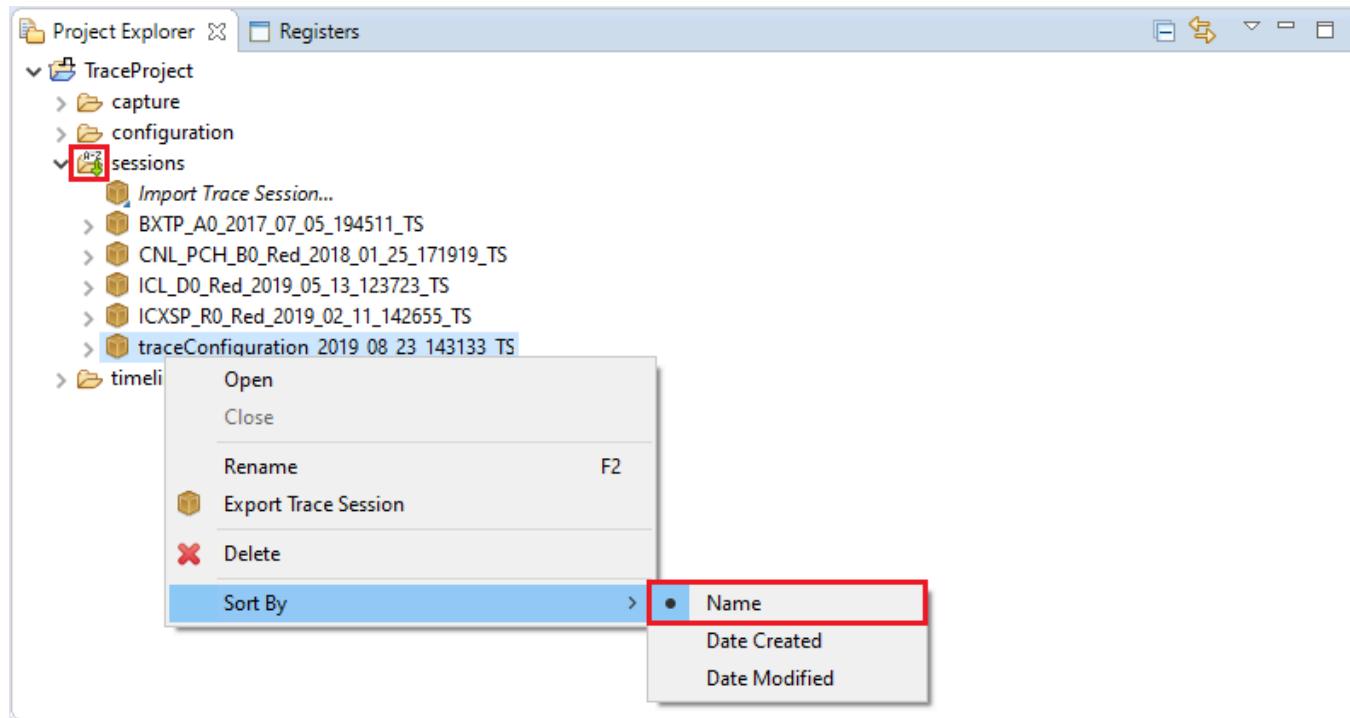
- Right-click the folder and select **Sort By > Name**



- Right-click one of the session files and select **Sort By > Name**.



The option **Name** in the **Sort By** menu is now selected and a status icon is displayed on the top-right corner of the folder icon to indicate that the files in the **sessions** folder are sorted by name in ascending order.



The status icon can be one of the following:

- Name Ascending; Name Descending
- Date Created Ascending; Date Created Descending
- Date Modified Ascending; Date Modified Descending

To change the sorting order, click the same selection again. For example, to sort the session files by name in descending order, select **Sort By > Name** again.

Export Data

The default way of exporting data (captured trace, trace session, configuration, and more) for further analysis is the following:

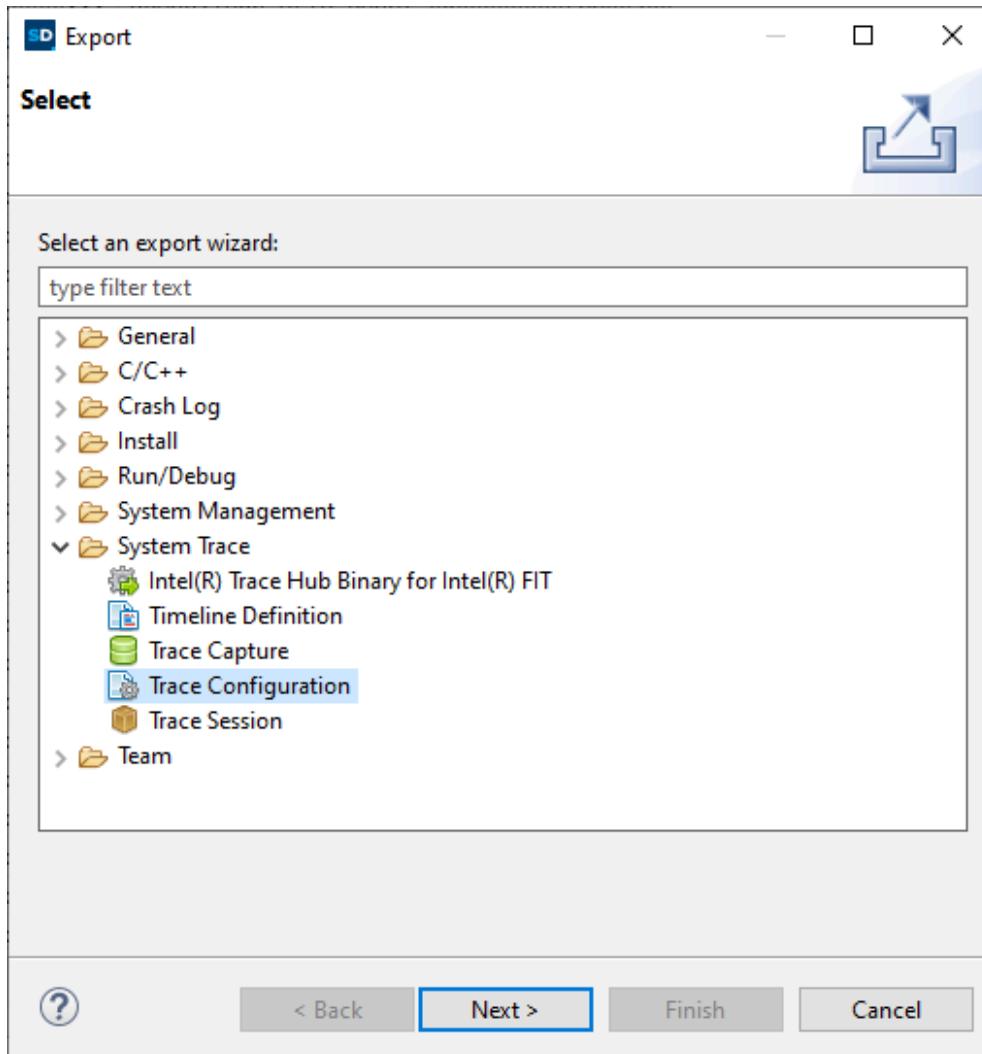
1. In the main menu bar, select **File > Export**
2. In the opened dialog box, expand the **System Trace** node and select the data type to export.

For definitions of each data type and default file extensions, see [File Types in System Trace](#).

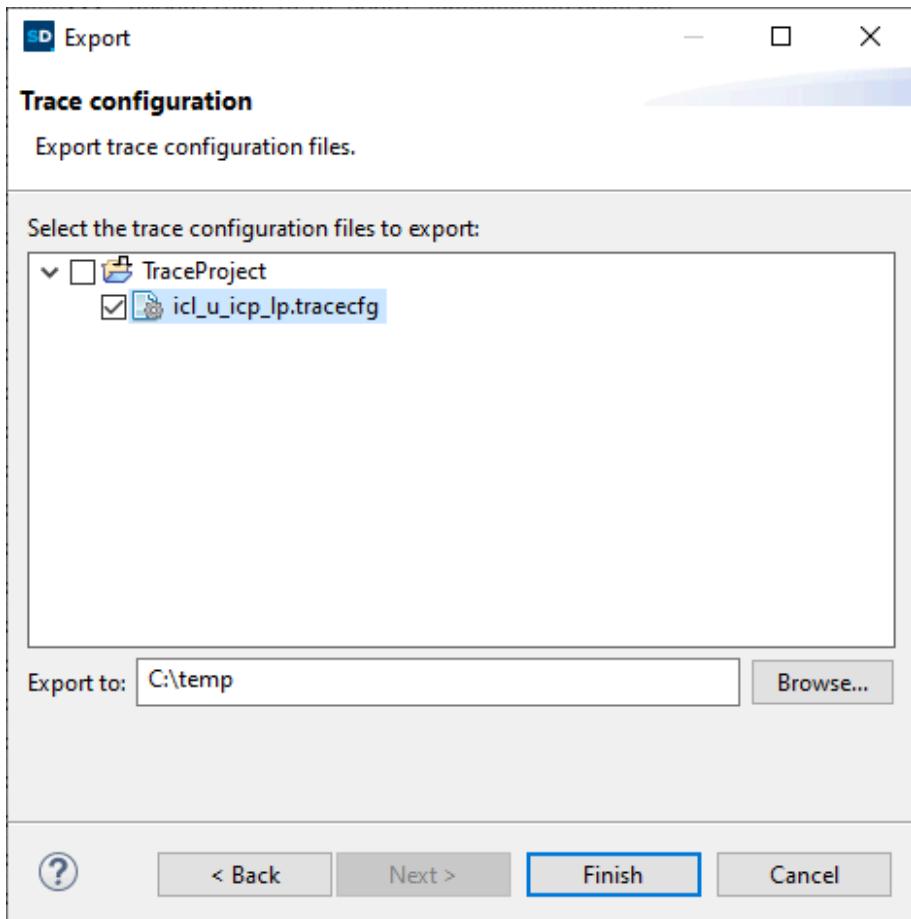
You can also export decoded trace messages in another format. See [Export Trace Messages to CSV](#).

Note

To export the Intel(R) Trace Hub Binary for Intel(R) Flash Image Tool (Intel(R) FIT), see [additional instructions](#).



3. In the next dialog box, check the instances to export and click **Browse** to specify the destination directory.



4. Click **Finish**.

! See also

- [Export Trace Messages to CSV](#)
- [Export Intel\(R\) Trace Hub Binary for Intel FIT](#)

Export Intel(R) Trace Hub Binary for Intel(R) Flash Image Tool (Intel(R) FIT)

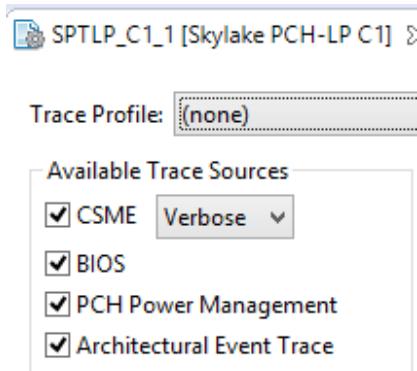
Firmware images for different boards can be created using Intel(R) Flash Image Tool (Intel(R) FIT). The tool has an option for setting the Intel(R) Trace Hub portion of the firmware image. Normally, trace sources such as, for example, BIOS or CSME are disabled on target by default. The way to enable them is to attach Intel(R) Tracehub Restore image to BIOS firmware. This one can force the target to configure trace sources automatically. **The binary file (Intel(R) Trace Hub Binary) can be stitched to the target firmware image using a Intel(R) FIT tool. This enables the sources to be active on a target power-on.** The only condition is to be connected to the target and to do the live tracing once it gets powered.

Alternatively, the same file same can be found in any capture/config folder on the file system. Every "start tracing" operation generates this file with a name ith_restore.bin.

Intel(R) Flash Image Tool requires Intel(R) Trace Hub Binary to generate the firmware image. Intel(R) Trace Hub Binary represents values of all MSRs of the Intel(R) Trace Hub. If the MSR values are set in a way that certain trace sources are enabled then the target will have these trace sources enabled by default.

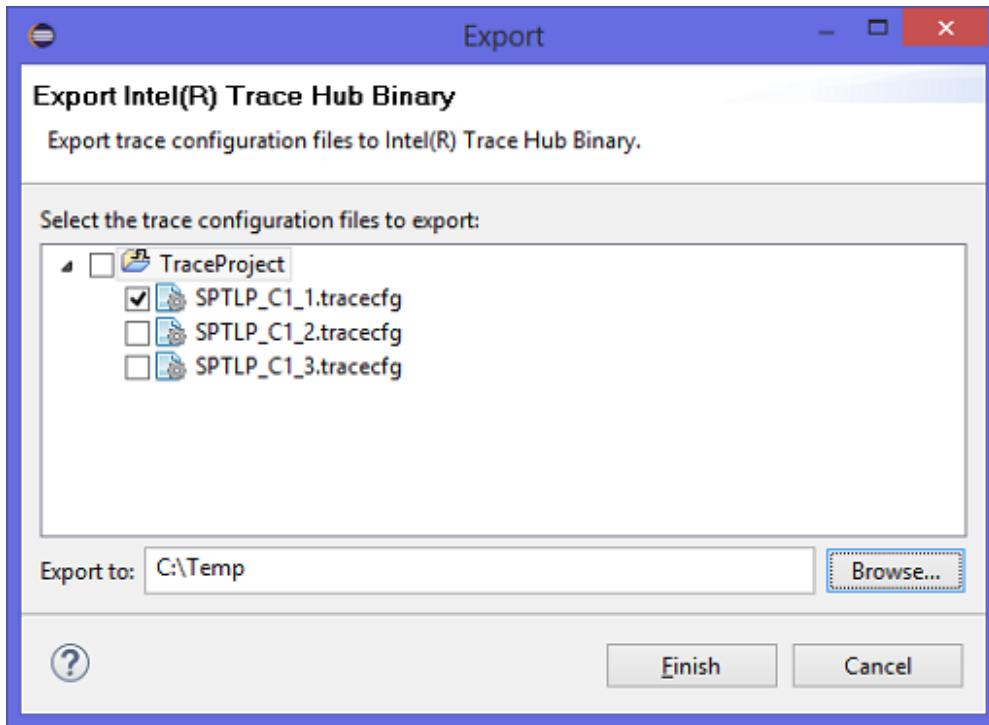
As an example, an Intel(R) Trace Hub Binary with all trace sources enabled will be created. This can be done with the following steps:

1. Select all trace sources for **SPTLP_C1_1.tracecfg** in Source and Destination panel.



2. Right-click on **SPTLP_C1_1.tracecfg** and go to **Export→Intel(R) Trace Hub Binary for Intel(R) FIT**

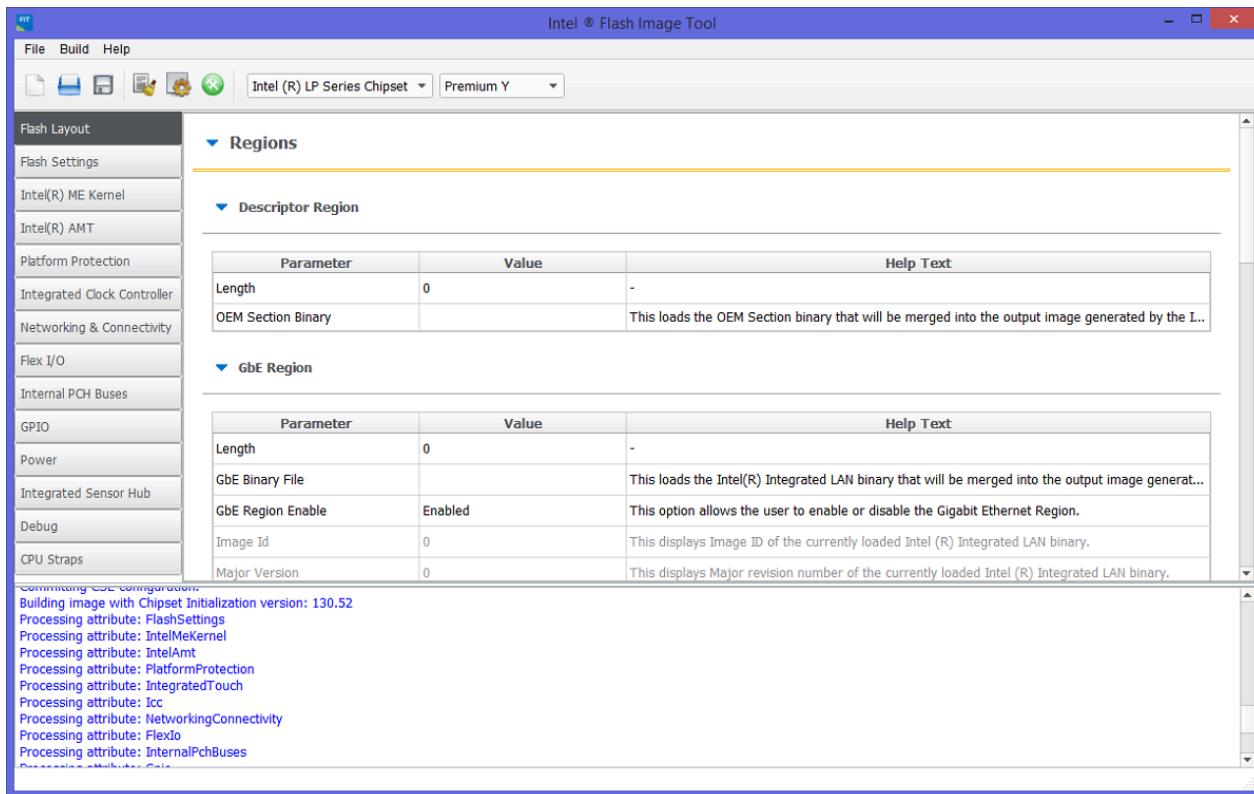
3. Choose the destination folder (c:\TEMP as an example)



and click to **Finish**.

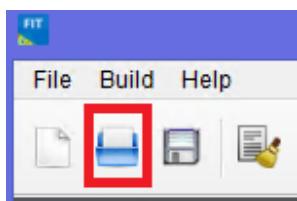
In the destination folder there will be now **Trace_Hub_Binary_For_Intel_Fit_SPTLP_C1_1.bin** file. This file has all trace sources enabled. It can be used in the Intel(R) Flash Image Tool to create a firmware image.

4. Open the Intel(R) Flash Image Tool



5. Set your own configuration manually (by setting every field in the tool manually) or load an existing firmware image.

6. Loading an existing firmware image can be done by simply opening a *firmware.bin* file.



The Intel(R) Flash Image Tool will populate all of its fields.

7. In the **Flash Layout** tab set Intel(R) Trace Hub Binary field to the newly created binary.

Parameter	Value	Help Text
Length	0	-
Intel(R) ME Binary File		This loads the Intel (R) ME binary that will be merged into the output image generated by the I...
Major Version	0	This displays Major revision number of the currently loaded Intel (R) ME binary.
Minor Version	0	This displays Minor revision number of the currently loaded Intel (R) ME binary.
Hotfix Version	0	This displays Hot-Fix revision number of the currently loaded Intel (R) ME binary.
Build Version	0	This displays Build version number of the currently loaded Intel (R) ME binary.
Chipset Initialization Version		This displays the current Chipset Initialization version contained in the currently loaded Intel (R)...
Chipset Initialization Binary		This loads the Chipset Initialization binary that will be merged into the output image generated ...
ChipsetInit Override Version		This displays the version of the Chipset Initialization Binary override, if specified.
Intel (R) Trace Hub Binary	C:\Temp\Trace_Hub_Binary_Fo...	This loads the Intel (R) Trace Hub binary that will be merged into the output image generated ..

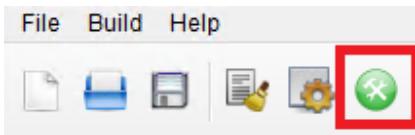
▼ PDR Region

Parameter	Value	Help Text
-----------	-------	-----------

! Note

If the pre-existing flash image is loaded this field will already have a value set to show the binary section of the loaded firmware image. In that case, change the value to point to the generated Intel(R) Trace Hub Binary file from the Trace tool.

8. Press the **Build Image** button to build the firmware image.



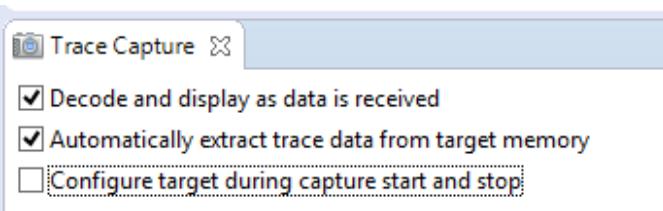
The image is by default stored as *output.bin* in the FIT root folder.

9. Flash the image to the target.

10. Now, when using the Trace tool the “Configure target during capture start and stop” option is unnecessary since the target is already configured.

! Note

This option can still be used; however, in this case the target power-on configuration will be overwritten.



! Note

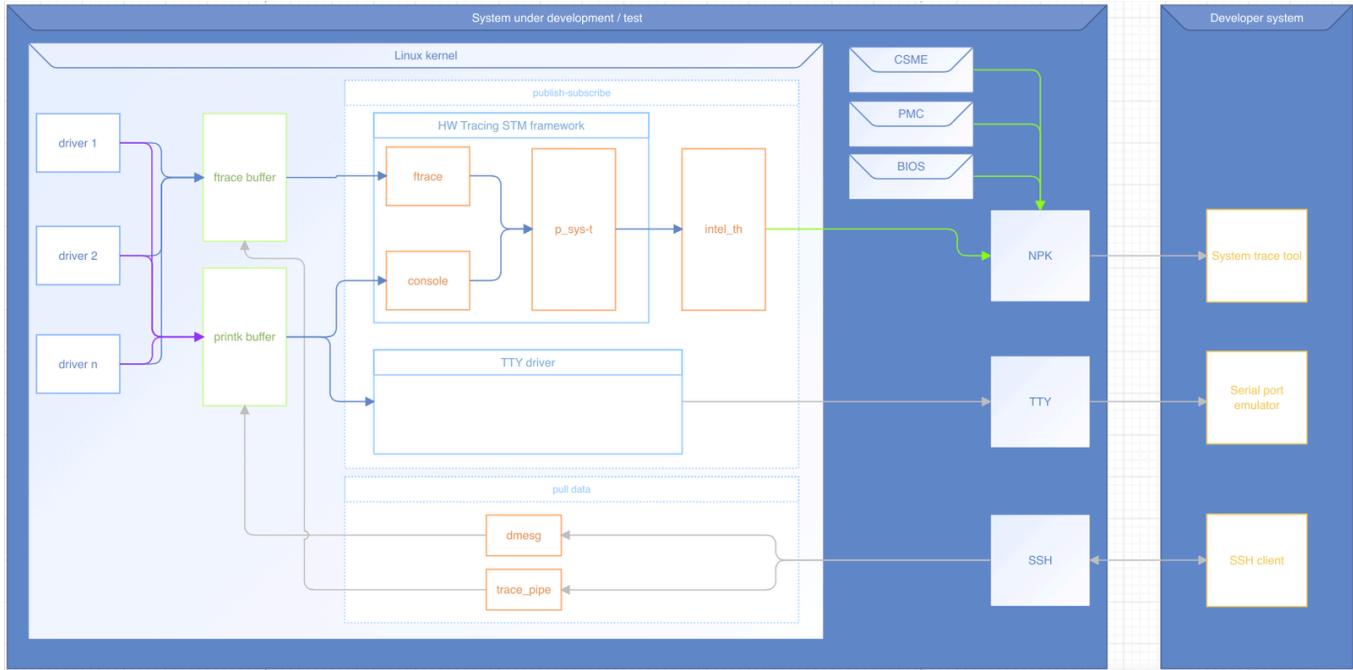
Intel(R) Trace Hub Binary will also be set to trace to the current trace destination.

! Important

Tracing to memory is not supported. One of the probes (for example, Intel(R) Direct Connect Interface (Intel(R) DCI) Out of Band (OOB)) must be selected as a trace destination.

Linux kernel tracing

Linux kernel tracing allows to pipe dmesg, userspace messages and kernel ftrace events through Intel(R) TraceHub resulting in a common firmware/kernel log.



The Linux kernel patches which are required for ftrace event tracing are shipped with the trace tool since they are not part of the upstream Linux kernel yet.

Patchset is based on Linux v6.1-rc1 and is tested against Ubuntu 22.04 LTS 5.15.0-41 kernel

```
git format-patch -v6 -o outgoing -base=v6.1-rc1 -3
```

The linux kernel patches can be found in Intel(R) System Trace installation folder, i.e.:

C:/IntelSWTools/system_debugger/XXXX-
nda/system_trace/agent/plugins/linux_trace/kernel_patch/linux_ftrace_support.zip
SHA256 9cceef2ec6af237a6eaa3391615f67134edbd2c60ad42cbbf2299c9d6adcbe0
MD5 0fb47da9a86becfd35560eefbf81ed5c0

Prerequisites

1. Following modules must be built as part of kernel configuration and loaded during startup or manually before any of described further steps are executed:

- Intel(R) TraceHub modules

- intel_th_msu
- intel_th_sth

```

<M> Intel(R) Trace Hub controller
<M> Intel(R) Trace Hub test kernel module
<M> Intel(R) Trace Hub PCI controller
<M> Intel(R) Trace Hub ACPI controller
<M> Intel(R) Trace Hub Global Trace Hub
<M> Intel(R) Trace Hub Software Trace Hub support
<M> Intel(R) Trace Hub Memory Storage Unit
<M> Intel(R) Trace Hub PTI output
[*] Intel(R) Trace Hub debugging

```

- STM Framework modules

- stm_console
- stm_ftrace
- stm_p_sys-t

```

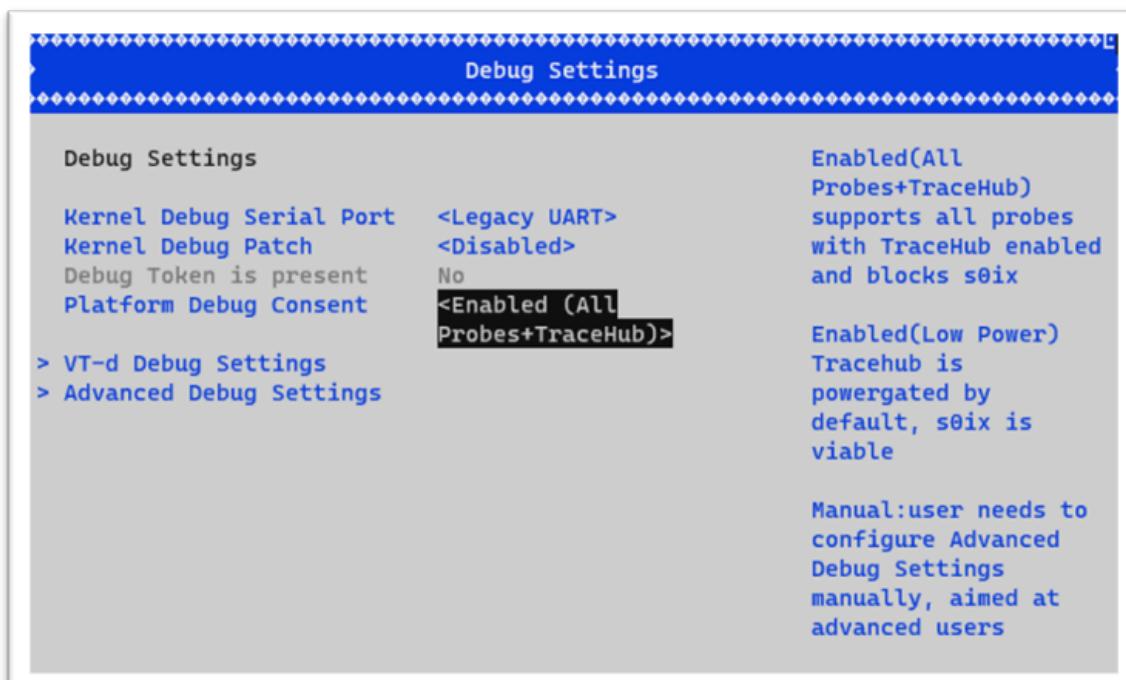
<M> System Trace Module devices
<M> Basic STM framing protocol driver
<M> MIPI SyS-T STM framing protocol driver
<M> Dummy STM driver
<M> Kernel console over STM devices
< > Heartbeat over STM devices
<M> Copy the output from kernel Ftrace to STM engine

```

2. Target agent must be copied/installed on target machine

3. BIOS setup

- Enable debug consent



- Leave other switches in *Advanced debug settings* default

Advanced Debug Settings		
USB3 Type-C UFP2DFP	<No Change>	This BIOS option enables kernel and platform debug for USB3 interface over a UFP Type-C receptacle, select 'No Change' will do nothing to UFP2DFP setting
Kernel/Platform Debug Support		
USB DbC Enable Mode	<No Change>	
DCI Enable	<Enabled>	
PCH Trace Hub Enable Mode	<Host Debugger>	
PCH TH Mem Buffer Size 0	<8MB>	
PCH TH Mem Buffer Size 1	<8MB>	
CPU Trace Hub Enable Mode	<Host Debugger>	
CPU TH Mem Buffer Size 0	<8MB>	
CPU TH Mem Buffer Size 1	<8MB>	
CPU Run Control	<No Change>	
USB Overcurrent Override for VISA	<Disabled>	
Processor trace memory allocation	<Disabled>	
JTAG C10 Power Gate	<Enabled>	
Three Strike Counter	<Enabled>	
CrashLog Feature	<Enabled>	
CrashLog On All Reset	<Disabled>	
CrashLog Ralarm Enable	<Enabled>	
CrashLog Clear Enable	<Disabled>	
CrashLog GPRS	<Disabled>	
PMC Debug Message Enable	<Disabled>	
Delayed Authentication Mode	<Disabled>	

4. Kernel commandline setup

To allow kernel write into Intel(R) TraceHub you need to switch intel_th driver into *Host mode*

1. Append Linux command line

```
GRUB_DEFAULT=0
GRUB_TIMEOUT_STYLE=countdown
GRUB_TIMEOUT=3
GRUB_DISTRIBUTOR='lsb_release -i -s 2> /dev/n
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX="intel_th.host_mode=Y"
```

2. Update bootloader settings sudo update-grub or sudo update-grub2
3. Reboot the machine
4. Check the setting was applied

```
cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.15.0-41-generic root=UUID=8cba1f3f-6f0b-4a27-816d-e15de21c91e2 ro intel_th.host_mode=Y quiet splash vt.handoff=7
```

Setup STM policy

This part is automatically set up by OnTarget agent.

Below setup script are provided for understanding and reference what agent does:

```

#!/bin/bash
set -x
set -e

# mount configfs
mkdir -p /config
mount -t configfs none /config

# create STP policy
# default policy - if not created and any other can be applied - writer gets -EINVAL
# NPK masters 257-258
mkdir -p /config/stp-policy/0-sth:p_sys-t.my-policy/default/
echo 257 258 > /config/stp-policy/0-sth:p_sys-t.my-policy/default/masters

# printk policy (1)
# NPK masters 261-262
mkdir -p /config/stp-policy/0-sth:p_sys-t.my-policy/console/
echo 261 262 > /config/stp-policy/0-sth:p_sys-t.my-policy/console/masters
echo 0-sth > /sys/class/stm_source/console/stm_source_link

# ftrace policy (2)
# NPK masters 259-260
echo 259 260 > /config/stp-policy/0-sth:p_sys-t.my-policy/ftrace/masters
# UUID should correspond to trace extension catalog
echo 6e5dd011-0a15-4380-b413-d1e61ce9d005 > /config/stp-policy/0-sth:p_sys-t.my-policy/ftrace/uuid
echo 0-sth > /sys/class/stm_source/ftrace/stm_source_link

```

Start tracing

TraceCLI

1. Start OnTarget trace agent: `sudo ./agent/bin/intel_trace_agent` with root permissions on SUT
2. Start TraceCLI and connect to target agent `trace_agent.connect(ip=<ip-address>, port=1534)`
3. Start regular trace capture `trace.start_capture()`
4. After test run (i.e. problem reproducer) stop the capture with `trace.stop_capture()`
5. Decode the file with `trace.decode()` command or use Eclipse UI to decode the trace capture

Eclipse UI

Linux OnTarget tracing flow is not yet supported in Eclipse UI

Known limitations

Passing arbitrary strings through ftrace events is not supported.

Example driver

Define 2 events in test driver: one for loading/unloading (th_test_trace_mod) and one timer event (th_test_trace_timer)

```

#define TRACE_SYSTEM
#define TRACE_SYSTEM th_test_trace

#if !defined(_TRACE_TH_TEST_TRACE_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_TH_TEST_TRACE_H

#include <linux/tracepoint.h>

TRACE_EVENT(th_test_trace_mod,
    TP_PROTO(bool start),
    TP_ARGS(start),
    TP_STRUCT__entry(
        __field(bool, start)
    ),
    TP_fast_assign(
        __entry->start = start;
    ),
    TP_printk("Module: %s", __entry->start ? "start" : "end")
);

TRACE_EVENT(th_test_trace_timer,
    TP_PROTO(uint32_t counter, const char* text),
    TP_ARGS(counter, text),
    TP_STRUCT__entry(
        __field(uint32_t, counter)
        __field(char, text[14])
    ),
    TP_fast_assign(
        __entry->counter = counter;
        memcpy(__entry->text, text, strlen(text));
    ),
    TP_printk("event: TraceHub test counter: %u %s", __entry->counter, __entry->text)
);

#endif

#include <trace/define_trace.h>

```

Use these events in your kernel driver i.e. trace_th_test_trace_timer(...)

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/timer.h>

#define CREATE_TRACE_POINTS
#include <trace/events/th_test_trace.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mikhail Lappo");
MODULE_DESCRIPTION("A test kernel module for Intel TraceHub.");
MODULE_VERSION("0.01");

static struct timer_list periodic_event_timer;
static uint32_t timer_counter = 0;

void fire_test_event(struct timer_list* timer)
{
    const char t1[14] = "**** test1 ***";
    const char t2[14] = "**** test2 ***";
    const char t3[14] = "**** test3 ***";
    printk(KERN_ERR "-----\n");
    printk(KERN_ERR "printk: Timer event in TH %u\n", timer_counter);
    trace_th_test_trace_timer(timer_counter++, t1);
    printk(KERN_ERR "-----\n");
    mod_timer(&periodic_event_timer, jiffies + msecs_to_jiffies(1000));
}

static int __init intel_th_test_init(void) {
    printk(KERN_INFO "Hello, Intel TH!\n");
    timer_setup(&periodic_event_timer, fire_test_event, 0);
    mod_timer(&periodic_event_timer, jiffies + msecs_to_jiffies(1000));
    trace_th_test_trace_mod(true);
    return 0;
}
static void __exit intel_th_test_exit(void) {
    trace_th_test_trace_mod(false);
    del_timer(&periodic_event_timer);
    printk(KERN_INFO "Goodbye, Intel TH!\n");
}

module_init(intel_th_test_init);
module_exit(intel_th_test_exit);

```

Appendix A: Setting Up Hardware

This chapter describes target-specific setup steps to start live capturing from the Intel(R) Trace Hub device on the target.

Ice Lake - Integrated Sensor Hub

This section describes how to configure the target for live trace streaming using the System Trace.

Prerequisites

- Trace Message Catalog

The Integrated Sensor Hub uses a message catalog for kernel messages. You have to install the decode catalog to be able to receive usable traces from the Integrated Sensor Hub. Please contact Intel support to receive the message catalog file.

- Windows OS: Integrated Sensor Hub Driver*

Windows* OS communicates with Integrated Sensor Hub firmware using the kernel driver.

Integrated Sensor Hub drivers are part of Intel(R) Best Known Configuration Package Manager (Intel(R) BKC Package Manager) and are available for download as part of the Integrated Firmware Image (IFWI) package.

The driver comes with the Integrated Sensor Hub Firmware Development Kit tools available at *C:\Intel_ISS_FDKTools*.

One of the tools included in the package is the SensorViewer: a basic tool to analyze the current sensor state. In case SensorViewer fails to identify sensors, run the Troubleshooter tool to get more information.

Setup

1. Apply the OEM debug token.

To debug the Integrated Sensor Hub (ISH) firmware, you have to stitch your OEM debug token with the Integrated Firmware Image (IFWI). Contact Intel support to learn how to generate the debug token.

2. Enable Integrated Sensor Hub firmware in the FitC tool.

The image below shows the correct settings with the `issC.bin` file used as input:

The screenshot displays the FitC tool's configuration interface with three main sections: Integrated Sensor Hub, ISH Image, and ISH Data.

Integrated Sensor Hub:

Parameter	Value	Help Text
Integrated Sensor Hub Supported	Yes	This setting allows customers to enable / disable ISH on the platform.
Integrated Sensor Hub Initial P...	Enabled	This setting allows customers to determine the power up state for ISH.

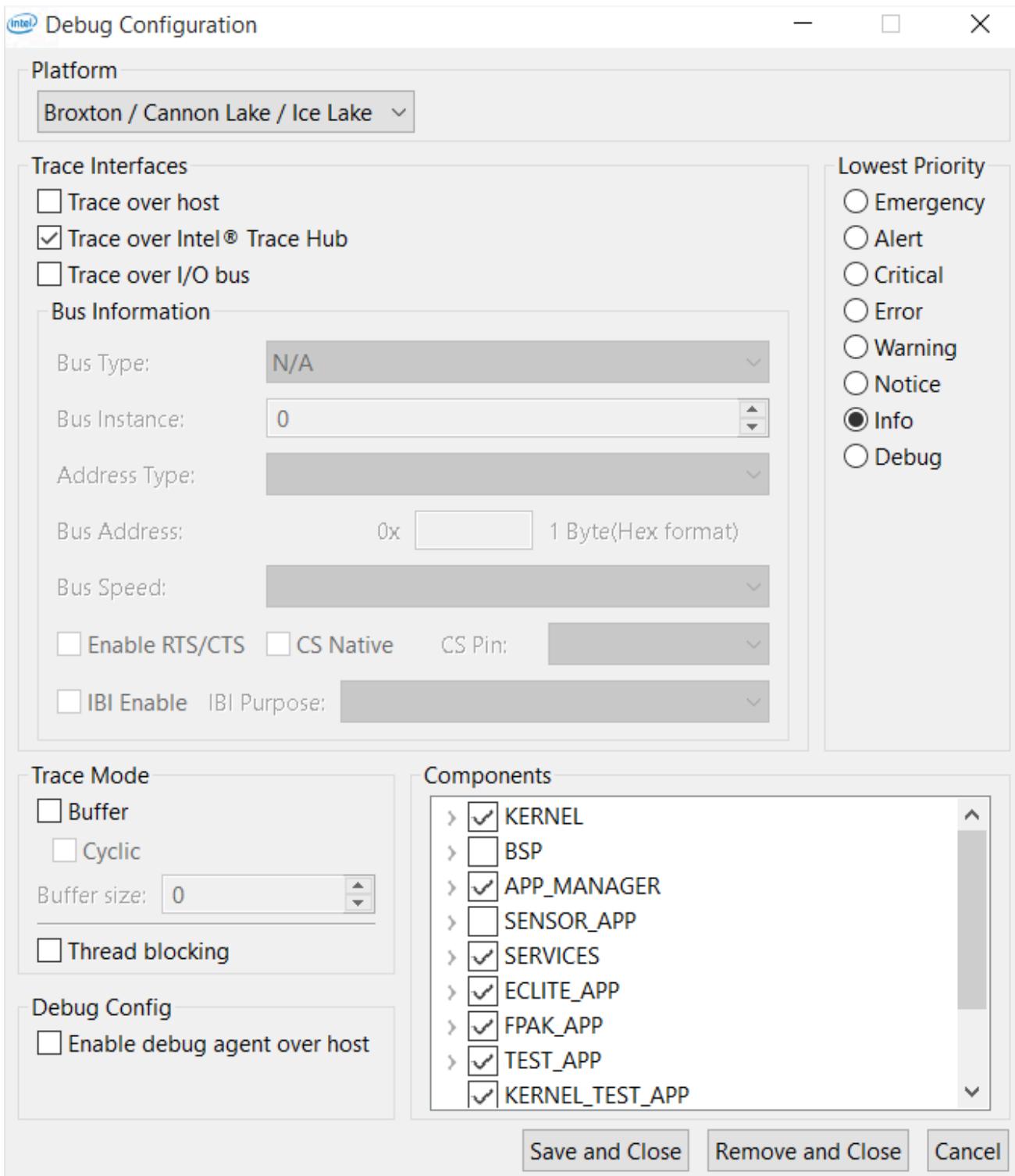
ISH Image:

Parameter	Value	Help Text
Length	0x40000	Total size (in bytes) of the ISH c...
Input File	C:\Users\...\workspace_icl_dbg\DummySensor\bin\icf\Debug\issC.bin	Path to your ISH firmware binary...

ISH Data:

Parameter	Value	Help Text
PDT Binary File	C:\Users\...\workspace_icl_dbg\DummySensor\bin\PDTs\ICL_PDT\ICL_PDT.bin	Path to your PDT binary file.

3. (Optional) Enable the Intel(R) Trace Hub messages at compile time using the PDT Editor. Select the PDT Editor settings as shown below:



4. Connect to the target using an Intel(R) Direct Connect Interface cable.

5. Setup BIOS.

To enable tracing, enter the BIOS and change the settings as follows:

- Go to Intel Advanced Menu > PCH-IO Configuration > Trace Hub Configuration Menu. Set **Trace Hub Enable Mode** to <Host Debugger>.
- Go to Intel Advanced Menu > PCH-IO Configuration. Set **Intel(R) Direct Connect Interface (Intel(R) DCI) Enable (HDCIEN)** to <Enabled>.
- Go to Intel Advanced Menu > CPU Configuration. Set **Debug Interface** and **Intel(R) Direct Connect Interface** to <Enabled>.

Save and exit the BIOS.

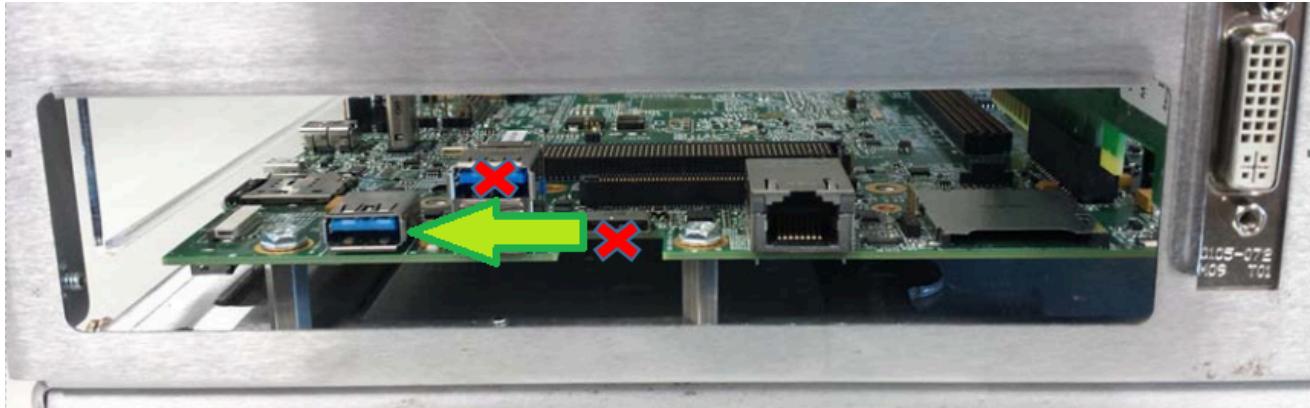
6. Select proper trace sources

Create a trace configuration file and select proper trace sources. For instructions, see [Select Trace Sources](#).

Skylake PCH-H and Skylake PCH-LP

1. Connect the hardware.

Connect a USB cable to the target side of the Closed Chassis Adapter (CCA) and to the solitary USB port on the target board. Connect a USB cable to the host side of the CCA and to the host running the Intel(R) System Debugger NDA.



2. Setup BIOS.

To enable tracing, enter the BIOS and change the settings as follows:

- Go to Intel Advanced Menu > PCH-IO Configuration > Trace Hub Configuration Menu. Set Trace Hub Enable Mode to <Host Debugger>.
- Go to Intel Advanced Menu > PCH-IO Configuration. Set Intel(R) Direct Connect Interface (Intel(R) DCI) Enable (HDCIEN) to <Enabled>.
- Go to Intel Advanced Menu > CPU Configuration. Set Debug Interface and Intel(R) Direct Connect Interface to <Enabled>.

Save and exit the BIOS.

Broxton P

1. Setup hardware for Intel(R) Direct Connect Interface (Intel(R) DCI) USB 3.x Debug Class (USB3 cable).

For APL RVP-1 reference board with a MIPI 60 connector, the TRST pin needs to be pulled high. The fix is to short TRST to VDD with a 50-Ohm resistor. The picture below shows the workaround for a Zebax ZX100 breakout adapter:



The shown adapter is not the only working breakout adapter on the market. Any Samtec60 QTH breakout board should work.

2. Connect hardware using Intel(R) DCI USB 3.x Debug Class (USB3 Cable)

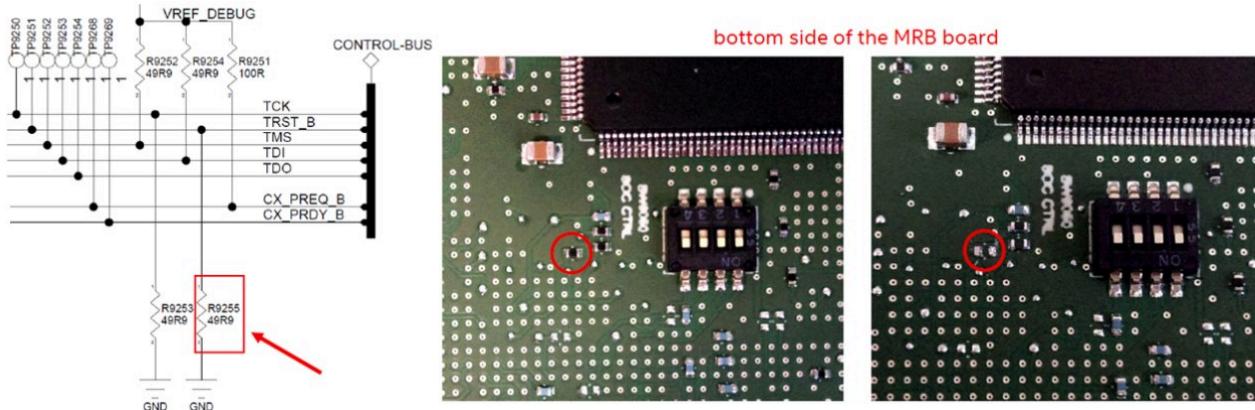
Make sure that you have an Intel(R) DCI USB 3.x Debug Class debug cable available. These cables can be identified based on their yellow or orange connectors.

Connect the host machine's super speed USB port (USB3) with the target machine's super speed USB (USB3) port, using an Intel(R) DCI USB 3.x Debug Class debug cable.

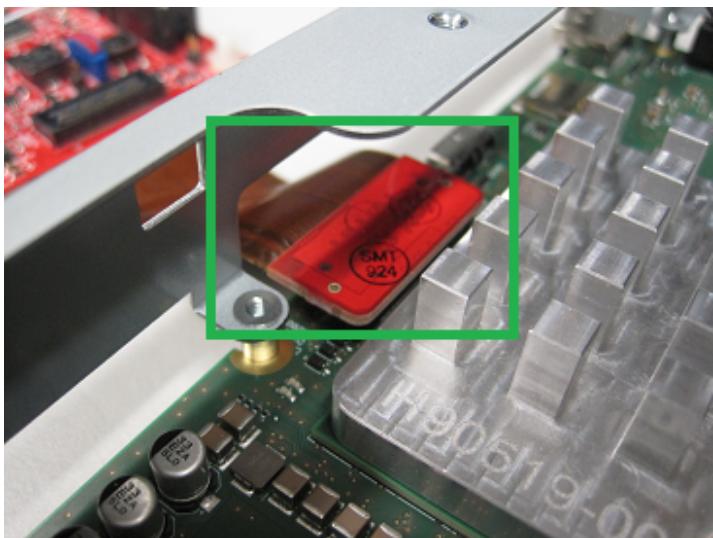
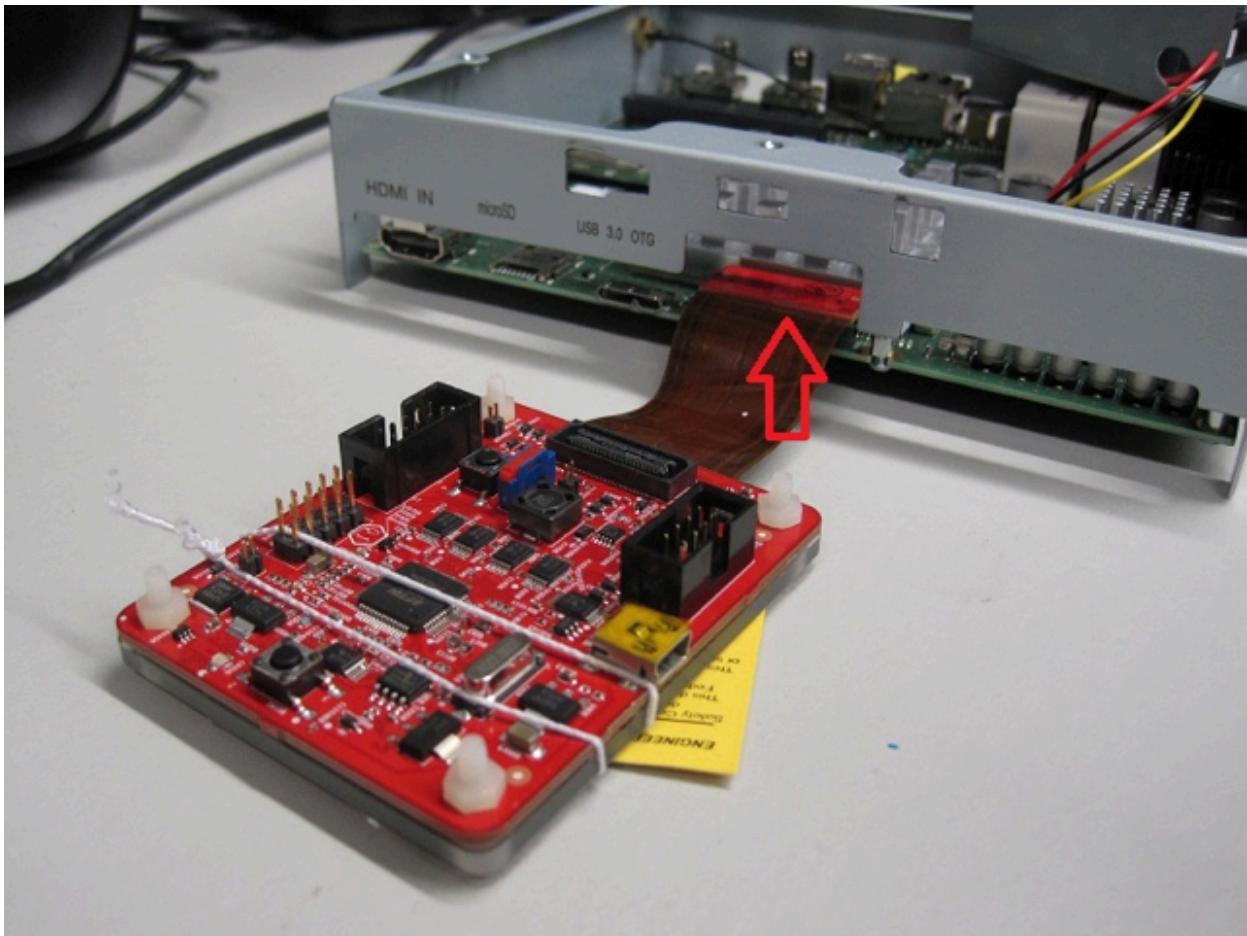
Gordon Peak Modular Reference Board (GP MRB)

Prerequisites

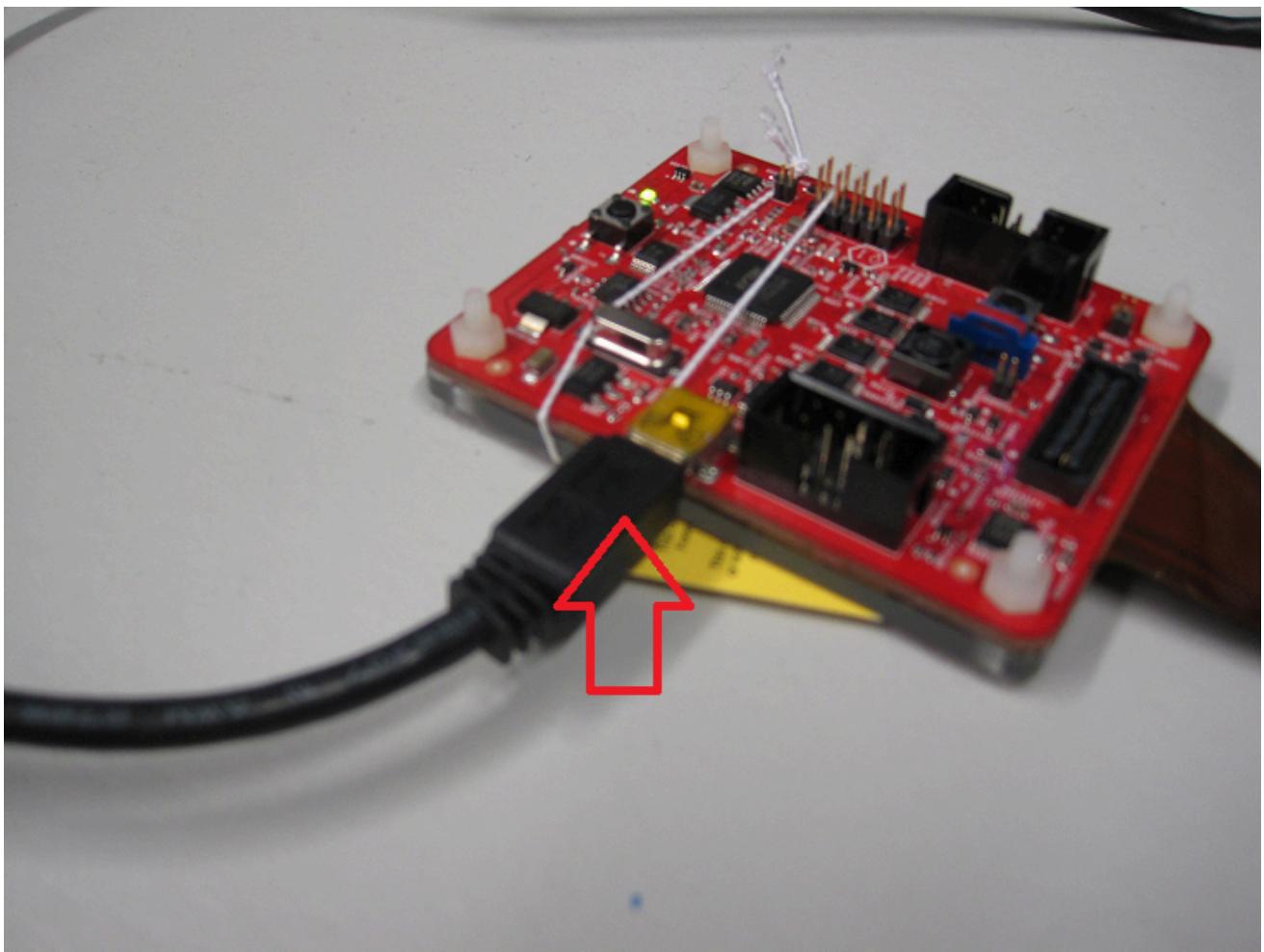
- Hardware Rework to enable Trace using Intel(R) Direct Connect Interface (Intel(R) DCI): Remove R9255 from the GP MRB board.



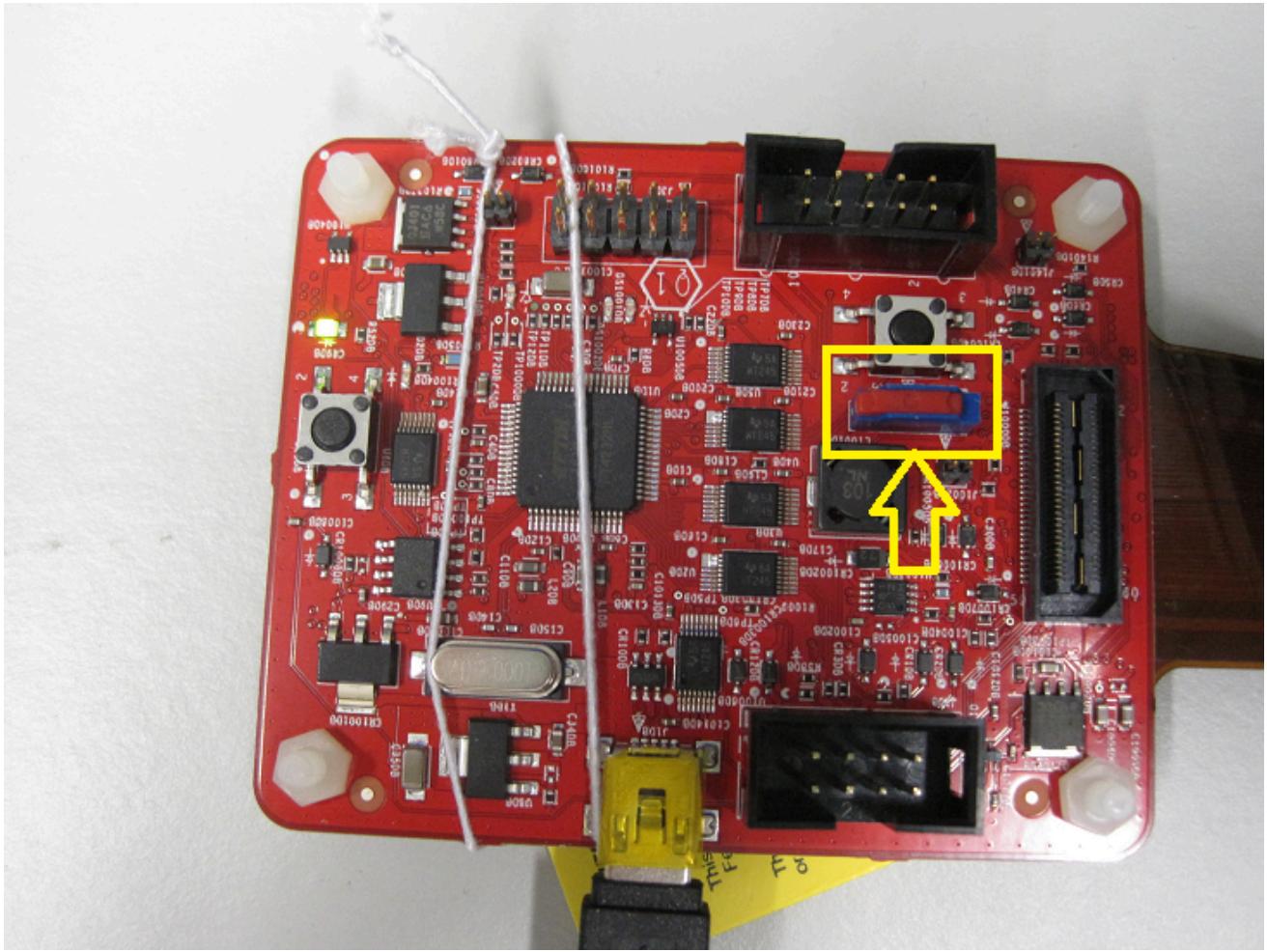
- Connect a DebugAdapter, shown in the pictures below, to the MRB-DebugAdapter port (green square).



- Make sure the DebugAdapter has power (USB cable to e.g. the host computer):

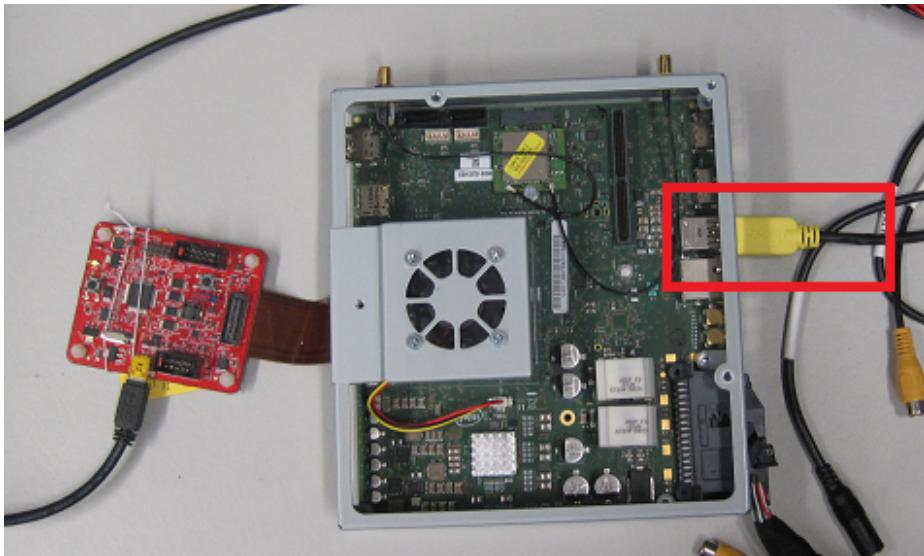


- Make sure to have the DebugAdapter switch in the correct position:

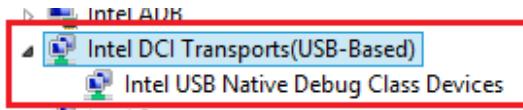


Setup

1. Make sure that you have an Intel(R) DCI USB 3.x Debug Class debug cable available. These cables can be identified based on their yellow or orange connectors.
2. Connect the host machine's super speed USB port (USB3) with the target machine's super speed USB (USB3) port, using an Intel(R) DCI USB 3.x Debug Class debug cable:



3. Make sure that the target system shows up in the Windows device manager:



Once you have fulfilled all requirements in this chapter, you can switch on the target.

Appendix B: Tracing with Simics(R) Simulator

System Trace allows you to capture traces from a Simics(R) simulator. The simulation can run on your local or a remote system.

At the moment only the capturing of traces is supported. To configure traces, you need to enable it in the IP (Simics(R) simulator) directly.

Configure Simics(R) Simulator

Follow instructions below to set required parameters:

Northpeak

Enable Northpeak inside Simics(R) simulator to be able to capture traces from it.

Other

Add the following to the startup Script:

```
$nPk_Consoles = TRUE  
$nPk_sb_Enable = TRUE  
$nPk_nb_Enable = TRUE
```

Configure and Enable Trace

Add the following North Peak configuration snippet after [run-command-file](#) of the startup script for the Simics(R) simulator.

Note, that script customization might be needed, as NPK location depends of Simics(R) image.

Script uses `$system.mb.soc.north_peak` to access NPK. On different image path might be different. Please refer to specific image documentation for NPK path.

For example, for GRR image, use the following path: `$system.mb.soc0.c_die0.npk` instead of:
`$system.mb.soc.north_peak`.

Configuration snippet:

```

# Configure NPK Traces

script-branch {

    while TRUE {

        local $system = $system

        wait-for-hap X86_Processor_Reset

        echo "Configure NPK after reset"

        # Enable Low Power Path

        $system.mb.soc.north_peak->bar_npk_LPP_CTL=0x13

        # Enable all BIOS traces

        $system.mb.soc.north_peak->bar_npk_SCRPD2 = 0xffffffff

        # Set "Debugger In Use" for FW, enable Intel(R) Direct Connect Interface (Intel(R) DCI)

        $system.mb.soc.north_peak->bar_npk_SCRPD0 = 0x01004000

        # Enable all master 256+, output Intel DCI

        $system.mb.soc.north_peak->bar_npk_GSWDEST = 9

        # Enable all master 0-256, output Intel DCI

        foreach $i in (range 32) {

            $system.mb.soc.north_peak->bar_npk_SWDEST[$i] = 0x99999999

        }

    }

}

# Enable Low Power Path

$system.mb.soc.north_peak->bar_npk_LPP_CTL=0x13

# Enable all BIOS traces

$system.mb.soc.north_peak->bar_npk_SCRPD2 = 0xffffffff

# Set "Debugger In Use" for FW, enable Intel DCI

$system.mb.soc.north_peak->bar_npk_SCRPD0 = 0x01004000

# Enable all master 256+, output Intel DCI

$system.mb.soc.north_peak->bar_npk_GSWDEST = 9

```

```
# Enable all master 0-256, output Intel DCI

foreach $i in (range 32) {

    $system.mb.soc.north_peak->bar_npk_SWDEST[$i] = 0x99999999

}
```

On Target Trace: Capture trace capture directly on device under test

Collecting traces is fundamental step to debug firmware and software. Traditionally Intel(R) System Trace Tool required that the user use Host and Target system to capture firmware, software, and hardware traces. Connecting Host to a target allows to collect boot logs and trace across low power, but challenging for collecting traces in automaton pools, for in-field test pools or small factor devices with single USB connector. On-Target Trace was developed to address limitation of host-based trace capture. Within new capability we split trace use cases into two buckets.

1. Host based solution aim to trace boot (pre-OS) and traces across power states (S4, S5) and low power. Recommended to be used by platform debug engineers, BIOS, CSME developers for active development and triage.
2. An on-target solution to collect logs during the OS working state, with limitations for S4 and low power. Recommended to be used in in-field validation and automation, by driver developers, audio, graphics, AI developer to get driver logs in correlation with firmware.

Since on target doesn't require Host system connection, it doesn't depend on Intel® Direct Connect Interface (Intel® DCI) enablement and can be enabled as standalone item as part of token management knob.

The on-target solution requires installation of the ITH driver. Follow ITH driver installation

On-Target Trace is part of Target Agent and deployed with the Intel(R) System Trace Tool as console tool. In order to start using On-Target script, open ISD_shell and type *intel_ontarget_trace*

```
C:\ Administrator: C:\windows\system32\cmd.exe
Welcome to the Intel System Debugger Dev Shell
From here you are able to run CLI interfaces included in Intel System Debugger
This feature is considered a proof of concept and can change at any time!

Example application available in this shell: python, ipccli, isd_cli, intel_tracecli, windbg_dci, windbg_preview, windbg
_simics, intel_sysdbg
C:\IntelSWTools\system_debugger\2246-nda>intel_ontarget_trace --help
usage: intel_ontarget_trace [-h] [-v] [--audio] [--acpi] [-dbgprint] [--csme] [--gsc] [--vpu] [--capture] [--decode]
                            [--decode-file DECODE_FILE] [--catalog-location CATALOG_LOCATION] [--symbols SYMBOLS]
                            [--memory-size {8,64,128,512,1024,2048,4096}]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose output
  --audio              Enable Audio trace use case
  --acpi               Capture ACPI logs
  --dbgprint           Capture Windows OS DbgPrint logs
  --csme               Enable Intel(R) CSME trace use case
  --gsc                Capture Intel(R) Graphics System Controller trace
  --vpu                Capture Intel(R) Movidius(TM) VPU trace
  --capture             Capture trace to file
  --decode              Decoder captured trace to file
  --decode-file DECODE_FILE
                        Decode file
  --catalog-location CATALOG_LOCATION
                        Location for catalog files. This is persistent between invocations
  --symbols SYMBOLS    Location for pdb or tmh files.
  --memory-size {8,64,128,512,1024,2048,4096}
                        Size of buffer in MB to be used by Intel(R) Trace Hub.

C:\IntelSWTools\system_debugger\2246-nda>
```

To get started with Target Agent, select trace sources, memory size and action.

```
C:\ Administrator: C:\windows\system32\cmd.exe - intel_ontarget_trace --capture --decode --vpu --csme
C:\IntelSWTools\system_debugger\2246-nda>intel_ontarget_trace --capture --decode --vpu --csme
[06:53:03] [INFO      ] [ISD      ] Using workspace directory 'C:\IntelSWTools\system_debugger\2246-nda' for console
session
[06:53:05] [INFO] Intel64 Family 6 Model 170 Stepping 1, GenuineIntel identified as mtl_m
[06:53:05] [INFO] On target script detected platform: mtl_m
[06:53:05] [INFO] Connecting to trace agent 127.0.0.1:1534.
[06:53:05] [INFO] Launching C:\IntelSWTools\system_debugger\2246-nda\system_trace\agent\bin\intel_trace_agent.exe on:
TCP:127.0.0.1:1534
[06:53:22] [INFO] Sending signal to configure trace start
(   •   ) Trace is running. Waiting for user input...
Press Enter to stop capture.
Press CTRL-C to terminate capture.
```

Execute action on click Enter.

Depending on selected action, capture will be saved to file and decoded if requested.

Switch to High Bandwidth USB transport with Trace Profile

For USB connection to targets, a new profile is added to the trace configuration editor.

This setting configures platform to enable High Bandwidth streaming option using USB3.2.

[Show Advanced Trace Configuration view](#)**▼ Profiles**

Low power trace with Intel(R) DCI USB2 DBC

Low power trace with Intel(R) DCI USB2 DBC

High bandwidth trace with Intel(R) DCI USB3 DBC

BIOS reserved System Memory

Intel(R) Trace Hub Memory

Selecting this profile will check the high bandwidth configuration option in the editor:

Platform Configurations

- Apply low power overrides
- Block and Drain
- High-Bandwidth Transport
- TCSS USB3DBC Disconnect in PkgC10

The new configuration will be applied after the editor is saved and the event is passed via the Target Connection Assistant (TCA) to OpenIPC for configuring the platform connection.

Note:

- For now this option is made available only on trace configurations for Meteor Lake platforms.
- Please note that this configuration may not be fully applied, if OpenIPC does not enable it.
- The high bandwidth setting might incur side-effects on the target (e.g. prevent to enter low-power sleep states)

Appendix X: Trace Message Viewer Preview

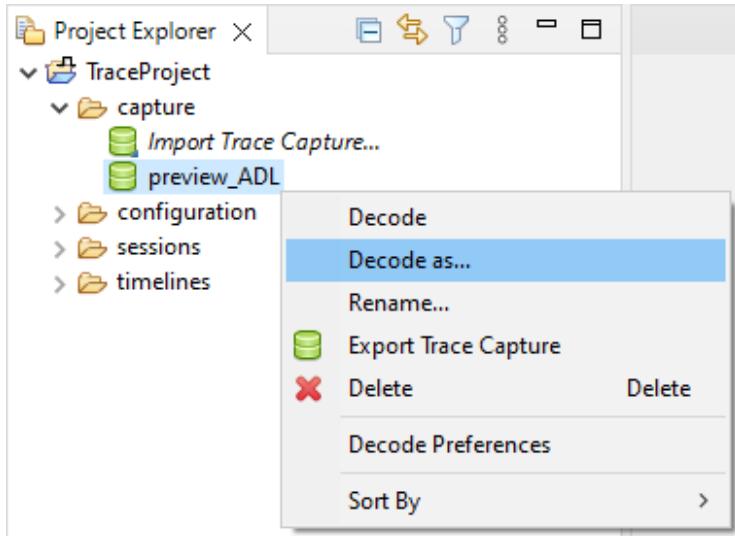
This chapter describes the preview version of the Message Viewer, which is the successor of [Message View](#).

Get Started

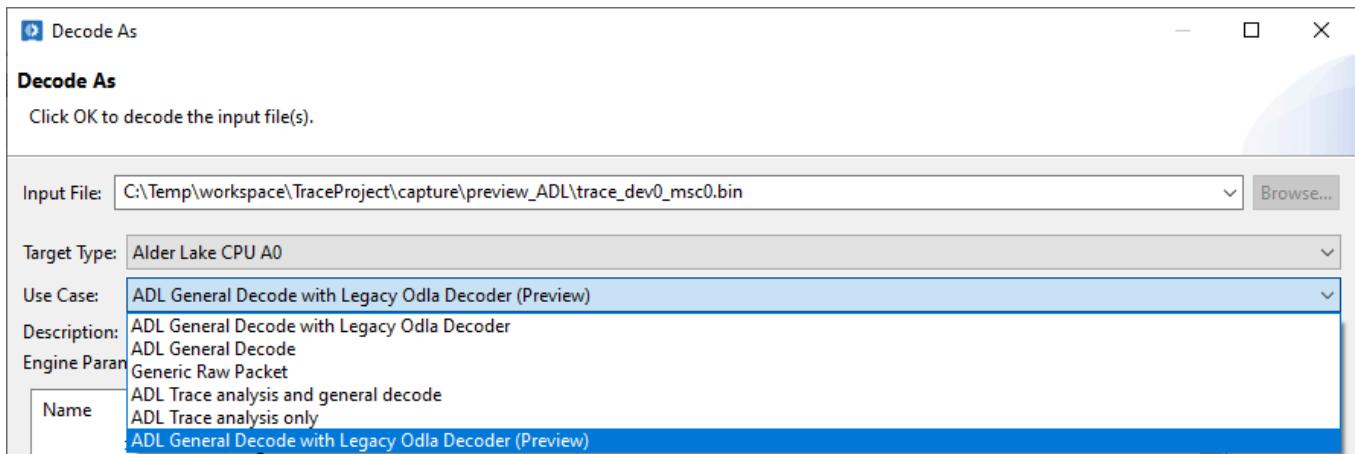
The Message Viewer displays a table with all decoded trace messages, that are previously captured from the target. You can get the capture file by [capturing trace](#) or by [importing pre-captured trace file](#).

Display the capture data in the Message Viewer by following the steps below:

1. Right-click on the capture file in the **Project Explorer** and select **Decode as...**



1. In the **Decode As** dialog, click on the **Use Case** dropdown and select the general decode that ends with **(Preview)**.
2. Click **OK** to decode and display the data in the Message Viewer.



Analyze Trace Messages with Message Viewer

The Message Viewer provides the functionalities to ease the analysis process.

Message Viewer

With the new message viewer you are able to open and view multiple sessions simultaneously in different viewers.

You can also open the same session in a new viewer with the context menu option:

Right-click -> Context Menu -> Open in a new message viewer

Table

Row Header

The first two columns of the table are reserved for row header. The first column contains the [Annotation](#) functionality. The second column is the message index, which displays the order of the messages shown.

Column

The columns in the Message Viewer are representing a set of trace fields. See customizing [column](#) for more reference.

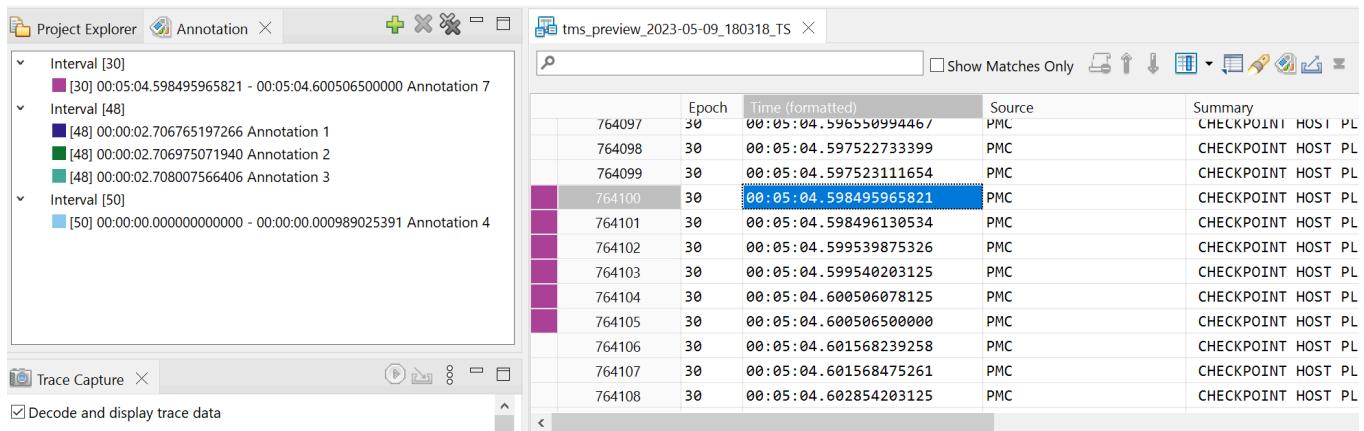
Annotation

Annotation supersedes the [mark feature](#). You can annotate a message by clicking the message directly on the row header, on the left side of the message index. Remove the annotation by clicking the message on the row header again.

You can also annotate multiple messages by following the steps below:

1. Select the messages you want to annotate.
2. Right-click on one of the selected messages to open the context menu.
3. Select **Annotation > Add...** to add a new annotation.

You can view all annotations by opening the **Annotation** view by selecting the  [Open annotation view](#) button in the toolbar.



Epoch	Time (formatted)	Source	Summary
30	00:05:04.598495965821	PMC	CHECKPOINT HOST PL
30	00:05:04.597522733399	PMC	CHECKPOINT HOST PL
30	00:05:04.597523111654	PMC	CHECKPOINT HOST PL
30	00:05:04.598495965821	PMC	CHECKPOINT HOST PL
30	00:05:04.598496130534	PMC	CHECKPOINT HOST PL
30	00:05:04.599539875326	PMC	CHECKPOINT HOST PL
30	00:05:04.599540203125	PMC	CHECKPOINT HOST PL
30	00:05:04.600506078125	PMC	CHECKPOINT HOST PL
30	00:05:04.600506500000	PMC	CHECKPOINT HOST PL
30	00:05:04.601568239258	PMC	CHECKPOINT HOST PL
30	00:05:04.601568475261	PMC	CHECKPOINT HOST PL
30	00:05:04.602854203125	PMC	CHECKPOINT HOST PL

Hovering over annotation in the table gives you a list of annotations for that particular row in the tooltip.

The screenshot shows the Project Explorer and Annotation views in a software interface. The Project Explorer view on the left lists intervals [30], [48], and [50] with their respective annotation details. The Annotation view on the right displays a table of annotations with columns for ID, Start Time, End Time, and Duration. A context menu is open over the annotation at index 7, showing options like Edit, Remove, and Delete.

ID	Start Time	End Time	Duration
764096	00:05:04.598495965821	00:05:04.600506500000	Annotation 7
764097	00:05:04.595491488933	00:05:04.599539875326	Annotation 6
764098	00:05:04.600506078125	00:05:04.604562743816	Annotation 8
764099	00:05:04.593569930664	00:05:04.599540203125	Annotation 10
764100	00:05:04.591528572266	00:05:04.594530769857	Annotation 1
764101	00:05:04.598495965821	Annotation 10	Annotation 2
764102	00:00:02.706765197266	Annotation 1	Annotation 3
764103	00:00:02.706975071940	Annotation 2	Annotation 4
764104	00:00:02.708007566406	Annotation 3	Annotation 5
764105	00:00:00.000000000000	00:00:00.000989025391	Annotation 6

You can edit the annotation from the menu option in the context menu in the message viewer.

You can also edit it from the **Annotation** view:

1. Select the annotation you want to edit.
2. Right-click on the selected annotation.
3. Select **Edit...** to open the configuration dialog.

The screenshot shows the Annotation view with a context menu open over the annotation at index 1. The menu includes options for Edit, Remove, and Delete. The annotation at index 1 is highlighted in blue.

Annotation 1 details:

- [48] 00:00:02.706765197266 Annotation 1

4. Press **Show More >>>** to fine-tune the annotation.

Edit Annotation

X

Edit Annotation

Annotate across a range of timestamps

Interval:	48	Start timestamp (ps):	2768959988932	End timestamp (ps):	2768959988932
Name:	Annotation 1			Color:	<div style="background-color: #00008B; width: 20px; height: 15px;"></div>
Description:	<input type="text"/>				
Show Less <<<					
Label:	<input type="text"/>	Label style:	Fit	Label alignment:	Axis
Start y-axis:	0.0	End y-axis:	0.0	Line style:	Dotted
Anchor style:	Circle	Anchor position: <input checked="" type="checkbox"/> Top left <input checked="" type="checkbox"/> Top right <input checked="" type="checkbox"/> Bottom left <input checked="" type="checkbox"/> Bottom right			
<input checked="" type="checkbox"/> Show tooltip		<input checked="" type="checkbox"/> Fill annotation background		Background opacity:	<div style="width: 100px; height: 10px; background-color: #00008B; margin-left: 10px;"></div>
?			Save	Cancel	

5. Press **Save** to apply the modification.

Note

The annotation is based on a timestamp. Annotating a message will annotate other messages with the same timestamp.

	Epoch	Timestamp	Summary	MasterID
	6721	0x01	IN(0x000003FE)	0x0018
	6722	0x01	583290918567279	0x0018
	6723	0x01	583290918567279	0x0018
	6724	0x01	600883104611695	0x0018
	6725	0x01	583290918567279	0x0018
	6726	0x01	600883104611695	0x0018
	6727	0x01	583290918567279	0x0018
	6728	0x01	600883104611695	0x0018

Clicking an Annotation in the Annotation view automatically selects the corresponding message for the Annotation in the Message view. This sync of selection is only possible when the messages are ordered chronologically (for example ordered by timestamp) in the Message view, since Annotations are identified by the timestamp of messages.

Font

Message viewer uses the default eclipse text font which can be changed from eclipse preferences.

General -> Appearance -> Colors and Fonts -> Basic -> Text Font

Quick Search

The Message Viewer provides a functionality to quickly find matching text and highlight it in the Message Viewer.

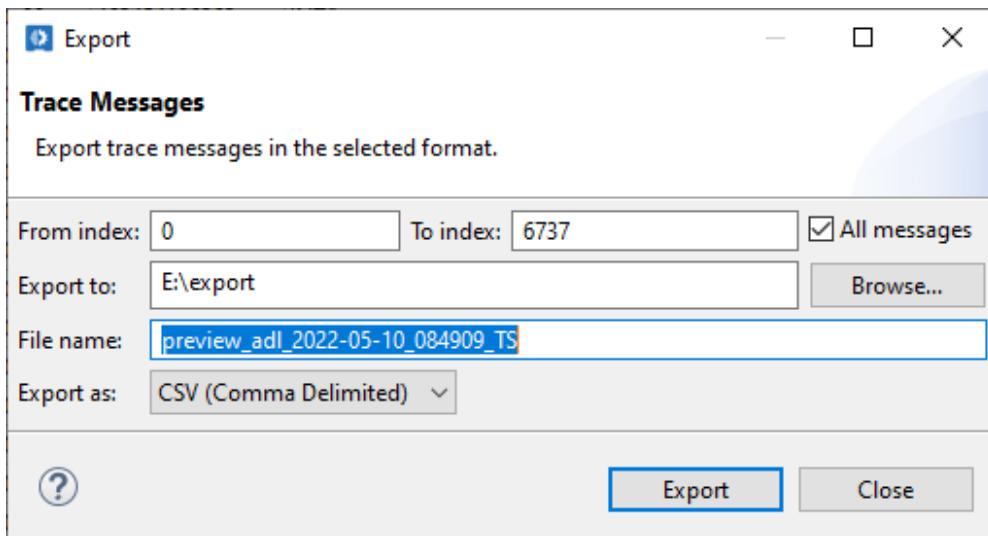


- Type the text in the search bar in the toolbar. The Message Viewer will highlight the matching text as you modify the text in the search bar. Press **Enter** to highlight the rows containing the matched text. The green bar underneath indicates the search progress.
- **Show Matches Only** **Show matches only:** Check the box to display only the matched messages.
-  **Reset search results:** Reset the search results in the Message Viewer. Select this button to remove the highlights in the table.
-  **Select the previous match:** Go to the previous match in the Message Viewer and select the row.
-  **Select the next match:** Go to the next match in the Message Viewer and select the row.

Export

The trace messages can be exported to a CSV or TXT file through the following steps:

1. Right-click on the message viewer to open context menu.
2. Select **Export...** menu to open the export dialog.
3. In the export dialog, you can specify the message index you want to export, the location, the file name and the format.
4. Click **Export** to start exporting the messages.



Column

You can customize the columns in the Message Viewer to display specific column and its contents.

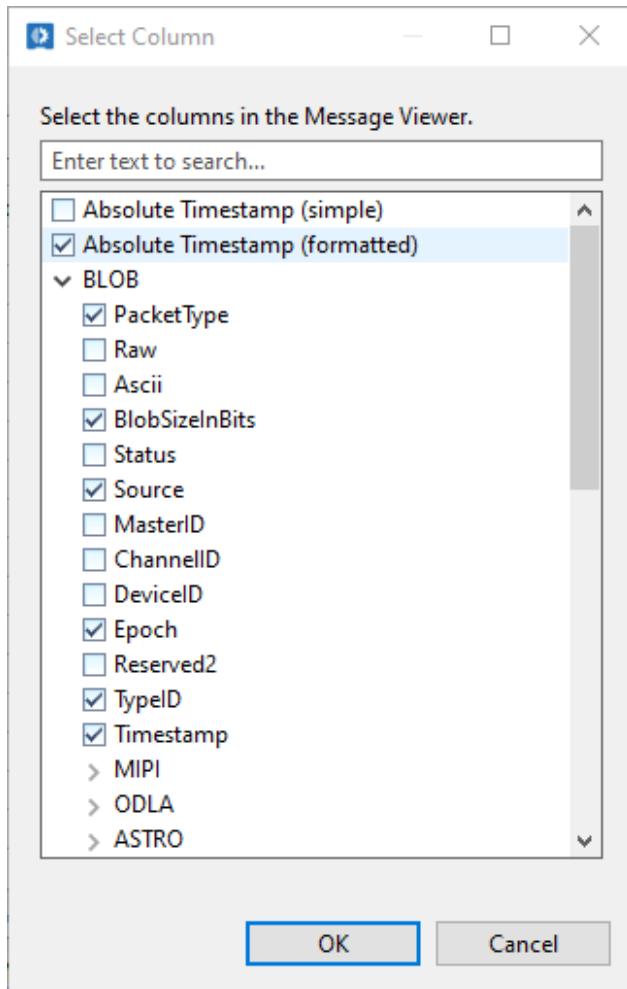
Column Selection

You can add a new column with the following steps:

1. Open the **Select Column** dialog by either:

- Clicking the **Add a column** button on the toolbar 
- Or right-clicking on the table to open the context menu and choosing **Select columns...** menu

2. In the **Select Column** dialog, check the packet field you want to show as column.

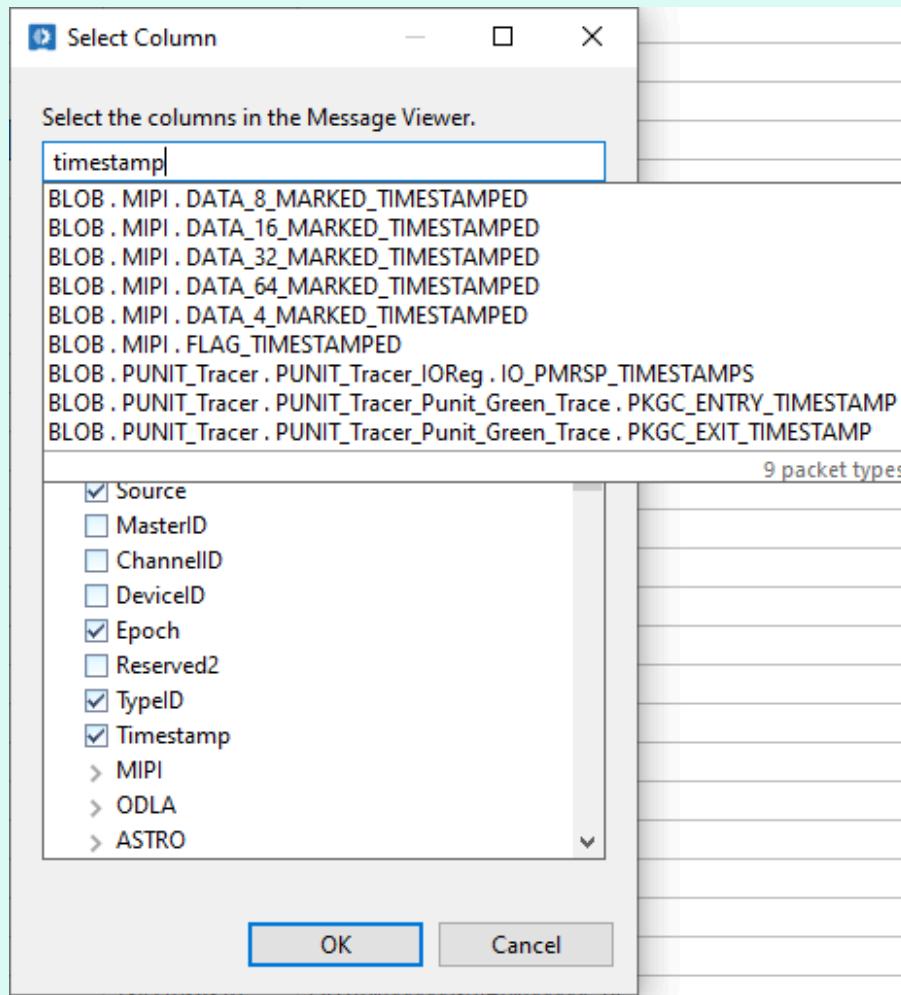


The Message Viewer is equipped with extended timestamp formats. To view the different formats, select the columns through the column selection dialog:

- Normalized time: The timestamp relative to the beginning of each interval.
- Absolute time: The recorded timestamp.
- Relative time: The Timestamp relative to a specified timestamp. To activate this format, select on the message, which the timestamp should be relative to. Right-click on the message and select **Relative to this message**.
- Accumulated time: The normalized time added to the duration of previous intervals.

Tip

Use the **Search** feature to get some suggestions of the packet types. Expand the packet type and check the packet field you want to include in table.



Column Configuration

The column display of the Message Viewer is customizable.

You can reorder the columns by drag-and-drop the column to the preferred place.

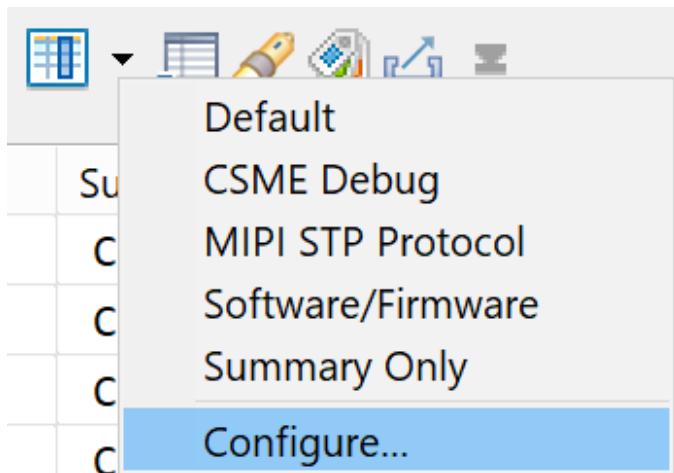
You can change the value format by right-clicking the column you want to modify, select **Change value format** in the context menu, and select the desired format.

	Epoch	Time (formatted)	Source	Summary
0	0x00	00:00:00.000000000000	PMC	CHECKPOINT
1	0x00	aa·aa·aa aaaaaaaa449511	PMC	CHECKPOINT
2	0x00		Copy	>
3	0x00		Go to index...	Ctrl+G
4	0x00		Annotation	>
5	0x00		Search	>
6	0x00		Export...	
7	0x00		Open in a new viewer	
8	0x00		Change column format	>
9	0x00		Relative time to this message	
10	0x00		Select columns...	
11	0x00		Configure columns...	
12	0x00		Column presets	>
13	0x00		Help	F1
14	0x00			
15	0x00	00:00:00.006939500000	PM	
16	0x00	00:00:00.007991194335	PM	
17	0x00	00:00:00.007991380208	PM	
18	0x00	00:00:00.008959361002	PM	
19	0x00	00:00:00.008959527669	PM	(default)

You can also further customize the columns with the **Column Configuration dialog**.

You can open the dialog by either:

- Right-clicking in the Message Viewer and select **Configure Columns...** from the context menu.
- Or by expanding the drop-down button in the toolbar and select **Configure Columns...** from the menu.



In the **Column Configuration** dialog, you can add a new column, reorder, adjust the width and value format of each column.

! Tip

Use the Search feature to get some suggestions of the packet fields.

The screenshot shows the 'Message Viewer' window with a 'Configure Column' dialog box overlaid. The dialog box allows users to configure column settings for the current message viewer. It includes a table for setting column names, widths, and formats, and buttons for OK, Cancel, Select..., Add, Up, Down, and Remove.

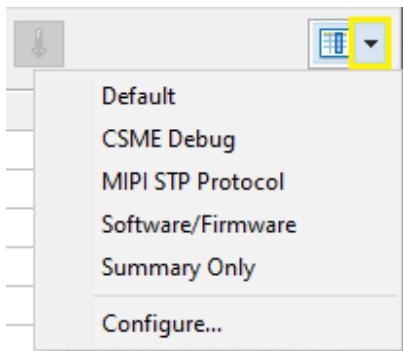
Name	Width	Format
Epoch	50	(default)
Timestamp	100	(default)
Summary	185	(default)
PacketType	180	(default)

Below the table, there is a list of messages with columns for Epoch, Timestamp, Summary, and PacketType. The first few messages are:

Epoch	Timestamp	Summary	PacketType
0x01	78419554107	IN(0x00001808)	BLOB.AET.CORE.PORT_IN_ADDR
0x01	78419559200	IN(0x00001808)=0x009F5477	BLOB.AET.CORE.PORT_IN
0x01	78419641354	OUT(0x00001830)=0x00000002	BLOB.AET.CORE.PORT_OUT
0x01	78421036810	OUT(0x00000080)=0x00000C14	BLOB.AET.CORE.PORT_OUT
0x01	78421226253	OUT(0x00000080)=0x00000C5F	BLOB.AET.CORE.PORT_OUT
0x01	78421414943	OUT(0x00000080)=0x00000C26	BLOB.AET.CORE.PORT_OUT
0x01	78422924571	OUT(0x00000080)=0x00000C6F	BLOB.AET.CORE.PORT_OUT

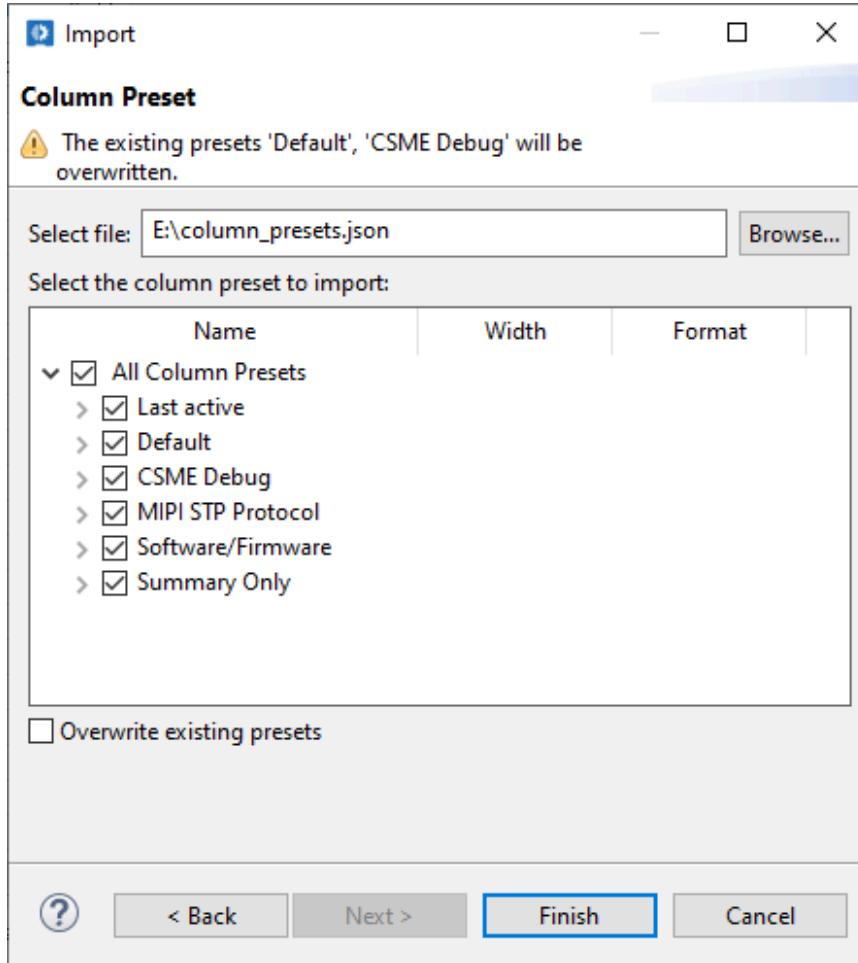
Column Preset

The column preset feature allows you to save column settings and load them easily. In the Message Viewer, expand the drop down in the toolbar to show the pre-defined column presets. Select the column preset to load and apply it to the current Message Viewer.



To import column presets, do the following:

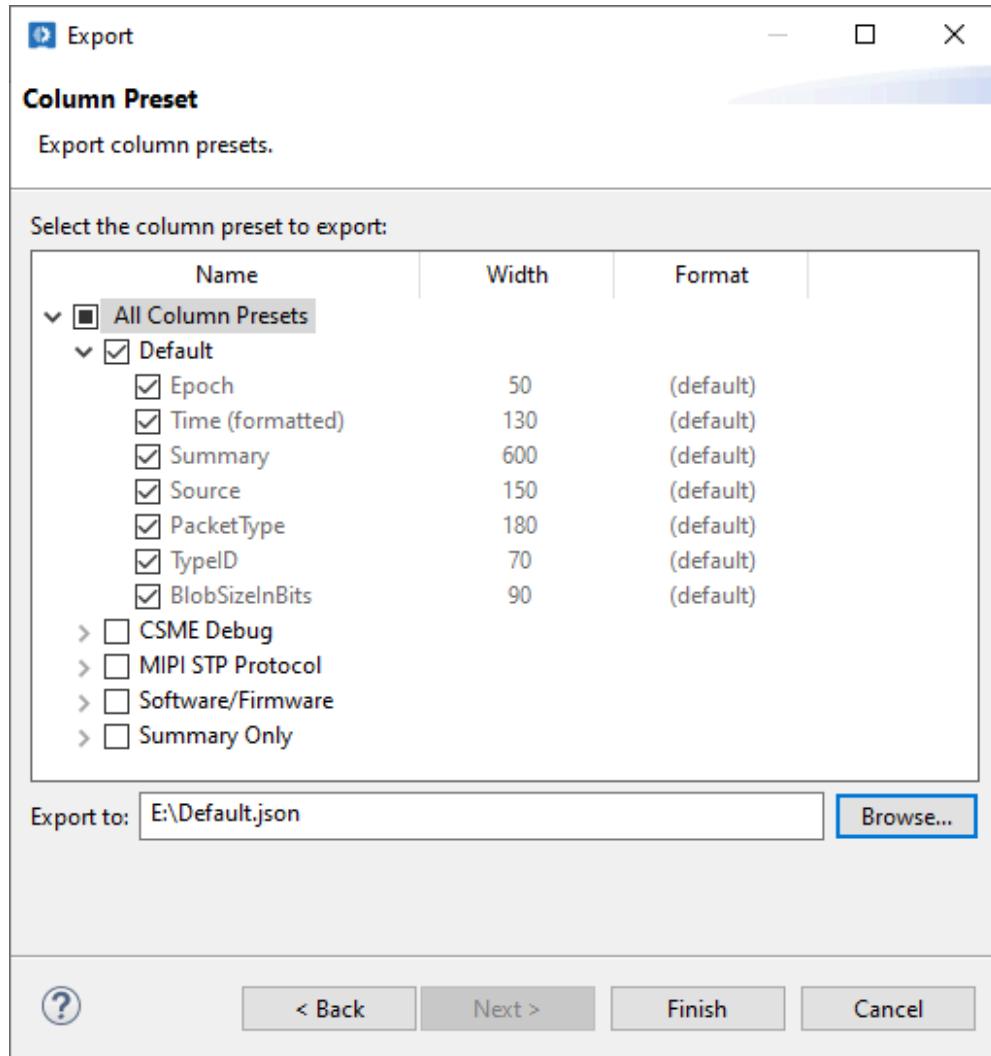
1. On the menubar, click **File > Import...** to open the import dialog.
2. Select **Column Preset** to open the import column preset dialog.
3. Click **Browse...** to open the file selection dialog.
4. Select the **.json** file you want to import and click **Open**.
5. Check the presets you want to import.
6. In case of existing column presets with the same name, a warning message will appear on the top of the dialog.



7. Uncheck the conflicted presets if you want to keep the existing ones. Otherwise, check the **Overwrite existing presets** to replace the existing ones.
8. Click **Finish** to import the column presets.

To **export** existing column presets, do the following:

1. On the menubar, click **File > Export..** to open the export dialog.
2. Select **Column Preset** to open the export column preset dialog.
3. Check the presets you want to export.
4. Click **Browse...** to specify the location of the exported file.
5. Click **Finish** to export the column presets.

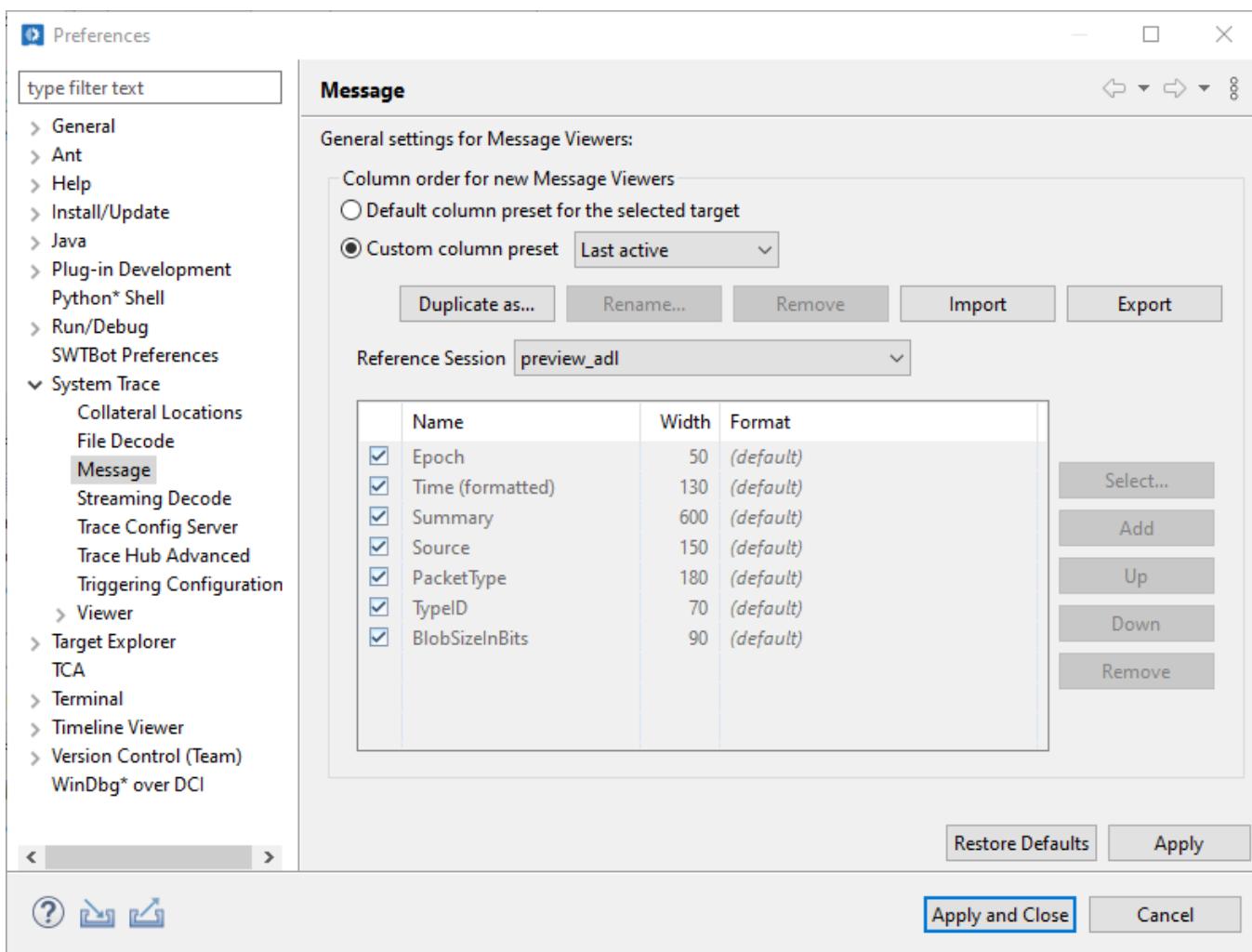


You can also open the import and export dialogs from the Column Configuration and Column Preference dialogs.

Column Preference

You can change the column order for new opened Message Viewers.

1. Select **Window > Preferences** to open the **Preferences** dialog box.
2. Navigate to **System Trace > Message** to change the setting.
3. Click **Apply** to save your changes or **Apply and Close** to save changes and close the dialog.



Timeline Viewer

The [Timeline Viewer](#) is available for both the [Message View](#) and the Message Viewer.

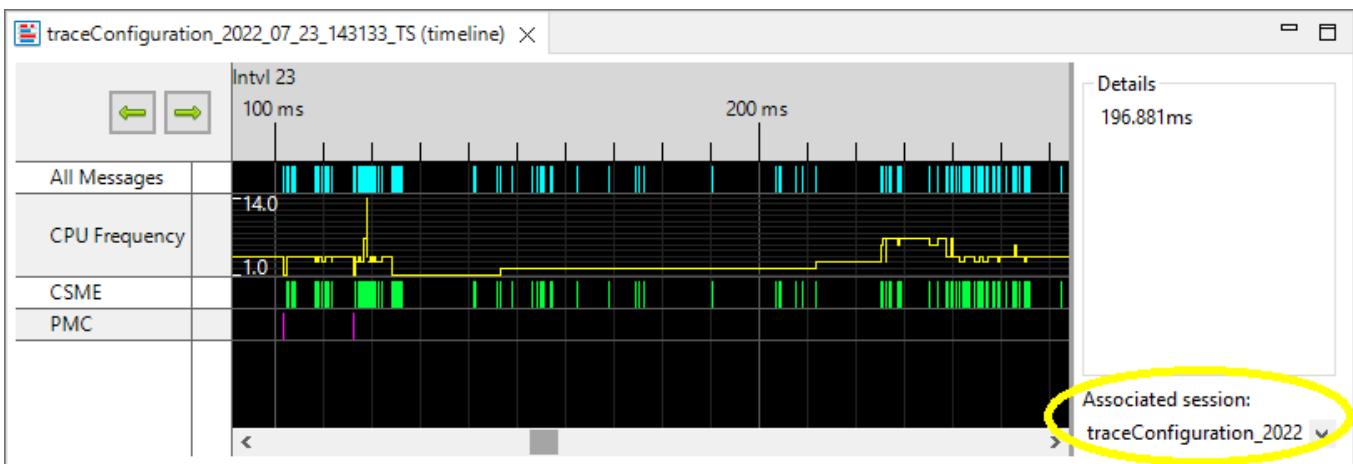
Timeline Viewer

The [timeline](#) capability is supported for both the service-based Message Viewer and the existing [Message View](#).

Session

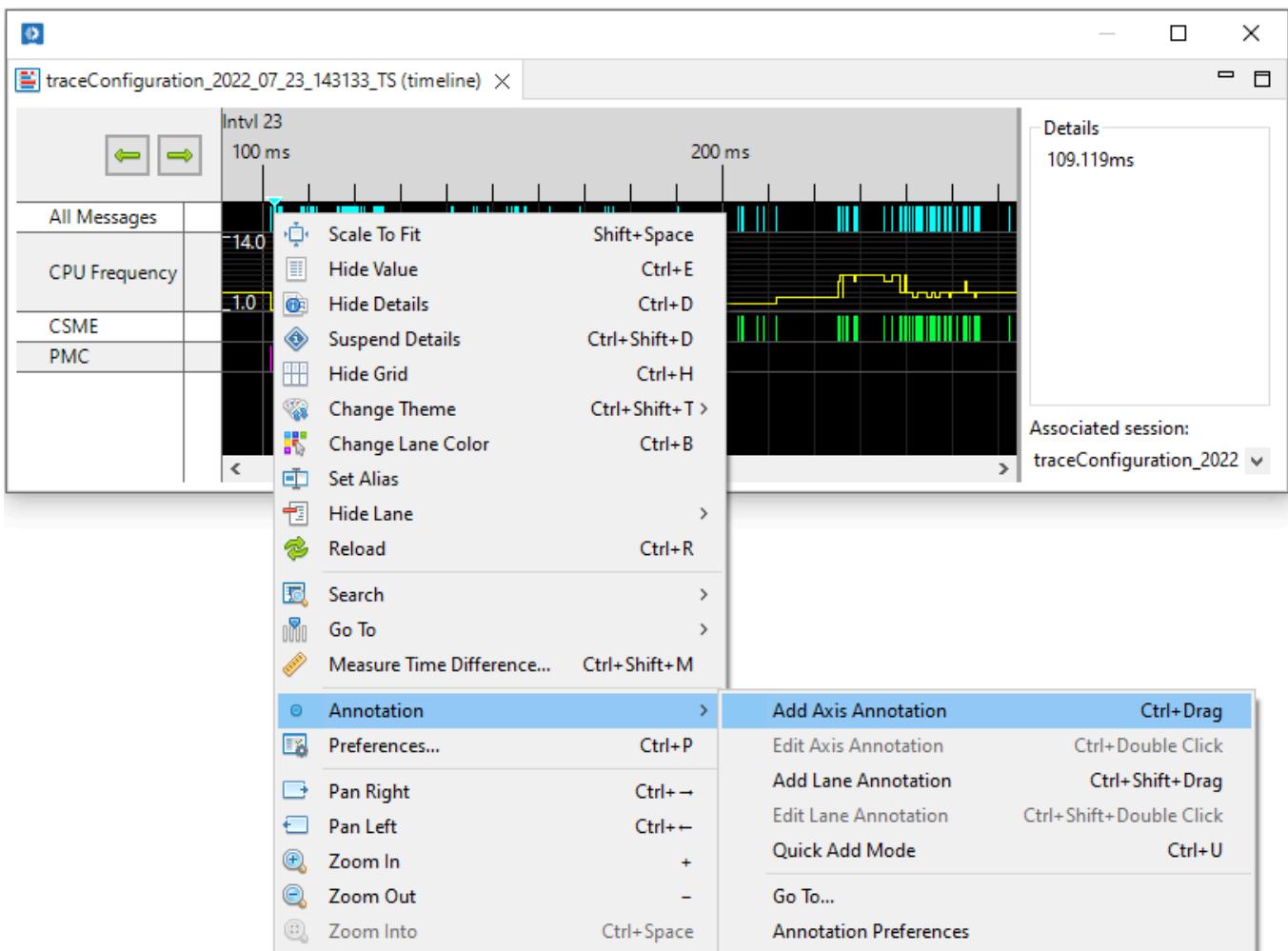
One of the enhancements added along with the release of the Message Viewer is the modularity between these two viewers. This allows easier navigation and comparison between different data without having to open the session file. If a trace session is opened in the existing Message View, it will become the associated session for all opened Timeline Viewers. This is consistent with the established behavior.

To change the associated session for that timeline, click on the drop-down button in the bottom right corner and select the session name that you wish to load.



Annotation

An annotation can be an option to mark a specific area in the Timeline Viewer. To annotate, right-click in the timeline and select **Annotation**.



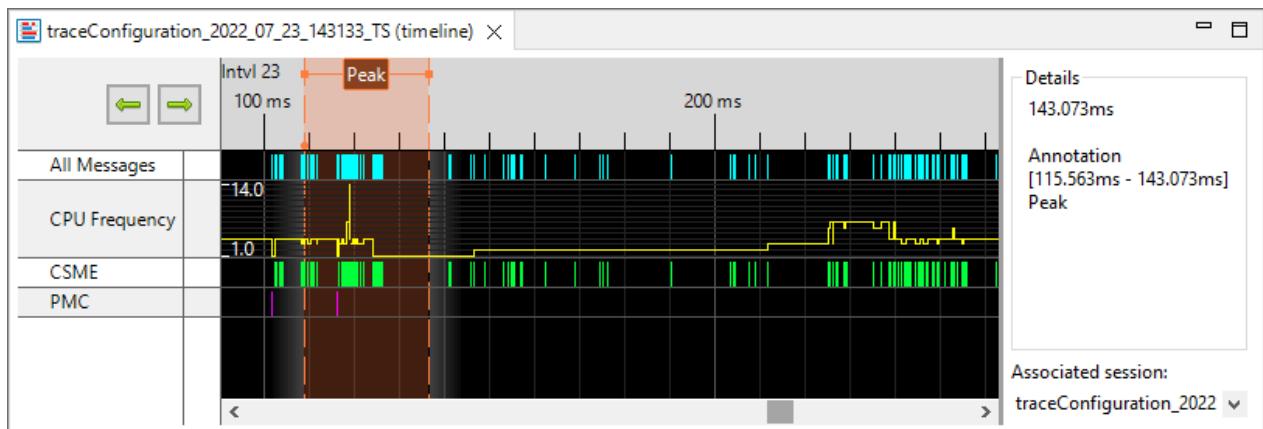
There are 2 types of annotations:

- **Axis Annotation:** Select this option to annotate a time region across all lanes. You can also open the dialog by pressing **Ctrl** while dragging through the region you want to annotate and release.

Add Axis Annotation

Interval:	23	Start timestamp (ps):	115562868652	End timestamp (ps):	143072664942
Name:	Annotation		Label: Peak		
Description:					
Index: 9031 [23]+00:00:00.108736918945					
Color:		Label style:	Fit label	Label alignment:	Axis
<input checked="" type="checkbox"/> Show tooltip	<input checked="" type="checkbox"/> Fill annotation background	<input checked="" type="checkbox"/> Line style: Dashed			
<input checked="" type="checkbox"/> Update default	Background opacity:				
OK Cancel					

Press **OK** to see the axis annotation appear in the Timeline Viewer.

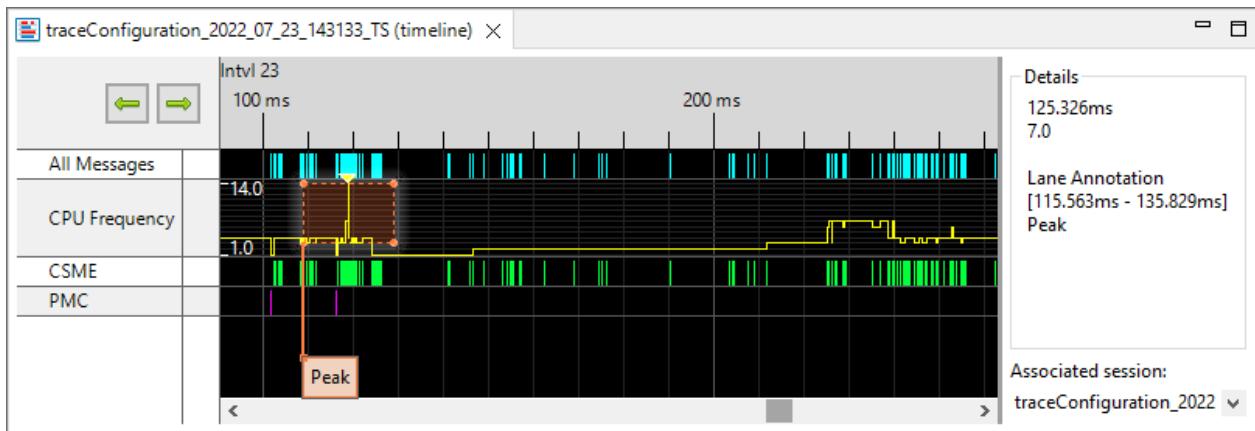


- Lane Annotation:** Select this option to annotate a time region on a specific lane. This can cover a range of selected values on the y-axis, if the lane type supports it. You can also open the dialog by pressing **Ctrl** and **Shift** while dragging through the area and release.

Add Lane Annotation

Interval:	23	Start timestamp (ps):	115562868652	End timestamp (ps):	135829168689
		Low:	3.260869565217391	High:	14.0
Name:	Lane Annotation				
Description:	Peak				
Index: 9031 [23]+00:00:00.108736918945					
Color:		Anchor style:	Circle	Anchor position:	<input checked="" type="checkbox"/> Top left <input checked="" type="checkbox"/> Top right <input checked="" type="checkbox"/> Bottom left <input checked="" type="checkbox"/> Bottom right
<input checked="" type="checkbox"/> Show tooltip	<input checked="" type="checkbox"/> Fill annotation background	<input checked="" type="checkbox"/> Line style: Dotted			
<input checked="" type="checkbox"/> Update default	Background opacity:				
OK Cancel					

Press **OK** to see the lane annotation appear in the Timeline Viewer.



Crash Log

The Crash Log feature of Intel® System Debugger allows you to decode and analyze the Crash Log files created by the Intel® Crash Log Technology platform-level feature. The Intel Crash Log Technology feature captures and preserves hardware state, allowing it to survive a set of platform reset types. Intel System Debugger includes Crash Log Viewer, a command-line decoder and analyzer for Crash Log files, as well as a Graphical User Interface (GUI) for the command-line decoder.

Intel(R) Crash Log Framework

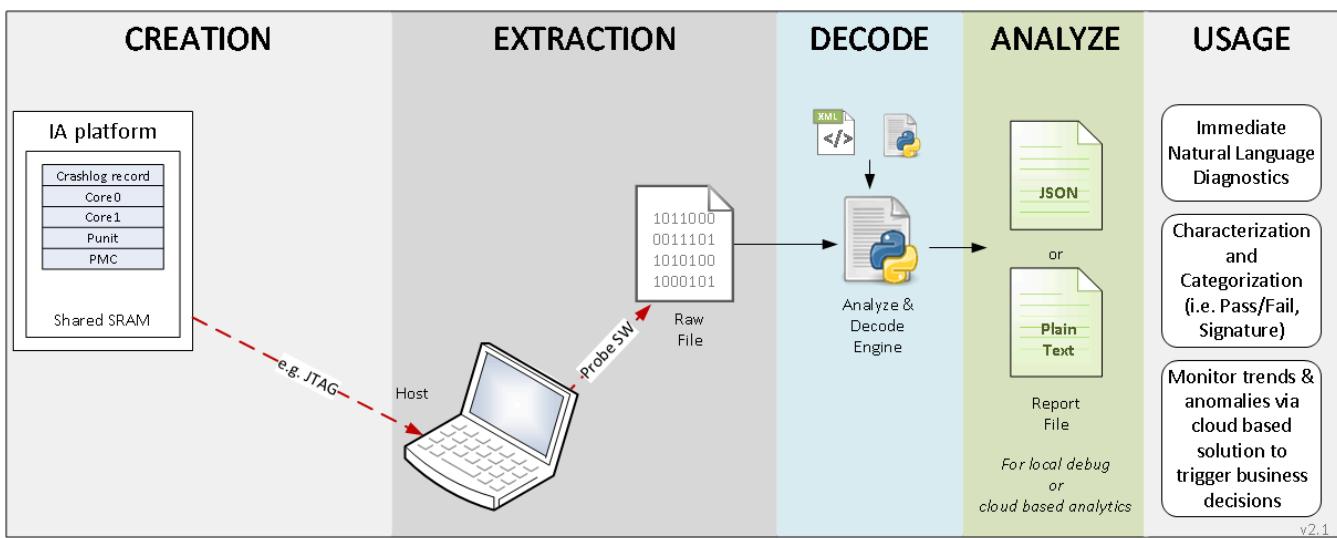
Overview

The Intel(R) Crash Log Framework provides the ability to create or update software to work with Intel(R) Crash Log Technology used to help in the triage and failure analysis process.

Intel(R) Crash Log Technology captures and preserves hardware state so that it survives a set of platform reset types. It creates a crash log that can be used by humans and external systems.

The Intel(R) Crash Log Framework includes software components to decode and analyze the log, providing immediate natural language diagnostics of the crash. Crash data survives resets, so System Firmware can notify the Operating System and save the log to permanent storage.

Below is a diagram showing the Crash Log flow and usage.



Frequently Asked Questions

Am I required to enable Crash Log?

The Crash Log collection is enabled by default on the supported platforms. Just extract the data after the crash, or view the Crash record extracted by the OS-based extraction component.

Crash Log feature can be enabled or disabled in the System Firmware settings.

Are Crash Logs stored for all crashes?

No. The exact details depend on the Crash Log capabilities specific to the Intel architecture based platform.

Do Crash Logs persist for all crashes?

No. Unfortunately some scenarios trigger a power-cycle and no log of the crash is saved.

What is in a Crash Log Report?

The *Crash Log Report* contains data and analysis. The scope of the data is specific to the platform. In general, the trigger of the crash, state of platform-based components, and analysis information to help diagnose the crash are included in the report.

How do I get the Crash Log?

The exact procedure for extracting a Crash Log into a file depends on the platform model, firmware support and operating system.

- On early platform boot phases or when the firmware does not support the *Crash Log Extraction*, the Crash Log may be obtained by reading the Crash Log hardware storage using out-of-band extractions methods (JTAG, eSPI, ...).
- When the firmware has already extracted the Crash Log from the hardware storage, the Crash Log may be obtained by reading the BERT structure of the platform ACPI tables (in-band extraction).

For more details about the Crash Log extraction procedures for a specific platform, please contact your Intel representative.

Prerequisites

This section discusses the prerequisites for obtaining a Crash Log.

Platform

Platform must support Crash Log as a platform feature. When available, the feature is enabled by default but may be disabled by the IAFW.

Extraction

- Debug/Test Connection ¹ (e.g., Intel(R) Direct Connect Interface (Intel(R) DCI))
- Debug/Test Software Framework ¹ (e.g., OpenIPC)

Decoding

- Python* 3.x ²
- pip ²
- wheel ²

[1] ([1](#), [2](#)) May be provided by third-party or by Intel outside of Intel(R) Crash Log Framework.

[2] ([1](#), [2](#), [3](#)) Provided by third-party.

Installation

The Intel(R) Crash Log Framework is distributed as a wheel file. The package contains the decoder engine and the platform-specific collateral. Before installing the package, make sure a Python* Wheel package is installed in the system:

```
$ pip install wheel
```

The installation of the package can then be done by using `pip`. The following command automatically installs the required dependencies and overrides any previously installed versions of the decoder.

```
$ pip install intel.crashlog-...200-py3-none-any.*
```

If used behind a proxy, `pip` needs to be configured to use it with the `--proxy=https://user@proxy_addr:port` option.

Verifying Setup and Usage

This section provides examples of expected behavior of the Intel(R) Crash Log Framework.

Note: The Intel® Crash Log Framework package installs the `intel_crashlog` executable in the `<PYTHON_HOME>Scripts` directory on Windows* and `<PYTHON_HOME>/bin` on Linux* and macOS*. It is recommended to add this directory to the PATH environment variable.

If the package is correctly installed, the following command should print the version number of the Intel(R) Crash Log Framework:

```
$ intel_crashlog --version
Intel(R) Crash Log Framework, Version 3.x Beta
Copyright (C) 2017-2019 Intel Corporation. All rights reserved.
```

The command-line tool consists of the `intel_crashlog` executable. It takes as parameter a Crash Log file and an action to perform on this input.

```
$ intel_crashlog <command> <crashlog_file>
```

The complete list of supported commands can be obtained by using:

```
$ intel_crashlog --help
```

Some Crash Log samples are provided in the framework. They are installed in the Python's site packages directory and can be used to verify the Intel(R) Crash Log Framework setup:

```

$ cd /path/to/python/site-packages/intel/crashlog/collateral
$ cd intel/crashlog/collateral/ICL/all/all/white/crashlog/tests/samples
$ intel_crashlog analyze ./3strike_timeout.crashlog
Three-strike timeout - TIMEOUT.MLC.WDT
=====
Three-strike timeout detected.
Machine-Check Address: 0xFFFFF802ACA8FE34.
Outstanding transaction(s):

- Thread 1 access (Memory read) to address: 0xfe1fe204, State: 6
...

```

Obtaining a Crash Log

Enabling the Crash Log Feature

The System Firmware has the ability to control the Crash Log platform feature. An entry in the IAFW settings allows the Crash Log feature to be configured (depending on the platform, it can usually be found in the *Debug Settings* page)



Extraction

The Intel(R) Crash Log Framework provides a set of extraction scripts. They can be invoked by using the `extract` subcommand.

```
$ intel_crashlog extract <method> <output_file>
```

- `<method>` : extraction method to use. Multiple methods may be specified. If none is specified, the default extraction method is used.
- `<output_file>` : Crash Log Raw File to produce.

Out-of-Band Extraction

The out-of-band extraction methods use a debug connection for the extraction and are based on Intel(R) In-Target Probe (Intel(R) ITP) scripting. As most of the out-of-band extraction methods are defined per platform, it is recommended to explicitly specify the platform model using the `--product <product>`, `--variant <variant>`, and `--stepping <stepping>` options.

Depending on the platform, multiple out-of-band extraction methods may be available. One of the most common extraction method consists of reading the *Crash Log Storage* from the Memory-mapped input/output (MMIO) space.

For using the out-of-band extraction flow with a debug connection based on Intel(R) In-Target Probe, the location of the OpenIPC installation folder must be specified by the `IPC_PATH` environment variable.

Assuming that an OpenIPC process is already configured and running on the host, the default extraction method can be invoked by the following command:

```
$ intel_crashlog extract sample.crashlog
```

Note: This script requires the Crash Log Storage to be already mapped into the address space.

Out-of-Band Manual Trigger

It is sometimes useful to manually trigger the collection of a Crash Log. When an out-of-band trigger method is available in the platform under test, it can be called from the command line tool by using the `trigger` subcommand:

```
$ intel_crashlog trigger
```

If this command is completed successfully, a Crash Log is generated on the platform and can be extracted using the [Out-of-Band Extraction](#).

Decoding and Analyzing a Crash Log Raw File

Once the [Crash Log Raw File](#) has been extracted, it can be decoded and analyzed. This section provides examples of how to interface with the Crash Log Raw File using a command line interface or a Python* API.

Command-Line Interface

The command-line tool is implemented through the [intel_crashlog](#) executable. It aims to provide a simple way to invoke the functionality of the Intel(R) Crash Log Framework. The following sections describe how it can be used to interact with a pre-extracted Crash Log raw file.

Decoding

The [decode](#) subcommand produces a JSON-formatted output representing the values of the fields contained in the Crash Log. The output can be redirected to a file by using the [-o <file>](#) option. If none is specified, the JSON is printed in the standard output.

```
$ intel_crashlog decode /path/to/crashlog_file
{
    "crashlog_data": {
        "PMC": {
            "art_timestamp": "0x4e476eec781",
            "crashlog_reason": "0x8",
            "crashlog_version": "0x1001001",
            "crashlog_completion_status": "0x800001ff1",
            "pmc_fw_engineering_version": "0x0",
            "pmc_fw_release_version": "0x4e",
            "reset_sequence_id": "0x0",
            ...
        },
        ...
    }
}
```

The decoding of two Crash Log files can be compared with the [diff](#) command:

```
$ intel_crashlog diff example1.crashlog example2.crashlog
--- example1.crashlog
+++ example2.crashlog
@@ -21,5 +21,5 @@
-         "art_timestamp": "0x6737b32eL",
+         "art_timestamp": "0x1be83d64L",
```

Analysis

The complete analysis of a Crash Log file can be displayed with the following command:

```
$ intel_crashlog analyze /path/to/crashlog_file
```

The goal of the analysis is to provide diagnostic information to characterize the crash based on the Crash Log data.

The output of this command is formatted in Markdown* and adopts the following structure:

```
Subsystem Name - BUCKET
=====

```

```
Subsystem analysis
```

where:

- **Subsystem Name**: Name of the system analyzed in the section.
- **BUCKET**: The field representing the category of the analysis. It can be:
 - **NO_ERROR_DETECTED**: the subsystem analyzed is healthy and is highly unlikely to be involved in the platform crash.
 - **UNKNOWN**: the analysis could not make a conclusion about the state of the subsystem.
 - **FAILED**: Analysis failed due to an internal error.
 - Other values represent the type of the error detected in the subsystem.
- **Subsystem analysis**: A Markdown-formatted explanation of the state of the subsystem during the platform crash.

Display the summary of the analysis with the **summary** command. The output is expected to be empty if no platform errors have been identified in the Crash Log file:

```
$ intel_crashlog summary /path/to/crashlog_file
```

Triage

Since the **analyze** subcommand is intended to produce a report in a natural language, it may not be suitable for automatic processing. For a given Crash Log raw file, the **triage** subcommand aggregates the buckets returned by the analyzers and prints the most relevant ones, sorted by level

of severity:

```
$ intel_crashlog triage /path/to/crashlog_file
HW_DETECTED_DETECTIED.PCIE.TL
TIMEOUT.PUNIT.DISPLAY
```

When a Crash Log sample does not report any error, the `NO_ERROR_DETECTED` bucket will be returned by the `triage` command.

```
$ intel_crashlog triage manually_triggered.crashlog
NO_ERROR_DETECTED
```

Using a Collateral Patch

The Intel(R) Crash Log Framework can be dynamically extended by using *Collateral Patches*. These are occasionally used to provide a support for an uncommon or not yet officially supported Crash Log layout in the mainstream release.

The typical steps to use a collateral patch are:

1. If archived, extract the collateral patch. Example:

```
$ unzip ICPN_patch_rev02.zip
$ tree ./ICPN_patch_rev02
./ICPN_patch_rev02
`-- ICP
    |-- N
    |   '-- all
    |       '-- white
    |           '-- crashlog
    |               '-- decode-defs
    |                   |-- PMC
    |                       |-- 1
    |                           '-- layout.xml
    |                       '-- 2
    |                           '-- layout.xml
    |                   '-- PMC_FW_Trace
    |                       |-- 1
    |                           '-- layout.xml
    |                       '-- 2
    |                           '-- layout.xml
    '-- all
        '-- all
            '-- white
                '-- crashlog
                    '-- extractors
                        '-- tap2sb.py
```

2. Specify the location of the extracted patch using the `--ct` option before the `command` argument:

```
$ intel_crashlog --ct <patch> <command> <crashlog_file>
```

Example:

```
$ intel_crashlog --ct ./ICPN_patch_rev02 info /path/to/crashlog_file
- 0000-027f: PMC_FW_Trace - ICP/N (RT: 0x02, PID: 0x005, Rev:0x02)
```

Miscellaneous Commands

List available analyzers for the specified Crash Log file:

```
$ intel_crashlog list /path/to/crashlog_file
global_reset
=====
- Path: CNP\all\all\red\crashlog\analyzers\global_reset.py
- Full name: Global Reset
- Full version: 1.4.0
- Description: Analyze global reset cause
...
```

Manually specify which analyzer to run (here, *system state*):

```
$ intel_crashlog analyze /path/to/crashlog_file system_state
System State
=====
Current state: S0
Attempted state: UNKNOWN
System state transition: UNKNOWN
```

Display the cause of the Crash Log trigger:

```
$ intel_crashlog analyze /path/to/crashlog_file crashlog_reason
PMC Crashlog Reason - NO_ERROR_DETECTED.CPU_TRIG
=====
This Crash Log was triggered by the CPU.

Punit Crashlog Reason - NO_ERROR_DETECTED.MANUAL.PUNIT
=====
Punit Crash Log has been manually triggered.
```

Export the Crash Log Report of the analysis to JSON:

```
$ intel_crashlog report -o crashlog_report.json /path/to/crashlog_file
```

Python* API

The Intel(R) Crash Log Framework can also be called from Python*. This section presents a typical usage flow using the Python* API.

The Crash Log module can be loaded into a Python* environment using:

```
>>> import intel.crashlog as crashlog
```

Decoding

First, the Crash Log has to be loaded from a file into a `bytearray` object:

```
>>> crashlog_dump = crashlog.extract_from_file("gblrst.crashlog")
```

The `bytearray` object can then be decoded to a register object:

```
>>> crashlog_regs = crashlog.decode(crashlog_dump)
```

The decoded values can be accessed from the register object as regular Python attributes:

```
>>> crashlog_regs.PMC.hest1
hests1 = 0x2112080
    batlow_sts = 0x0
    crda_sent = 0x0
    cts = 0x0
    flex_sku_done = 0x1
    grst_2_host = 0x0
    host_prim_rst_sts = 0x1
    ...
>>> int(crashlog_regs.PMC.hest1)
34676864
>>> int(crashlog_regs.PMC.hest1.host_prim_rst_sts)
1
>>> crashlog_regs.PMC.revision
2
```

Analysis

Finally, the decoded registers can be analyzed. The `analyze` function takes the register object as argument and returns a `dict` containing the analysis and the metadata of the scripts used.

```
>>> crashlog_analysis = crashlog.analyze(crashlog_REGS)
>>> crashlog_analysis
{
    'System State': [
        {
            'Autorun': False,
            'Description': 'Analyze system state',
            'Full name': 'System State',
            'Path': 'path/to/system_state.py',
            'Version': '1.1.0',
            'html': u'...',
            'markdown': [
                'System State',
                '-----',
                '',
                '- `Current state`: S0'
            ],
            'bucket': 'TIMEOUT'
        }
    ],
    ...
}
```

Triage

Use the `triage` function to obtain the list of the buckets suitable for the analyzed Crash Log from the analysis report. The list is ordered according to the severity level of each bucket.

```
>>> buckets = crashlog.triage(crashlog_analysis)
>>> buckets
["HW_EXCEPTION_DETECTED.PCIE.TL.PL", "TIMEOUT.PUNIT.DISPLAY"]
```

Glossary

Collateral Patch

A set of files used to extend the Intel(R) Crash Log Framework in order to support additional hardware targets.

Crash Log Extraction

Refers to the reading or transformation of a Crash Log from a Crash Log Storage to a computer file (Crash Log Raw File).

Crash Log Raw File

The Crash Log Raw File is a computer file representation of the crash data.

Crash Log Report

JSON file produced by the Intel(R) Crash Log Framework containing the complete interpretation of a Crash Log Raw File.

Crash Log Storage

Persistent memory storage containing the Crash Log.

Intel(R) Crash Log Framework Graphical User Interface

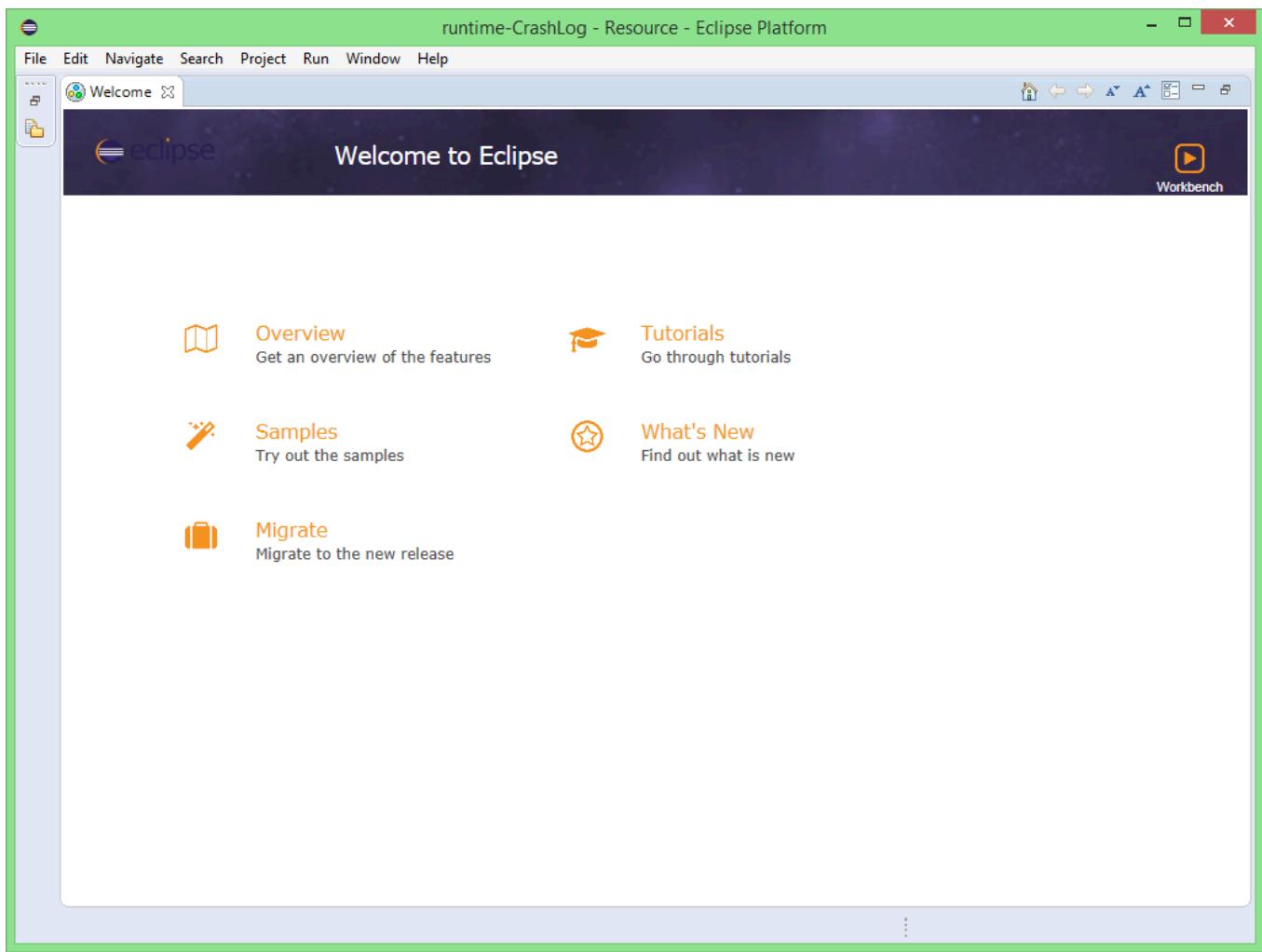
Overview

Intel(R) System Debugger - Crash Log GUI provides a user interface and additional visualization capabilities on top of Crash Log Framework to manually trigger Crash Log, extract, decode, and present Crash Log data.

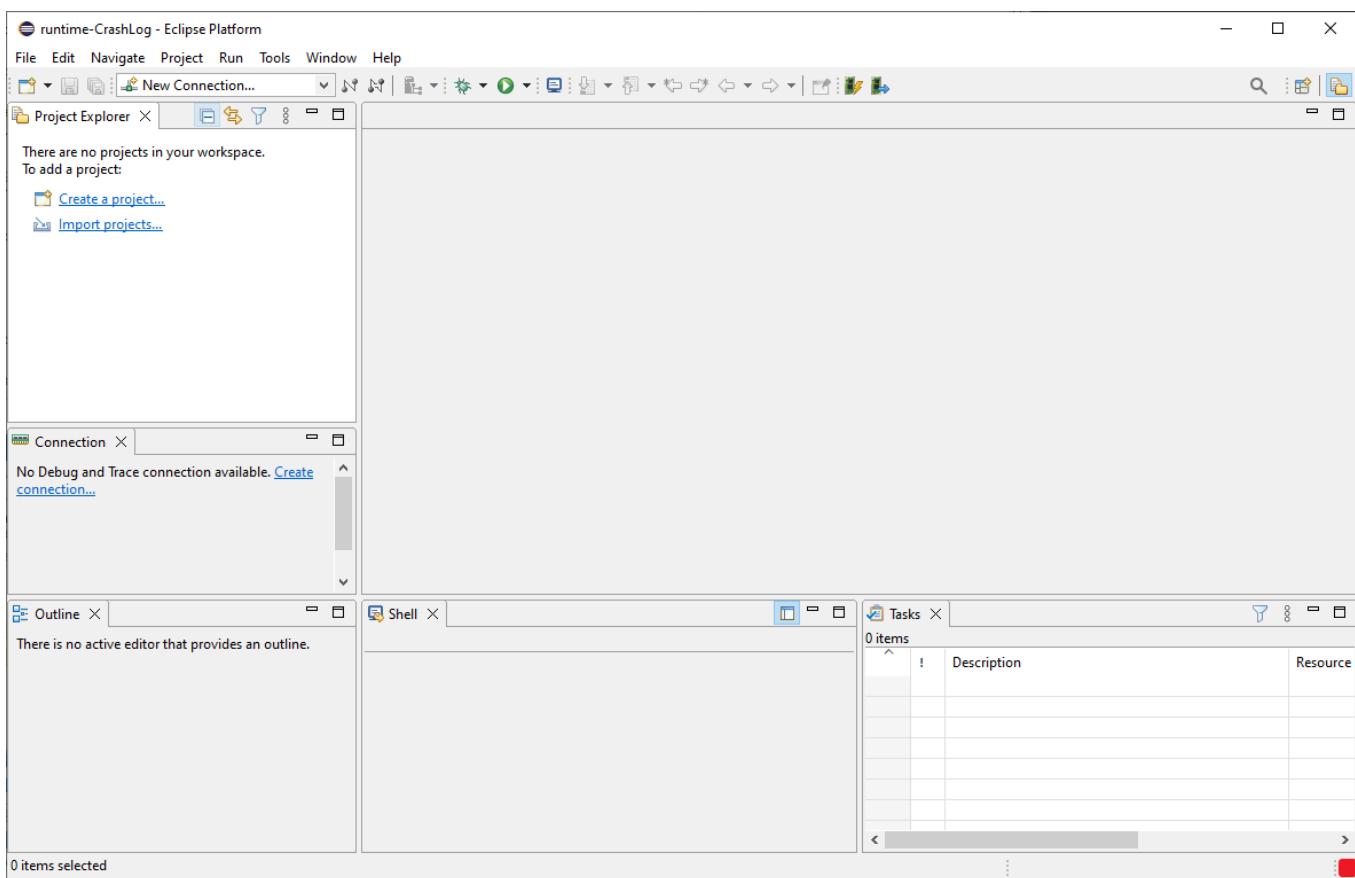
Getting Started

Open Crash Log Perspective

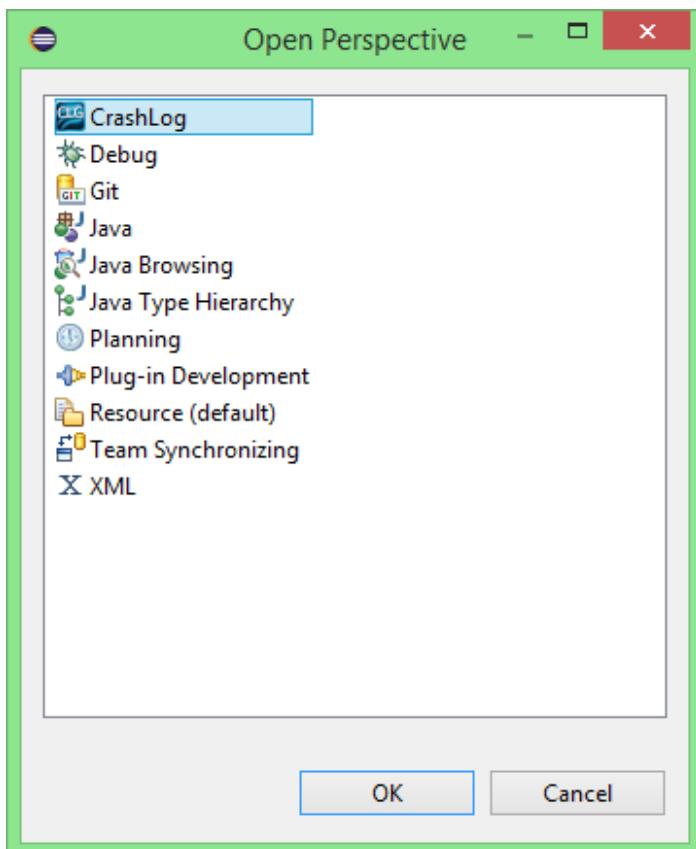
Launch Intel(R) System Debugger - Crash Log.



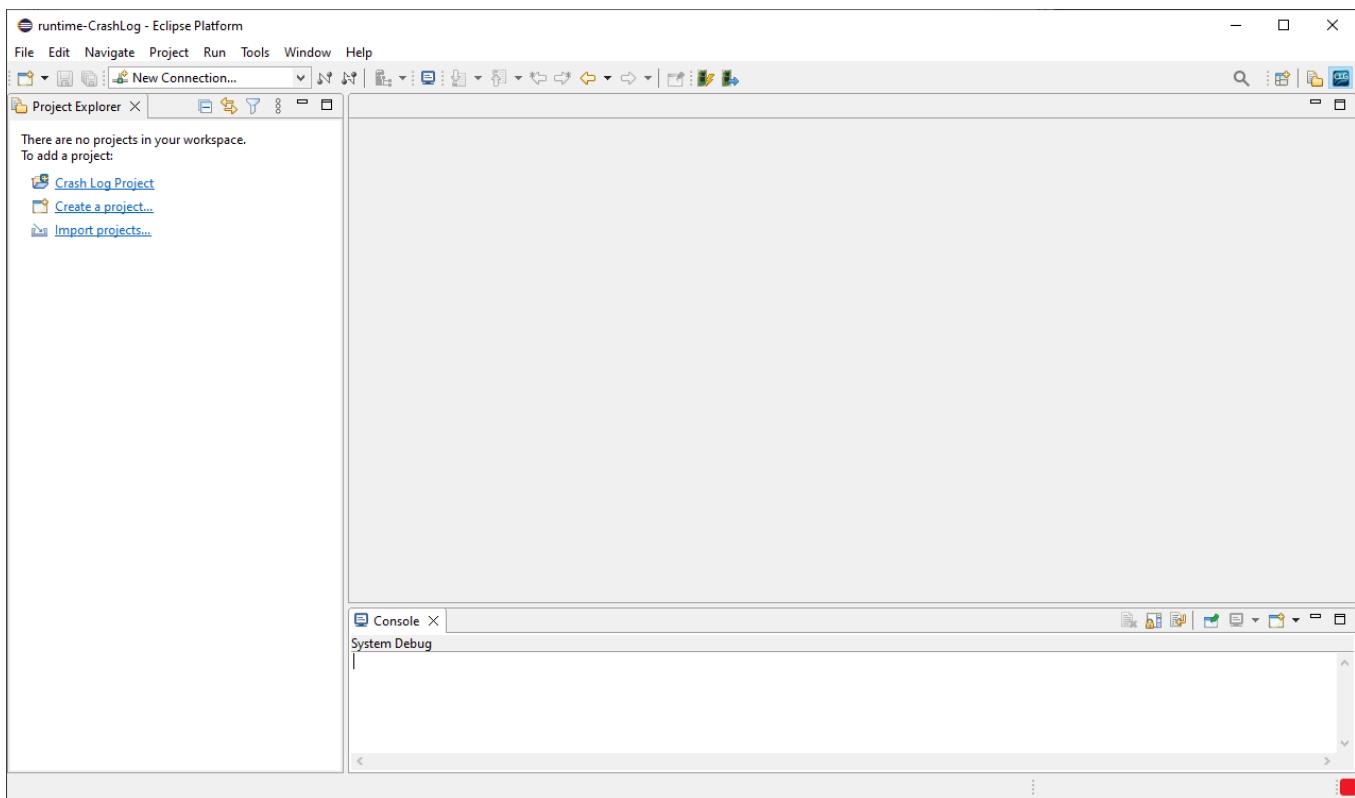
If the *Welcome* page is opened, close it by clicking the close button or the *Workbench* button at the top right corner.



Open the *Open Perspective* dialog via *Window > Perspective > Open Perspective > Other...* or by clicking the *Open Perspective* button at the top right corner.

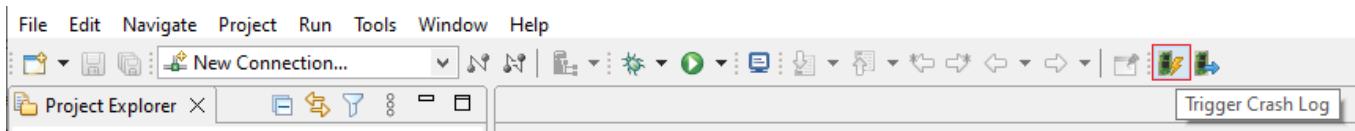


Select *Crash Log* and click the *OK* button.

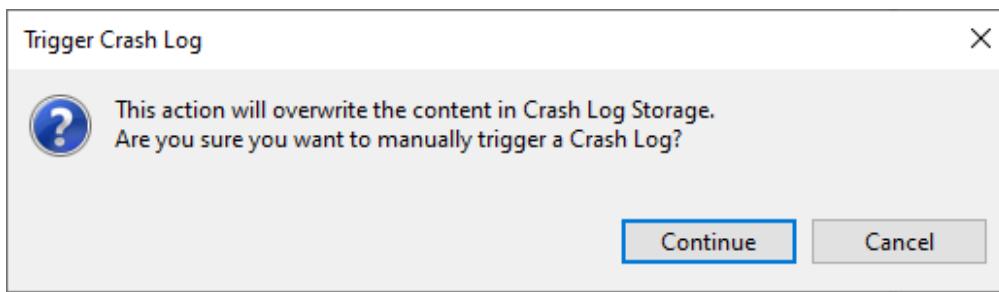


Extract Crash Log File

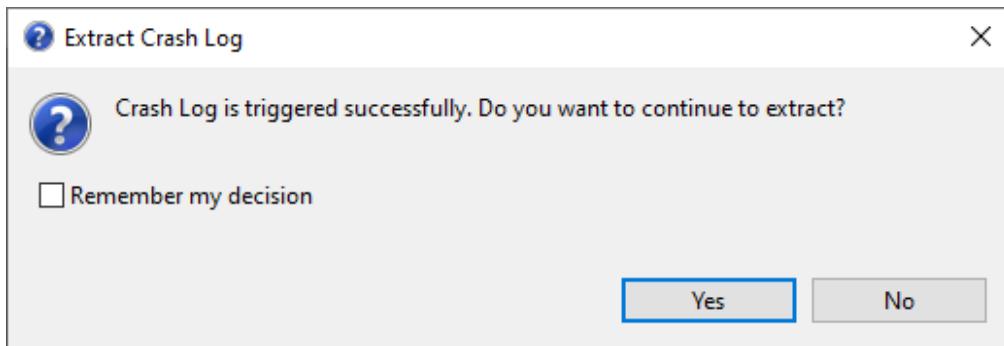
Trigger Crash Log Manually



When an out-of-band trigger is available in the platform under test, you can call it by clicking the *Trigger Crash Log* button in the editor toolbar.



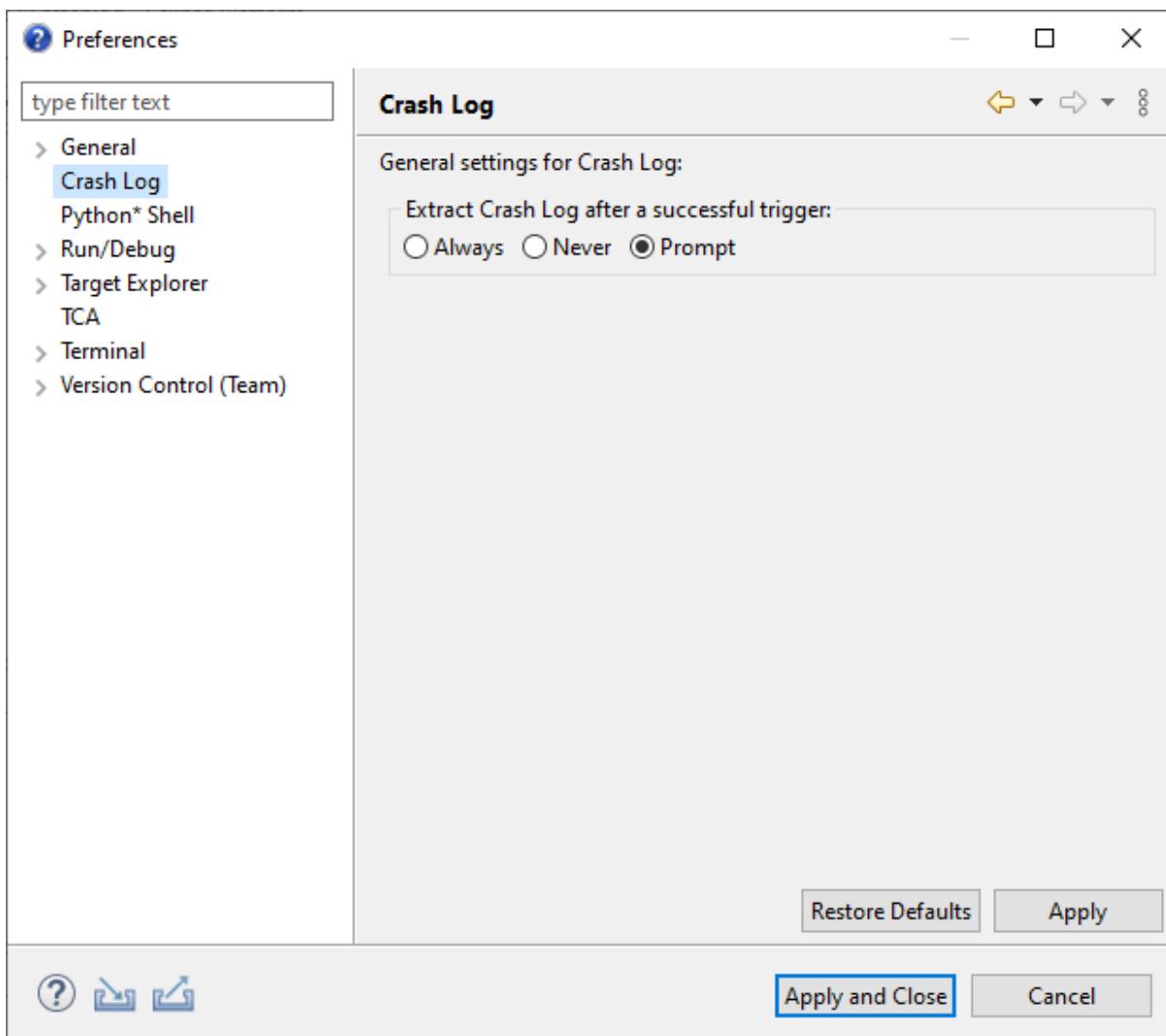
Click *Continue* to enable overwriting the content in Crash Log Storage.



When the Crash Log is triggered, click Yes to extract the Crash Log Raw File. The extracted file is decoded, analyzed, and displayed in the Crash Log Viewer automatically.

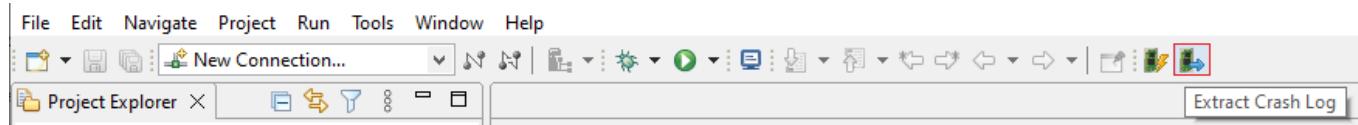
You can change this setting by the following steps below:

1. Select *Window > Preferences* to open the *Preferences* dialog box.
 2. Navigate to *Crash Log* and set your preferred option.
- #. Click *Apply* to save your changes or *Apply and Close* to save changes and close the dialog.



Extract Crash Log

You can extract an existing Crash Log from the Crash Log Storage.



Click *Extract Crash Log* in the editor toolbar. Alternatively, select *Run > Extract Crash Log* from the menu bar.

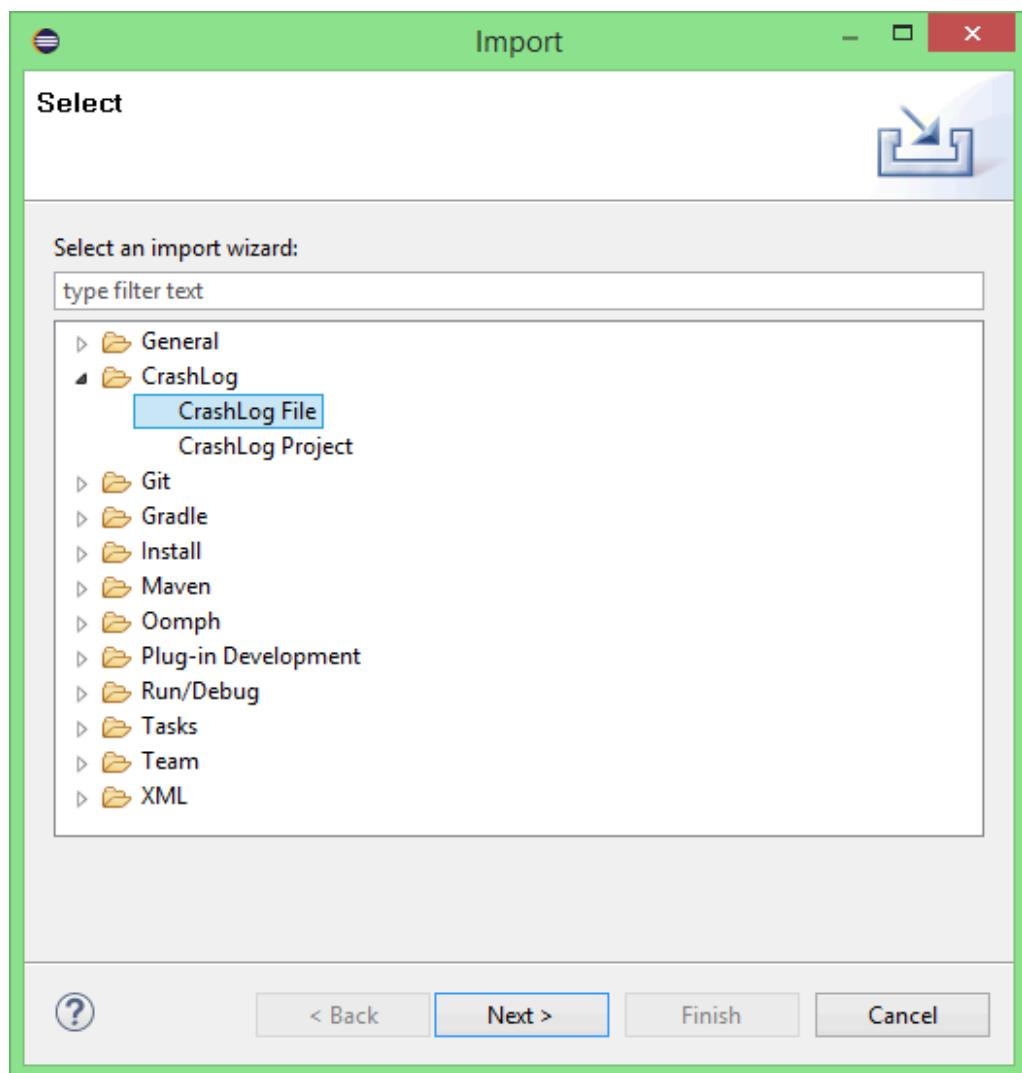


The extracted file is decoded, analyzed, and displayed in the Crash Log Viewer automatically.

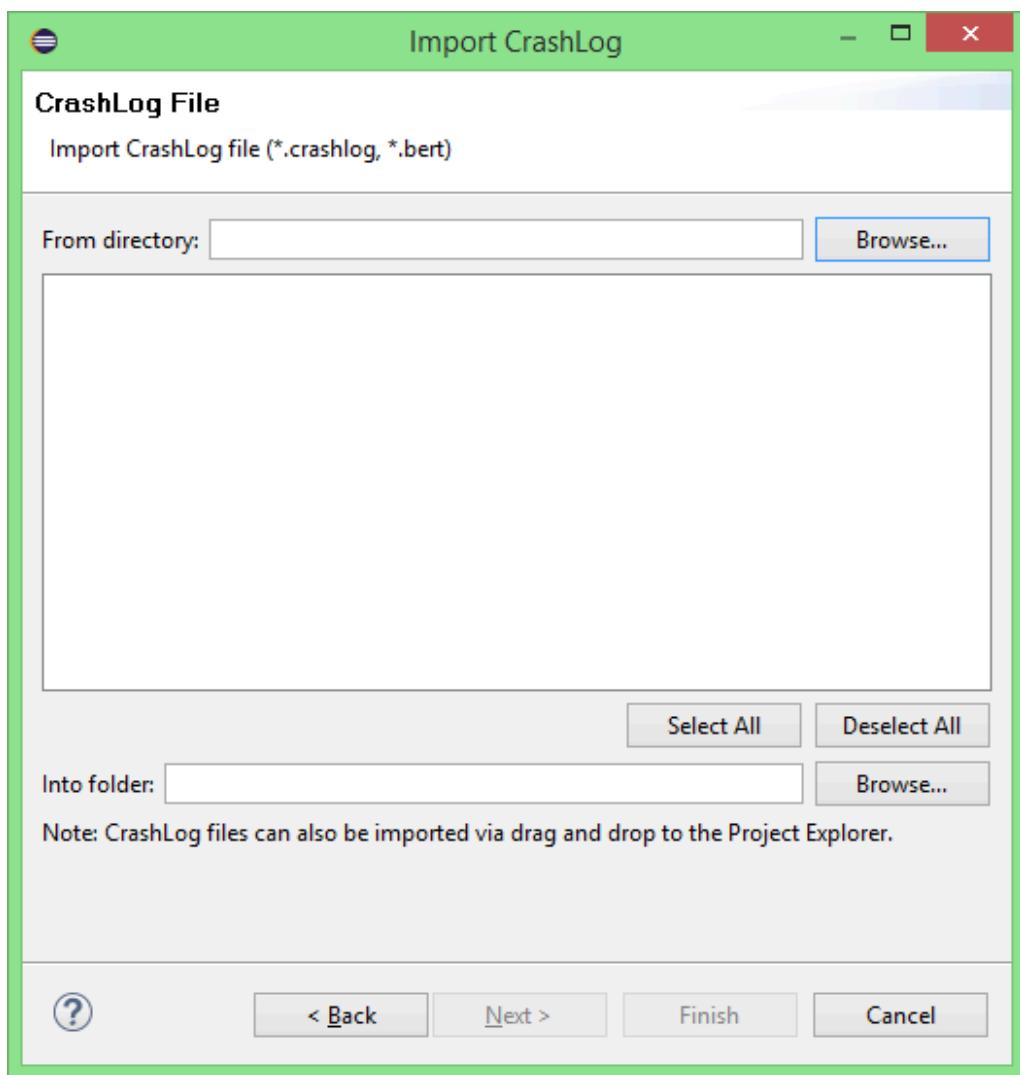
Once a Crash Log Raw File is extracted, it can be decoded and analyzed. The next section provides an example of how to view the Crash Log Raw File in the tool for the first time.

Import Crash Log File

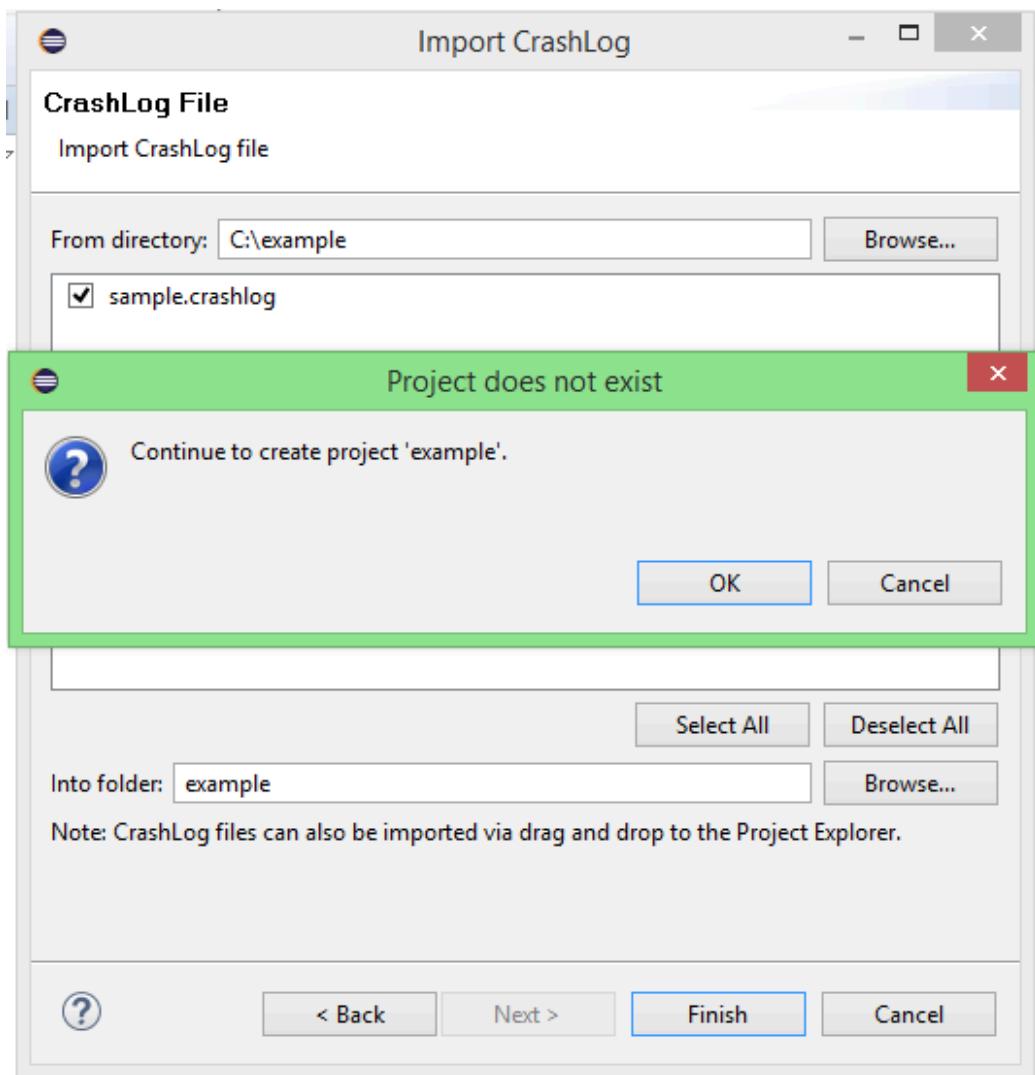
In *Crash Log* perspective, open *Import* dialog via *File > Import...* or right click on *Project Explorer* view and select *Import....*



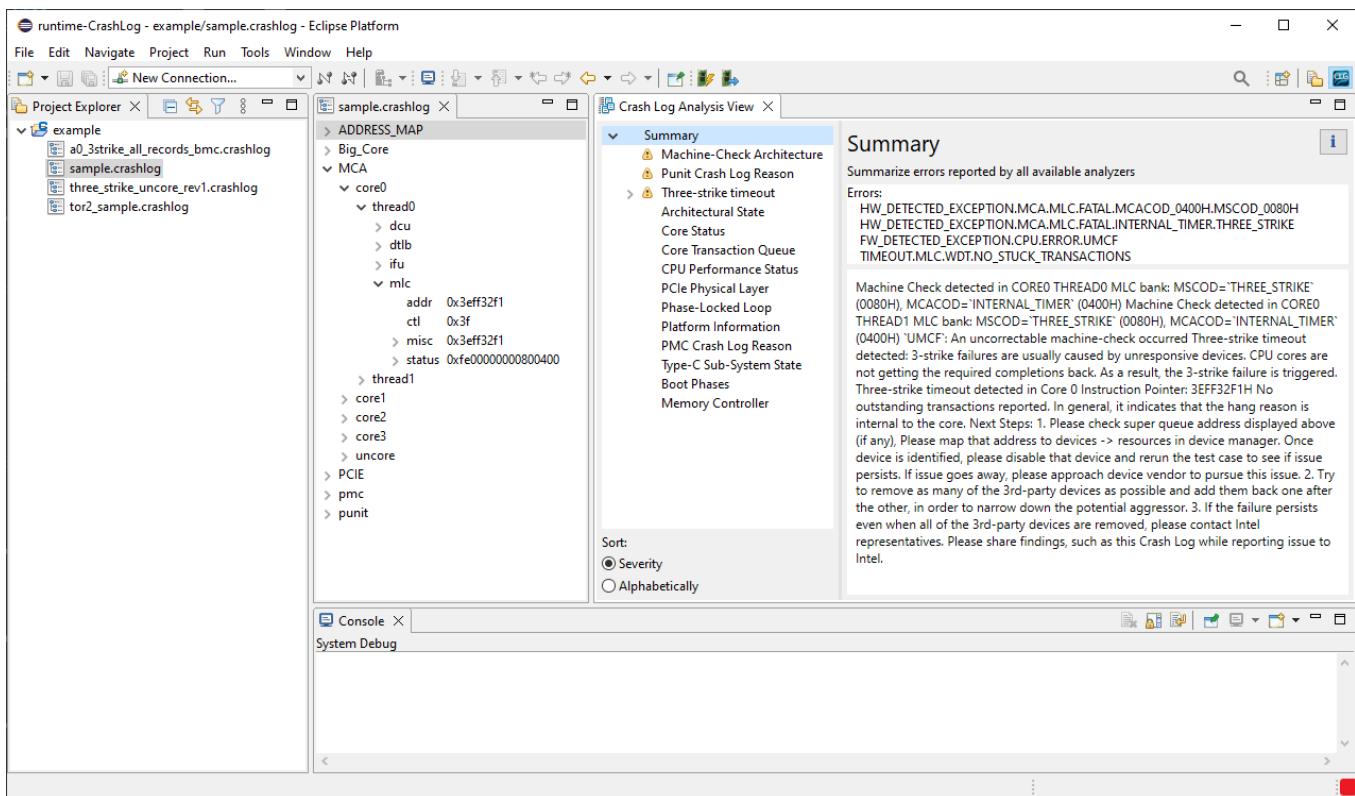
Select *Crash Log > Crash Log File* and click the *Next >* button.



Click the top *Browse...* button and select the directory that contains the Crash Log Raw File. Select files that should be imported. Enter a name into the bottom text field as the name of a new Crash Log project. Click the *Finish* button.



Click the OK button.



If only a single file is imported, it will be opened automatically. If multiple files are imported, double click on the imported file or right click on the file and select *Open With > Crash Log Viewer*.

The Crash Log data is presented in two separated windows:

- Crash Log Viewer

On the left is the *Crash Log Viewer* that is displaying registers and their values. Each Crash Log file is opened in a separate tab.

- Crash Log Analysis View

On the right is the *Crash Log Analysis View* that presents all available analyses. This view is showing analyses corresponding to file opened in the active *Crash Log Viewer*.

The analyzer tree has the *Summary* as the root node, which lists all detected errors. All analyzers are attached as dependency of the *Summary* node, where each of them also have the analyzers they depend on listed as child nodes. Each analyzer that has detected an issue is indicated with an icon with an exclamation mark.

Glossary

Crash Log Extraction

Refers to the reading or transformation of a Crash Log from a Crash Log Storage to a computer file (Crash Log Raw File).

Crash Log Raw File

The Crash Log Raw File is a computer file representation of the crash data.

Crash Log Storage

Persistent memory storage containing the Crash Log.

Crash Log Trigger

Refers to manually overwriting the content in the Crash Log Storage.

Intel(R) Debug Extensions for WinDbg*

This guide contains instructions on using Intel(R) Debug Extensions for WinDbg* to perform the following tasks:

- Basic debugging tasks
- Perform trace execution with Intel(R) Processor Trace
- Debug Advanced Configuration and Power Interface (ACPI) Machine Language (AML) code
- Debug Virtual Machine Monitors.

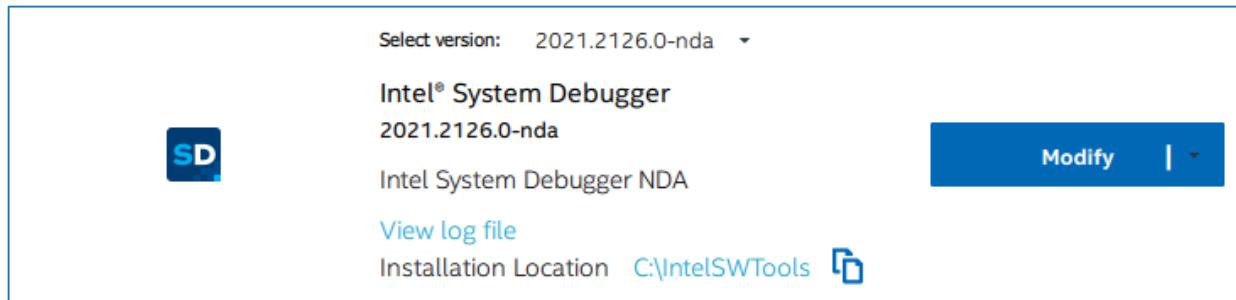
Get Started

To start working with the tool, complete steps described in the child pages of this chapter.

UI Theme by Intel

By default, a special theme designed by Intel is applied to your WinDbg* installation. You can modify your theme selection (uninstall or reinstall it) after the product installation as follows:

1. Launch your installer again.
2. Select the latest version of Intel(R) System Debugger NDA, click **Modify**, and select **Add/Remove components**.



3. Expand the Intel(R) Debug Extensions for WinDbg* component, (un)check the **Intel(R) UI theme for WinDbg*** box, and let the installer modify the product.



Intel(R) Debug Extensions for WinDbg*
2023.2314.0-nda | 1.3 GB



Debug and trace Windows* kernel and device drivers over Intel® Direct Connect Interface.

Optional



Intel(R) UI theme for WinDbg*
2023.2314.0-nda

Custom GUI theme for WinDbg* by Intel

Without this component installed, WinDbg* will use the default theme by Microsoft*.

Launch Intel(R) Debug Extensions for WinDbg*

1. WinDbg* must have access to kernel symbols for the connection to succeed; therefore, you must set the symbol file path beforehand.

Launch WinDbg* without using the batch script and set the symbol file path by selecting **File > Symbol File Path** and adding `srv*C:\Symbols*http://msdl.microsoft.com/download/symbols` to the path or by setting the `_NT_SYMBOL_PATH` global environment variable with a corresponding value. Save the workspace by selecting **File > Save Workspace** and close WinDbg*.

You need to do this only once before the first use.

2. Connect the host system to the target one with the Intel(R) In-Target Probe (Intel(R) ITP) or the Intel(R) Direct Connect Interface (Intel(R) DCI).
3. Power on the probe and the target system.
4. To launch Intel(R) Debug Extensions for WinDbg* with OpenIPC choose any of the following options:

- Click the desktop icon or open the **Start Menu** and search for **Intel(R) Debug Extensions for WinDbg* <year> NDA <version>**.
- Run `windbg_iajtag_console.bat` located at `<install_dir>\system_debugger\<version>` as follows:
 - `windbg_iajtag_console.bat` to launch the default Python*.
 - `windbg_iajtag_console.bat -r` to launch IPython*.
- Launch `isd_shell.bat` located in the root installation directory and run `windbg_dci` to invoke WinDbg*.
- Launch WinDbg* from the Intel(R) System Debugger Eclipse* IDE:
 - Launch the Intel(R) System Debugger.
 - Connect to the target using the Target Connection Assistant.
 - From the main menu, select **Tools > Intel(R) System Debugger NDA > Launch WinDbg* Over Intel(R) Direct Connect Interface (Intel(R) DCI)**.

You can change launch parameters such as WinDbg* path and startup arguments. In Eclipse*, go to **Window > Preferences > WinDbg/* over Intel(R) DCI**.

5. At this point, two Python* objects are available for debugging:

- `itp` - Intel(R) ITP interface
- `itpkd` - wrapper over WinDbg* and kernel debug console

Execute `windbg()` to halt the target and run a WinDbg* session. After that, WinDbg* starts connecting to the target.

If Microsoft Hyper-V is active on the debug target, the `windbg()` command results in one of the following states:

- If the target is halted, the WinDbg* UI is launched into the currently halted context. If Microsoft* Hyper-V is active, WinDbg* will likely start within its context.
- If the target is not halted, the WinDbg* over Intel(R) DCI implementation breaks into NTOSKRNL (NT OS).

6. (Optional) Specify the logical core to search for NT symbol. By default (for Windows* Software Development Kit (SDK) version 18500 or higher), WinDbg* uses Guess startup argument to search for NT symbol. This argument makes `windbg` to search in all available threads (logical cores).

You can change the default behavior as follows:

- To force WinDbg* to use NtBaseAddr startup argument, execute the following command before launching `windbg()`:

```
force_NTBaseAddr()
```

It will search for NT symbol in core 0.

- To select a particular logical core for the search, execute the following command:

```
select_logical_core_NTBaseAddr(<logical_core>)
```

It forces WinDbg* to use NtBaseAddr startup argument only search in the specified core. This is useful if you want to search in one particular core only. For example, when Microsoft* Hyper-V is running, and one of the cores is running NTOSKRNL.

To unselect this startup argument and use the default parameters, execute the following command:

```
select_logical_core_NTBaseAddr(-1)
```

Afterwards, you can halt the target any time by running the `halt()` command.

Target Connection

Use Target Connection Assistant (TCA):

1. Activate TCA with the following command:

```
!tca_select_profile
```

2. Choose the connection selection mode:

- [a]uto detect (recommended) - TCA attempts to automatically detect your target and the connection method used.

If TCA returns a list of possible targets, select the right one by entering its number from the list.

- [c]reate manual profile - use if your target cannot be automatically detected.

Choose the target from a list by specifying its number and choose the connection method the same way.

Note

Connecting to the target can take several minutes. Do not enter any commands until the connection is fully established and the connection confirmation message is displayed ([Target initialization succeeded](#)).

If the connection fails, you see an exception in the console. Check the network connection if [KDVersionBlock](#) is not found. Run the target for a while if the kernel is not found.

Using Intel(R) Debug Extensions for WinDbg* with WinDbg* Preview

System Requirements

Microsoft WinDbg* Preview. For download and installation instructions, see the [Windows* Debugging Tools documentation](#).

To start using Intel(R) Debug Extensions for WinDbg* with WinDbg* Preview, follow the steps below:

1. Connect to the target.
2. Launch [isd_shell.bat](#) located in the installation root directory.
3. Execute the [windbg_preview](#) command to start WinDbg* Preview.

Using Intel(R) Debug Extensions for WinDbg* over Intel(R) Direct Connect Interface

This chapter describes basic debugging operations that you can perform with Intel(R) Debug Extensions for WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI). Before you proceed, make sure you have checked [startup instructions](#).

Setting Low Voltage Margining Mode

Once you have connected to the target via Target Connection Assistant (see the [Get Started page](#) for details), you can set Low Voltage Margining (LVM) modes using the following commands:

[!tcasetlvm](#)

Select the LVM mode from the list.

`!showlvm`

Display the currently set mode.

Target Run Control

The list below introduces the basic set of run control commands:

`halt()`

Stop the execution of all general purpose cores.

`status()`

Get information about all general purpose cores executing when the target is halted.

`forensic.image_scans()`

Get information about the Microsoft* system application running in the halted debug context.

`go()`

Resume general purpose cores after stopping at a desired debug context.

To use these commands, ensure that the target is connected and launch the Python* console for WinDbg* any of the following ways:

- Click the desktop icon or open the **Start Menu** and search for **Intel(R) Debug Extensions for WinDbg* <year> NDA <version>**.
- Run `windbg_iajtag_console.bat` located at `<install_dir>\system_debugger\<version>`.
- Launch `isd_shell.bat` located in the root installation directory and run `windbg_dci` to invoke WinDbg*.

Setting Breakpoints

WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) offers probe mode breakpoint support for Virtual Machine Monitors (VMM). Use the `breakin()` function to stop WinDbg* after a specified event. After the event is reported, you can start WinDbg* over Intel DCI and continue the session or proceed with using the console mode.

To set a breakpoint in python console

Break into the host kernel. From this point, WinDbg* is able to load symbols and KdBaseAddress found.

```
breakin(NTOSKRNL)
```

Break into the probe mode after SMMENTRY event.

```
breakin(SMMENTRY)
```

Break into the probe mode after SMMEXIT event.

```
breakin(SMMEXIT)
```

Break into the probe mode after VMENTRY event

```
breakin(VMENTRY)
```

Break into the probe mode after VMEXIT event

```
breakin(VMEXIT)
```

Break into the host kernel preserving the state of a selected core (not resuming a core) to avoid loosing key information. <core_number> is an ordinal number of a core of interest (enumeration starts from 0).

```
breakin_ntoskrnl_halted_cores(<core_number>)
```

To set a breakpoint in WinDbg UI

From this point, WinDbg* is able to set breakpoint into Windows OS Initialization Phase0.

```
!initbreak
```

Break into the probe mode after SMMENTRY event.

```
!vmentrybreak
```

Break into the probe mode after SMMEXIT event.

```
!vmexitbreak
```

Break into the probe mode after VMENTRY event

```
!smmentrybreak
```

Break into the probe mode after VMEXIT event

```
!smmexitbreak
```

Commands to remove the breakpoints: Remove breakponit that to break into the host kernel.

```
!delinitbreak
```

Remove breakponit that to break into the probe mode after SMMENTRY event.

```
!delvmentrybreak
```

Remove breakponit that to break into the probe mode after SMMEXIT event.

```
!delvmexitbreak
```

Remove breakponit that to break into the probe mode after VMENTRY event

```
!delsmmentrybreak
```

Remove breakponit that to break into the probe mode after VMEXIT event

```
!delsmmexitbreak
```

Performing Full Memory Dump

Creating a memory dump can be useful in case Windows* OS installed on the debug target becomes unresponsive. You can also use a full memory dump to examine threads/processes of the target machine without the speed constraints of a JTAG connection.

Requirements

To perform a full memory dump using Intel(R) Debug Extensions for WinDbg*, you need:

- Intel(R) Direct Connect Interface (Intel(R) DCI) USB 2 or USB 3 cable
- Target system CPU formerly codenamed Cannon Lake or newer
- Symbols from the Windows* target

To create a full memory dump:

1. Ensure that the target is connected.

2. Launch Python* console for WinDbg* any of the following ways:

- Click the desktop icon or open the **Start Menu** and search for **Intel(R) Debug Extensions for WinDbg* <year> NDA <version>**.
- Run `windbg_iajtag_console.bat` located at `<install_dir>\system_debugger\<version>`.
- Launch `isd_shell.bat` located in the root installation directory and run `windbg_dci` to invoke WinDbg*.

3. In the launched console, run the following command:

```
forensic.generate_full_kernel_dump(<dump_filename>, timeout=<timeout>)
```

where `<dump_filename>` is the path to the dump file, and `<timeout>` is the amount of time, in milliseconds, that Intel Debug Extensions for WinDbg* will wait for response from the target system before failing. For example, you can use the following values:

```
forensic.generate_full_kernel_dump(r"c:\temp\dump.dmp", timeout=60000)
```

When Intel Debug Extensions for WinDbg* complete the process of capturing the memory dump, the command prompt becomes active again.

Verifying the Dump File

To make sure the memory dump file is not corrupted, launch the WinDbg* software and select **File > Open Crash Dump**. In the Open Crash Dump dialog box, select the dump file that you created and wait for it to load.

Collecting Stop Information

Intel(R) Debug Extensions for WinDbg* provide a way to collect general information about a Stop error (also known as the bugcheck error and the Blue Screen of Death). This capability can be useful for cases when you cannot obtain a functional full memory dump or you do not require a high level of detail.

You can also execute BSOD AR Generator from WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI). It can help you triage the BSOD and provides next steps and automated analysis from the current Stop error.

Important

To execute the commands described below, symbols from Windows* target must be loaded.

To collect information about a Stop error:

1. Ensure that the target is connected.
2. Launch Python* console for WinDbg* any of the following ways:
 - Click the desktop icon or open the **Start Menu** and search for **Intel(R) Debug Extensions for WinDbg* <year> NDA <version>**.
 - Run `windbg_iajtag_console.bat` located at `<install_dir>\system_debugger\<version>`.
 - Launch `isd_shell.bat` located in the root installation directory and run `windbg_dci` to invoke WinDbg*.
3. In the launched console, run the following command to collect general information:

```
forensic.get_bsod_info(<filename>)
```

where `<filename>` is the path to the output text file.

4. (Optional) Execute the BSOD AR Generator to analyze system state:

```
forensic.bsod_ar_generator(<filename>)
```

where `<filename>` is the path to the output text file.

Depending on the type of the Stop error, the output text file may contain a combination of the following information:

- List of loaded modules with offsets and symbols information
- Call stacks of all threads
- Output of the `!analyze -v -f` WinDbg* command.

Intel(R) Debug Extensions for WinDbg* for Intel(R) Processor Trace (Intel(R) PT)

The Intel(R) Debug Extensions for WinDbg* for Intel(R) Processor Trace is designed to help WinDbg* users by extending their debugging tool set with execution tracing. The extension allows for easy setup of Intel(R) Processor Trace (Intel(R) PT) by abstracting hardware configuration and then reconstructing and displaying execution flow from the collected trace data. It integrates with other WinDbg* features like symbolization and high-level source display.

Intel(R) Processor Trace is a new technology for low-overhead execution tracing. It facilitates debugging a program by exposing an accurate and detailed trace of the program's activity, and its triggering and filtering capabilities help identifying and isolating the relevant program executions.

Intel(R) PT records information about software execution on each hardware thread using dedicated hardware facilities. After execution completes, a software can process the recorded trace data and reconstruct the exact program flow.

Intel(R) PT use cases include:

- Control flow tracing

This includes recorded trace data program flow information (for example, branch targets, branch taken/not taken indications) and program-induced mode related information (for example, Intel(R) Transactional Synchronization Extensions (Intel(R) TSX) state transitions). Debuggers can use it to reconstruct the code flow that led to a certain location. Whether this is a crash site, a breakpoint, a watchpoint, or simply the instruction following a function call we just stepped over.

- Debugging stack corruptions

When the call stack has been corrupted, normal frame unwinding usually fails or may not produce reliable results. Intel(R) PT can be used to reconstruct the stack backtrace based on actual CALL and RET instructions.

Requirements and Constraints

Since the extension is built around the Intel(R) Processor Trace technology, it requires the technology to be available on the processors the target uses. Supported are Intel(R) Processor Trace implementations starting with the 6th generation Intel(R) Core™ processor family.

This extension supports kernel-mode debug for live targets only.

The extension also works with the Microsoft Kernel Debugger*.

! Note

Some of the extension features require a GUI (for example, source view links) and are not available for the Microsoft Kernel Debugger*.

Loading the Intel(R) Debug Extensions for WinDbg* for Intel(R) Processor Trace (Intel(R) PT)

When installed correctly, WinDbg* detects the extension automatically. If WinDbg* does not detect and load the extension automatically or if the extension was installed manually, perform one of the following actions:

- Place `ptext32.dll` and `sysdbg64.dll` in a directory where WinDbg* searches for extensions by default.

These are usually all directories in the `PATH` environment variable and some others. To display all searched directories, use the `.expath` command in WinDbg*.

- Add the path to `ptext32.dll` and `sysdbg64.dll` to the `_NT_DEBUGGER_EXTENSION_PATH` environment variable. Create that variable, if it does not exist.
- Use the `.expath+ <directory>` command in WinDbg* to add the directory containing `ptext32.dll` and `sysdbg64.dll` to the search path.

! Note

This setting only applies to the current debug session and is not saved when WinDbg* is closed.

Usually, WinDbg* loads detected extensions automatically. If this fails, you can load the extension manually. Use the `.load ptext32` or `.load sysdbg64` command in WinDbg*, depending on whether you are using a 32-bit or 64-bit version of WinDbg*.

For more information on loading debug extensions in WinDbg*, see the WinDbg* help system.

Commands

By default, all the commands provided by this extension – except for `!decode`, `!help`, and `!version` – apply to all the processors on the target machine. For example, the `!enable` command enables tracing on all the processors, not the currently selected one only. Or, as another example, the commands to install a new IP filter do so on all the processors. If this behavior is not desired, it can be changed by specifying the `/here` flag. For example, `!enable /here` enables tracing on the currently selected processor only.

As mentioned, the major exception to this rule is the `!decode` command. This always applies to the currently selected processor only. To see trace decodes from all the processors, users must manually switch processors using the `~s` command. For example, for a target machine with two processors:

```
1: kd> ~0s  
  
0: kd> !decode  
  
...  
  
0: kd> ~1s  
  
1: kd> !decode  
  
...  
  
1: kd>
```

Displaying the Current Configuration

To display the currently used Intel(R) Processor Trace (Intel(R) PT) configuration, use the `!showconfig` command. This prints a table of all the relevant model-specific registers (MSRs) that make up the Intel(R) PT configuration. After that, the extension prints a human-readable interpretation of that raw data, like, for example, `Tracing is enabled`.

Following is an example output on a target machine with two processors:

```
0: kd> !showconfig
```

```
Intel(R) Processor Trace configuration on processor 0
```

MSR	address	value
IA32_RTIT_CTL	570	0000000100002104
IA32_RTIT_STATUS	571	0000000000000002
IA32_RTIT_OUTPUT_BASE	560	0000000000001000
IA32_RTIT_OUTPUT_MASK_PTRS	561	000000000000007f
IA32_RTIT_CR3_MATCH	572	0000000000000000
IA32_RTIT_ADDR0_A	580	fffff80002851c10
IA32_RTIT_ADDR0_B	581	fffff80002851e81
IA32_RTIT_ADDR1_A	582	0000000000000000
IA32_RTIT_ADDR1_B	583	0000000000000000

```
Tracing is disabled.
```

```
CR3 filtering is disabled.
```

```
IP filter configuration:
```

```
0: fffff80002851c10 - fffff80002851e81 enable
```

```
1: 0000000000000000 - 0000000000000000 unused
```

```
Table of Physical Addresses (ToPA) output method used.
```

```
Table of Physical Addresses (ToPA) output configuration details:
```

```
.-- TABLE 000000000001000
```

```
`-- REGION base 000000000006000, size 8K, flags ---- STOP
```

```
Intel(R) Processor Trace configuration on processor 1
```

MSR	address	value
IA32_RTIT_CTL	570	0000000100002104
IA32_RTIT_STATUS	571	0000000000000002
IA32_RTIT_OUTPUT_BASE	560	0000000000002000
IA32_RTIT_OUTPUT_MASK_PTRS	561	000000000000007f
IA32_RTIT_CR3_MATCH	572	0000000000000000

IA32_RTIT_ADDR0_A	580	fffff80002851c10
IA32_RTIT_ADDR0_B	581	fffff80002851e81
IA32_RTIT_ADDR1_A	582	0000000000000000
IA32_RTIT_ADDR1_B	583	0000000000000000

Tracing is disabled.

CR3 filtering is disabled.

IP filter configuration:

0: fffff80002851c10 - fffff80002851e81 enable

1: 0000000000000000 - 0000000000000000 unused

Table of Physical Addresses (ToPA) output method used.

Table of Physical Addresses (ToPA) output configuration details:

.-- TABLE 000000000002000

`-- REGION base 000000000008000, size 8K, flags ---- STOP

Enabling and Disabling Tracing

To start recording trace data (enable tracing), use the `!enable` command. To stop recording trace data (disable tracing), use the `!disable` command.

Note

As long as tracing is enabled, the trace output buffer gets filled. When the buffer is full, older trace data is overwritten by newer trace data. To make sure useful trace data is not overwritten, it is recommended to disable tracing as early as possible.

To show the current enable state, use the `!showenable` command.

The following code snippet is an example how to use the `!enable` and `!disable` commands together with `!showenable`:

```
2: kd> !showenable

<0> Tracing is disabled.

<1> Tracing is disabled.

<2> Tracing is disabled.

<3> Tracing is disabled.

2: kd> !enable

2: kd> !showenable

<0> Tracing is enabled.

<1> Tracing is enabled.

<2> Tracing is enabled.

<3> Tracing is enabled.

2: kd> !disable /here

2: kd> !showenable

<0> Tracing is enabled.

<1> Tracing is enabled.

<2> Tracing is disabled.

<3> Tracing is enabled.

2: kd> !disable

2: kd> !showenable

<0> Tracing is disabled.

<1> Tracing is disabled.

<2> Tracing is disabled.

<3> Tracing is disabled.
```

Output Configuration

The extension always recognizes any existing hardware configuration it finds on the target. There are several alternatives for an output configuration:

- BIOS / UEFI firmware

With firmware that is aware of Intel(R) Processor Trace (Intel(R) PT), you can set up an Intel(R) PT specific memory allocation. In this case, the firmware allocates a dedicated memory area and reserves it in a memory map for further use. Operating systems recognize this reserved memory

range and do not use it. When the firmware reserves a memory region for Intel(R) PT, it also configures the Intel(R) PT output MSRs accordingly and indicates that Intel(R) PT output configuration is ready to be used. The extension recognizes this setup.

No further configuration (from the user's side) is required.

- Windows* /burnmemory or /maxmem boot parameter, or the truncatememory or removememory boot configuration data (BCD) option

With these boot parameters or BCD options, it is possible to reserve memory above a certain specified address, so that it is not used by the kernel. For more information on how to use this configuration, refer to the MSDN article [Boot Parameters to Manipulate Memory](#). When this memory reservation technique is used, the Intel(R) PT hardware needs to be configured accordingly through the !setoutputconfig command (see below).

- Windows* (BCD) with a `{badmemory}` object and its `badmemorylist` value

The BCD can hold a special `{badmemory}` object that describes a defective RAM. The object's `badmemorylist` value contains a list of page frame numbers (PFNs) not used when Windows* allocates physical memory. Because the kernel treats all entries on that list the same, this mechanism can be used to reserve arbitrary page frames that are not defective. The kernel does not use those frames. When this memory reservation technique is used, the Intel(R) PT hardware needs to be configured accordingly through the `!setoutputconfig` command (see below).

To use this feature, run the command-line interface (CLI) with the administrator rights and issue the following commands:

```
bcdedit /set badmemoryaccess no  
bcdedit /set badmemorylist <list of PFNs, space-delimited>
```

PFN is the number of the page frame, not its (base) address. For example, the PFNs for addresses 0x1000 and 0x1234abc are 0x1 and 0x1234.

The `!setoutputconfig` command creates a simple output configuration fitting into a specified memory range and programs the Intel(R) PT hardware accordingly. You can specify the memory range by giving a start address and a size. It is assumed that this memory can be exclusively used by the Intel(R) PT hardware, and no other entity (for example, the operating system or any other software) reserves and uses it.

```
!setoutputconfig [/here] /start <expr> /size <expr>  
  
/start <expr> - Start address of available buffer  
  
/size <expr> - Size of available buffer  
  
/here - Create and set output configuration on the currently selected processor only
```

When the `/here` option is not set (default), the extension creates an overall output configuration in which all the processors have the same buffer size and the provided memory range is used to its maximum. Also, the extension correctly aligns the output buffers. When the `/here` option is set, the extension searches for the largest supported buffer size under correct alignment that fits into the specified memory range and programs the current processor's Intel(R) PT hardware to use it.

IP Filter Configuration

When tracing the target software execution, the Intel(R) Processor Trace (Intel(R) PT) hardware records every event. To focus on information of your interest, you can use IP filtering. Using this feature, you can specify address ranges where the software execution is traced.

For example, to trace the execution of a specific function only, you can program the hardware with the address range containing the function. The execution trace is only recorded for instructions executed within that range. For instruction executions outside of that range, execution trace is not recorded.

IP filters facilitate working with the Intel(R) Debug Extensions for WinDbg* for Intel(R) PT, because the debug agent executes all the actions (for example, run-control like stepping, reading registers and memory) on behalf of WinDbg* on the target and communicates regularly with WinDbg*. Usually, tracing all those actions of the debug agent is not desired. Using IP filters, the Intel(R) PT hardware only records the code execution of interest.

The most basic form of installing an IP filter is to use the `!setipfilter` command to specify an address range where code execution is traced. For convenience, the extension also provides commands to set a filter based on a function name with `!setipfilterfunction` or a module name with `!setipfiltermodule`, where the extension automatically looks up the address range of that function or module.

Displaying the IP Filter Configuration

```
showipfilters [/all] [/here]

/all - Also show filters that are currently unused
/here - Show IP filter configuration for the currently selected processor only
```

Setting / Adding a New IP Filter

Configure Intel(R) Processor Trace IP filtering using a memory range

```
!setipfilter [/stop] [/here] [/id <expr>] <start> <end>

<start> - Start address of filter region

<end> - End address of filter region

/stop - Configure for trace-stop

/id <expr> - ID of filter to overwrite

/here - Set IP filter configuration on the currently selected processor only
```

Configure Intel(R) Processor Trace IP filtering using a function symbol

```
!setipfilterfunction [/stop] [/here] [/id <expr>] <name>

<name> - Name of function to filter on

/stop - Configure for trace-stop

/id <expr> - ID of filter to overwrite

/here - Set IP filter configuration on the currently selected processor only
```

Configure (R) Processor Trace IP filtering using a module name

```
!setipfiltermodule [/tracestop] [/here] [/id <expr>] <name>

<name> - Name of module to filter on

/stop - Configure for trace-stop

/id <expr> - ID of filter to overwrite

/here - Set IP filter configuration on the currently selected processor only
```

Deleting an IP Filter

```
Delete Intel(R) Processor Trace IP filter region based on an address range
```

```
!delipfilter [/here] <start> <end>
```

<start> - Start address of filter region

<end> - End address of filter region

/here - Delete IP filter on the currently selected processor only

```
Delete Intel(R) Processor Trace IP filter region based on a function symbol
```

```
!delipfilterfunction [/here] <name>
```

<name> - Name of function

/here - Delete IP filter on the currently selected processor only

```
Delete Intel(R) Processor Trace IP filter region based on a module name
```

```
!delipfiltermodule [/here] <name>
```

<name> - Name of module

/here - Delete IP filter on the currently selected processor only

```
Delete Intel(R) Processor Trace IP filter region based on its ID
```

```
!delipfilterid [/here] <expr>
```

<expr> - Number/ID of filter region to delete

/here - Delete IP filter on the currently selected processor only

```
Delete all Intel(R) Processor Trace IP filter regions
```

```
!delipfilterall [/here]
```

/here - Delete IP filter on the currently selected processor only

CR3 Filter Configuration

The Intel(R) Processor Trace (Intel(R) PT) hardware also provides CR3 filtering, which is based on address spaces. This means that the execution trace is only recorded if the CR3 register value matches the CR3 filter value. Since CR3 values can be considered as process identifiers, you use this feature to only record code executed in the context of a certain process.

Configure Intel(R) PT CR3 filtering as follows:

```
!setcr3filter [/here] <expr>
```

where:

`<expr>`

CR3 filtering value

`/here`

Option to modify CR3 filter configuration on the currently selected processor only

Disable Intel(R) PT CR3 filtering as follows:

```
!delcr3filter [/here]
```

where:

`/here`

Option to disable CR3 filtering on the currently selected processor only

Timestamps

Enabling Timestamps in Intel(R) Processor Trace (Intel(R) PT) Output

You can configure Intel(R) Processor Trace (Intel(R) PT) to include timestamps into its output by invoking the command `!enabletimestamps <options>`.

```

!enabletimestamps [/tsc] [/mtc] [/cyc] [/here] [/mtcfreq <expr>] [/cycthres <expr>]

/tsc - Enable Time Stamp Counter (TSC)

/mtc - Enable Mini Time Counter (MTC)

/mtcfreq <expr> - MTCFrequency value

0 - max frequency

possible options shown if parameter is not provided.

(For example b01001 allow [0 and 3])

(space-delimited)

/cyc - Enable Cycle-Accurate Mode

/cycthres <expr> - CycThreshold value 0 - no threshold

CycThresh>0, CYC packets are generated only after a

minimum number of cycles have passed since the last CYC packet

possible options shown if parameter is not provided.

(For example b01111 allow [0,1,2,3])

(space-delimited)

/here - Enable on the currently selected processor only

Enable timestamps (only JTAG connection)

```

Types of Timestamp Packets in Intel(R) Processor Trace (Intel(R) PT)

There are three types of Intel(R) Processor Trace (Intel(R) PT) timestamp packets that Intel(R) CPUs can generate:

- Time Stamp Counter (TSC) packets

TSC packets contain the approximate wall-clock time of the event that generated the packet. TSC packets are generated by events such as packet generation enable and sleep state wake. TSC packets do not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after the TSC time. There is not a precise IP to which to bind the TSC packet. The value of a TSC packet is the current TSC value as returned by the `RDTSC` instruction.

You can enable TSC packets by passing the option `/tsc` to the `!enabletimestamps` command.

- Mini Time Counter (MTC) packets

MTC packets provide a periodic indication of the wall-clock time. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. As their payload, MTC packets contain the Common Timestamp Copy (CTC). The CTC is an 8-bit range taken from the value of the core crystal clock, starting from the N -th bit. Intel PT sends an MTC packet every time the value of CTC changes; therefore, the value of N controls the frequency of MTC packets.

You can enable MTC packets by passing the option `/mtc` to the `!enabletimestamps` command.

Additionally, you can pass the option `/mtcfreq N` to specify the frequency of the MTC packets. The frequency value N means that an MTC packet is sent every time bit N of the core crystal clock changes. The value 0 means that Intel PT sends an MTC packet every tick of the core crystal clock.

- Cycle Count (CYC) packets

CYC packets provide the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets. A CYC packet precedes other packets generated in the same cycle and provides the precise cycle time of the packets that follow.

You can enable CYC packets by passing the option `/cyc` to the `!enabletimestamps` command.

Additionally, you can pass the option `/cycthres T` to specify the threshold for CYC packet generation. The threshold value $T = 0$ means that no threshold is in use, and a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. A threshold value $T > 0$ means that CYC packets are generated only after at least T cycles have passed since the last CYC packet.

For more information about Intel PT packets, see the [Intel\(R\) 64 and IA-32 Architectures Software Developer's Manual, volume 3C](#)

You can enable and configure Intel PT packets independently of each other, but you have to enable at least one type of packets to enable timestamps.

Choosing Your Timestamp Packet Configuration

Timestamps consume the Intel PT buffer even when no instruction trace is recorded. You have to choose your timestamp packet configuration taking into account the buffer size and the active filters.

Note the following points when choosing your configuration:

- Enabling only Time Stamp Counter (TSC) packets means that you do not have a precise indication of instruction trace timing.

- Enabling Mini Time Counter (MTC) and Cycle Count (CYC) packets, but not TSC packets, gives you timing data relative to 0. Enabling TSC packets gives you the ability to analyze execution with other timing sources.
- Enabling MTC and CYC packets is key to getting precise timing information. Use `/mtcfreq 0` and `/cycthresh 0` to get the maximum level of precision.
- The decoder uses the combination of MTC and CYC packets to track instruction execution time. Increase the MTC frequency and decrease the CYC threshold to find the best combination to fit your needs. Consider the following:
 - MTC packets are periodic and based on the core crystal clock. Using the maximum frequency value `/mtcfreq 0` allows the decoder to identify the timing of small sequences of instructions. Reducing the frequency reduces accuracy.
 - CYC packets rely on the processor core clock, which can experience delays depending on the P-State and thermal conditions. Increasing the threshold for CYC packets reduces the number of CYC packets stored in the Intel PT buffer at the cost of reduced accuracy.

Disabling Timestamp Packets in Intel(R) Processor Trace (Intel(R) PT)

To drop the current timestamp configuration of Intel(R) Processor Trace (Intel(R) PT), invoke the `!disabletimestamps` command.

```
!disabletimestamps [/here]

/here - Disable on the currently selected processor only

Disable timestamps
```

Extension Commands for Decoding Intel(R) Processor Trace

After code execution is traced, the extension provides the `!decode` command to download the trace, decode it, and display it. The `!decode /info` command displays the decoded amount of instructions. To display the reconstructed execution flow and navigate it, you can invoke `!decode` with no, one, or two range specifiers. If you do not provide the `/number` range specifier, the entire decoded trace is displayed. With two numbers, only the instructions in that decode range are displayed; for example, `!decode 15 23` displays decoded instructions 15 through 23. If you specify only one number, the first or last number of decode lines are printed, depending on whether the number is positive or negative. For example, `!decode 5` displays the first five lines, and `!decode -12` displays the last twelve lines.

In general, the `!decode` command displays the disassembly of all the decoded instructions. If symbols are available, addresses are displayed with symbol decoration. If sources are available and you want the trace decode to be annotated with source lines that were executed, you can set the `/src` option. If you only want source lines to be displayed, the `/srconly` option suppresses the display of instruction disassembly lines.

When source line annotation is requested, the beginning of the source line shows the source location as [`<source file name>:<line number>`] ; for example, [`mysourcefile.c:519`] . If debugger markup language (DML) is supported and enabled, these source locations are displayed as links – clicking such a link opens the source file and scrolls to the respective line.

To show timestamp information in decoded output, use the `/timestamps` option. See section [Timestamps](#) for information about enabling Intel Processor Trace timestamp packets. Additionally, you can specify the source of the base frequency for decoding the timestamps by selecting one of the following:

- `/cpuid_0x15_eax` and `/cpuid_0x15_ebx` - the Core Crystal Clock frequency value returned by the CPUID instruction is used for decoding.

You have to specify both options for this to work.

- `/nom_freq` - the nominal frequency specified in the model-specific register `MSR_PLATFORM_INFO[15:8]` is used for decoding.

```

!decode [/info] [/src] [/srconly] [/symbols] [/timestamps] [/offline] [/topa]

[ /family <expr> ] [ /model <expr> ] [ /stepping <expr> ]

[ /cpuid_0x15_eax <expr> ] [ /cpuid_0x15_ebx <expr> ] [ /output_base <expr> ]

[ /output_mask <expr> ] [<n>] [<start>] [<end>]

/info - Get decode information

/src - Interleave source lines (if available)

/srconly - Show only source lines (if available)

/symbols - Show export symbol names (if available)

/timestamps - Show timestamps (if available)

<n> - Number of lines to display;

    if positive print first n lines,
    if negative print last n lines

    (cannot be used together with <start> and <end>)

    (space-delimited, base 10)

<start> - Start of line number range (use together with <end>)

    (space-delimited, base 10)

<end> - End of line number range (use together with <start>)

    (space-delimited, base 10)

/family <expr> - Extended CPU Family ID to be used during decode (space-delimited)

/model <expr> - Extended CPU Model ID to be used during decode (space-delimited)

/stepping <expr> - CPU Stepping ID to be used during decode (space-delimited)

/cpuid_0x15_eax <expr> - CTC frequency configuration(cpuid_eax(0x15, 0)) (space-delimited)

/cpuid_0x15_ebx <expr> - CTC frequency configuration(cpuid_ebx(0x15, 0)) (space-delimited)

/offline - Decode Intel(R) Processor Trace data without connection to a target.

    (Following values have to be specified)

/output_base <expr> - Start address of available buffer (MSR 0x560) (space-delimited)

/output_mask <expr> - Size of available buffer (MSR 0x561) (space-delimited)

/topa - Specify ToPA output config option (single range is default)

Decode Intel(R) Processor Trace and display reconstructed execution history

```

Using the Extension

This example demonstrates how to set up a target for tracing and how to record a trace with the debugger.

Hardware used for this example:

- 6th generation Intel(R) Core™ processor target
- Windows* 8.1 Enterprise 64-bit version
- 8GB installed memory (RAM)
- FireWire (IEEE1394) connection to WinDbg*.

Reserving Memory on the Target for Trace Data

This section describes one method to reserve memory. For more information on reserving memory, see also the [Output Configuration](#) section.

For further information on Windows*, memory, and limitations, see also:

- [Mark Russinovich's Blog: Pushing the Limits of Windows: Physical Memory](#)
- [Memory Limits for Windows and Windows Server Releases](#)
- [How graphics cards and other devices affect memory limits.](#)

For further information on the BCD, see also:

- [Boot Configuration Data Editor Frequently Asked Questions.](#)

Note

Improper changes to the BCD system store can prevent Windows* from starting. Review the commands and their results carefully before restarting Windows.

To remove (reserve) memory on the target for Intel(R) Processor Trace (Intel(R) PT) data:

1. Run the CLI as administrator.
2. Check the physical memory.

```
systeminfo | find "Total Physical Memory"  
-> Total Physical Memory: 7,816 MB
```

3. Remove 1GB (from the currently booted OS).

```
bcdedit /set removememory 1024
```

4. Reboot the target.

```
shutdown /r /t 0
```

5. After reboot, check the physical memory to make sure it worked.

To **undo** the changes on the target (after finishing the debug/trace session):

```
systeminfo | find "Total Physical Memory"  
-> Total Physical Memory: 6,824 MB
```

1. Run the CLI as administrator.

2. Delete the value from the configuration.

```
bcdedit /deletevalue removememory  
-> The operation completed successfully.
```

Configuring a Trace Region in Memory

1. Launch WinDbg*.

2. Connect to the target.

3. Break into the debugger and perform these steps to use the reserved memory for trace data:

a. Load the ptext32 extension for a 32-bit or sysdbg64 for a 64-bit version of WinDbg*; for example:

```
.load sysdbg64
```

b. Configure an output region at 7GB with 64KB for each processor:

```
!setoutputconfig /start 0x1C0000000 /size 0x80000
```

Tracing a Windows* Kernel Function

To trace the frequently called `nt!KiSwapThread` Windows* kernel function, perform the following steps:

1. Set an IP filter (to avoid tracing debugger communication):

```
!setipfilter nt!KiSwapThread (nt!KiSwapThread+0x6e1)
```

2. Set a breakpoint at that function:

```
bp nt!KiSwapThread
```

3. Enable tracing, run twice (the first hit stops at that function, the second hit stops after tracing), and disable tracing:

```
!enable; g; g; !disable
```

4. Decode:

```
!decode;
```

WinDbg* dumps the recorded trace data:

```
6: kd> !decode

fffff803`bcc4adae 440fb6f0      movzx   r14d,al
fffff803`bcc4adb2 488bbbc8000000  mov      rdi,qword ptr [rbx+0C8h]
fffff803`bcc4adb9 0fba73740a     btr      dword ptr [rbx+74h],0Ah
fffff803`bcc4adbe 4c89a424b0000000  mov      qword ptr [rsp+0B0h],r12
fffff803`bcc4adc6 723b          jb      nt!KiSwapThread+0x1a3 (fffff803`bcc4ae03)
fffff803`bcc4adc8 4584f6          test    r14b,r14b
fffff803`bcc4adcb 0f858d010000  jne      nt!KiSwapThread+0x2fe (fffff803`bcc4af5e)
fffff803`bcc4add1 400fb6cd      movzx   ecx,bpl
fffff803`bcc4add5 440f22c1      mov      cr8,rcx
fffff803`bcc4add9 4c8b742460  mov      r14,qword ptr [rsp+60h]
```

Reconstructing Program Execution Flow

You can record and reconstruction the flow of your program execution by using the Last Branch Record (LBR) functionality. Follow the steps below:

1. Power the target on and let Windows* start.

2. Launch Intel(R) Debug Extensions for WinDbg*. The `sysdbg64.dll` is loaded automatically.

3. Enable recording of LBR by executing the command below:

```
!enablelbr
```

Note

Once LBR is enabled, Intel(R) Processor Trace (Intel(R) PT) is disabled as these methods are incompatible.

4. Press F5 or type g in the terminal to run the target for a while.

5. Stop the target by pressing the break button in the toolbar.

6. Display reconstructed flow with the following command:

```
!showlbrflow
```

The complete reconstructed flow is displayed in the terminal. The initial instruction corresponds to the first recorded branch and the final instruction corresponds to rip. See an example output below:

```
Flow reconstruction from Last Branch Records

fffff807'126621f5 493bc7      cmp    rax,r15
fffff807'126621f8 0f82e0d51700  jb     nt!HalpTimeStallExecutionProcessor+0x17d6ee
(fffff807'127df7de)
fffff807'126621f5 493bc7      mov    rcx,rsi
fffff807'126621f5 493bc7      mov    r15,rax
fffff807'126621f5 493bc7      sub    rcx,qword ptr [rsp+30h]
...
```

Alternatively, you can print blocks as follows:

```
!showlbr
```

Example output:

LBR stack

Address	Function
0xfffff8071264202b	nt!PpmConvertTime+0x3b
0xfffff8071264200a	nt!PpmConvertTime+0x1a
0xfffff807127e9789	nt!PpmUpdateTimeAccumulation+0x16ae6d
0xfffff8071267e98a	nt!PpmUpdateTimeAccumulation+0x6e
...	

! Note

Flow is reconstructed only if LBRs contain proper data. If Model-specific Registers (MSRs) associated with LBR are 0, the commands above will not return the flow. Use the command below for validation.

7. To check what LBRs have recorded, execute the following command:

```
!showmsr
```

Example output:

```
List of MSR Last Branch Records

Actual branch (TOS):28

TOS MSR_LASTBRANCH_x_FROM_IP MSR_LASTBRANCH_x_TO_IP

0 0xfffff807126621e4 0xfffff8071266238c
1 0xfffff807126623a0 0xfffff807126621e9
2 0xfffff807126621f0 0xfffff807127a63e0
3 0xfffff807127a63f3 0xfffff807127a6411
3 0xfffff807127a63f3 0xfffff807127a6411
...
```

You can also import LBRs to decode. Execute the following command to import LBR raw data separated by tabs:

```
!importlbr <LIP> <MSR_LBR_TOS> <LBR_FROM[0] LBR_TO[0] LBR_FROM[1] LBR_TO[1] ... LBR_FROM[31]
LBR_TO[31]>
```

The output contains blocks of addresses and symbols. See more details about this command [here](#).

See an example of the command and its output below:

```
> !importlbr 0xFFFFF8054D281382 0x0000000000000019 0xFFFFF8054D28878D 0xFFFFF8054D289333

LBR stack
Address          Function
...
0xfffff80543329abc nt!PpmUpdatePerformanceFeedback+0x16c
0xfffff80543329c11 nt!PpmUpdatePerformanceFeedback+0x2c1
...
```

Commands

- Enable or disable recording:

```
!enablelbr
```

Enable LBR recording. If CPU goes to sleep deeper to C2, the flag is cleared, so you must re-enable it manually. This command disables Intel(R) Processor Trace (Intel(R) PT).

```
!disablelbr
```

Disable LBR recording.

- Import LBRs to decode:

```
!importlbr <LIP> <MSR_LBR_TOS> <LBR_FROM[0] LBR_TO[0] LBR_FROM[1] LBR_TO[1] ... LBR_FROM[31]
LBR_TO[31]>
```

Import LBR raw data separated by tabs and print blocks of addresses and symbols. To print the flow of a block, click a corresponding link displayed in the output for the `!importlbr` command. The raw data must correspond to valid addresses in the actual memory space.

- Display data:

```
!showmsr
```

Print the value of the MSRs corresponding to LBRs. First it prints the value of Top of Stack (TOS), which is the last branch recorded. Then it prints two columns of MSRs. If the value of MSRs is 0, nothing has been recorded and reconstruction is impossible.

```
!showlbr
```

Print blocks of addresses and symbols. To print the flow of a block, click a corresponding line link.

```
!showlbrflow
```

Print the complete reconstructed flow from all LBRs.

- Select data to record:

```
!selectring0lbr
```

Enable recording branches when CPU is on ring zero.

```
!filterring0lbr
```

Disable recording branches when CPU is on ring zero.

```
!selectring1to4lbr
```

Enables recording branches when CPU is on any of the rings one to four.

```
!filterring1to4lbr
```

Disable recording branches when CPU is on any of the rings one to four.

```
!enablepowerevent [/here]
```

Enable power event tracing.

```
!disablepowerevent [/here]
```

Disable power event tracing.

where:

```
/here
```

Option to enable or disable power event tracing on the currently selected processor only.

See also

- [Intel\(R\) 64 and IA-32 Architectures Software Developer Manuals](#)
- [How to configure LBR \(Last Branch Record\) on Intel CPUs](#)

Debugging Advanced Configuration and Power Interface (ACPI)

Advanced Configuration and Power Interface (ACPI) is a framework that forms a subsystem within the host OS and serves as an interface layer between the system firmware (BIOS) and the host OS. This framework establishes a hardware register set to define power states (sleep, hibernate, wake, etc). As a result, the standard ACPI framework and the hardware register set enable power management and system configuration without directly calling firmware natively from the OS.

WinDbg* over the Intel(R) Direct Connect Interface (Intel(R) DCI) enables using the Microsoft* AMLI Debugger to debug Advanced Configuration and Power Interface (ACPI) Machine Language (AML) code. With WinDbg* over Intel(R) DCI, the Microsoft* AMLI Debugger is used as an extension to a native kernel debugger.

For more information about the framework structure, refer to the [ACPI documentation](#).

Before You Begin

! Important

For Windows* 10 version lower than 1803, checked builds of the Windows* ACPI driver (Acpi.sys) are used. Starting from version 1803, checked builds are no longer required for using Microsoft* AMLI Debugger.

Due to nature of JTAG hardware debugger design, Windbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) supports only commands from AMLI Debugger extensions. These extensions have a syntax of `!amli` command.

For more information, refer to the [Microsoft* AMLI Debugger documentation](#).

! Note

Executing debugger commands from the interactive Microsoft AMLI Debugger mode is not supported. The [extension reference](#) in this user guide describes exactly what is supported by WinDbg* over Intel(R) DCI. Please check the included reference first before referring to the general Microsoft* documentation.

Before you begin ACPI debugging, complete the following setup steps:

1. Ensure that the target system has the appropriate version of the Acpi.sys driver installed.
2. Connect to WinDbg* over Intel(R) DCI.
3. Reload ACPI symbols by executing the following command:

```
!reload /f acpi.sys
```

1. Configure the Microsoft* AMLI Debugger via the `!amli set` extension. For best experience, specify all options listed in the referred page.

Supported ACPI Extensions

WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) supports the following Advanced Configuration and Power Interface (ACPI) extensions:

!amli set

Sets or displays the Microsoft* AMLI Debugger options.

```
!amli set [<Options>]
```

Options parameter specifies one or more options to be set. Multiple options must be separated with spaces. Possible values include:

verboseon

Causes full debug output to be sent from the target computer. This option should be enabled at all times for effective AML debugging.

traceon

Activates ACPI tracing. This option is useful for tracking SMI-related hard hangs.

errbrkon

Breaks into the AMLI Debugger whenever the interpreter has a problem evaluating AML code.

spewon

Suppresses debug output.

dbgbrkon

Enables breaking into the AMLI Debugger.

If no options are specified, the current status of all options is displayed.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-set>

!amli dns

Displays the ACPI namespace for a particular object, the namespace tree subordinate to that object, or the entire namespace tree. This command is especially useful in determining what a particular namespace object is – whether it is a method, a fieldunit, a device, or another type of object.

```
!amli dns [/s] [<Name> | <Address>]
```

- */s* flag is set to recursively display the entire namespace subtree under the specified object.
- *Name* parameter specifies the namespace path.
- *Address* specifies the address of the namespace node.

If neither *Name* nor *Address* is specified, the entire ACPI namespace tree is displayed recursively (regardless of the */s* flag state).

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-dns>

!amli find

Finds an ACPI namespace object. The extension takes the name of any namespace object and returns its full path.

```
!amli find <Name>
```

Name parameter specifies the name of the namespace object (the final segment of the full path).

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-find>

!amli u

Unassembles the Advanced Configuration and Power Interface (ACPI) Machine Language (AML) code.

```
!amli u [<MethodName> | <CodeAddress>]
```

- *MethodName* parameter specifies the full path of the method name to be disassembled.
- *CodeAddress* parameter specifies the address of the AML code where disassembly must begin.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-u>

!amli lc

Displays brief information about all active Advanced Configuration and Power Interface (ACPI) contexts

```
!amli lc
```

If many contexts are displayed, the current one (the one being executed by the interpreter at the moment) is marked with an asterisk.

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-u>

!amli r

Displays detailed information about the current context of the interpreter or the specified context. You can use it when the Microsoft* AMLI Debugger prompt appears after an error is detected.

```
!amli r [<ContextAddress>]
```

ContextAddress parameter specifies the address of the context block to be displayed. You can get the address of a context block from the **Ctxt** field in the `!amli lc` extension output.

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-r>

!amli dl

Displays a portion of the AML interpreter event log.

```
!amli dl
```

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-dl>

!acpicache

Displays all of the Advanced Configuration and Power Interface (ACPI) tables cached by the Hardware Application Layer (HAL)

```
!acpicache [<DisplayLevel>]
```

DisplayLevel parameter specifies the detail level of the display. Possible values are 0 (default) for an abbreviated display and 1 for a more detailed display.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-acpicache>

!acpiinf

Displays information on the configuration of the Advanced Configuration and Power Interface (ACPI), including the location of system tables and the contents of the ACPI fixed feature hardware.

```
!acpiinf
```

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-acpiinf>

!acpiirqarb

Displays the contents of the Advanced Configuration and Power Interface (ACPI) IRQ arbiter structure, which contains the configuration of I/O devices to system interrupt controller inputs and processor Interrupt Dispatch Table (IDT) entries.

```
!acpiirqarb
```

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-acpiirqarb>

!amli dh

Displays the internal heap block of the Advanced Configuration and Power Interface (ACPI) Machine Language (AML) interpreter.

```
!amli dh [<HeapAddress>]
```

HeapAddress parameter specifies the address of the heap block. If the parameter is omitted, the global heap is displayed.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-amli-dh>

!fac

Displays a Firmware Advanced Configuration and Power Interface (ACPI) Control Structure (FACS).

```
!fac <Address>
```

Address parameter specifies the address of the FACS.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-fac>

!fad

Displays a Fixed Advanced Configuration and Power Interface (ACPI) Description Table (FADT).

```
!fad <Address>
```

Address parameter specifies the address of the FADT.

! See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-fad>

!mapic

Displays an Advanced Configuration and Power Interface (ACPI) Multiple APIC Table.

```
!mapic <Address>
```

Address parameter specifies the address of the Multiple APIC Table.

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-mapic>

!nsobj

Displays an Advanced Configuration and Power Interface (ACPI) namespace object.

```
!nsobj [<Address>]
```

Address parameter specifies the address of the namespace object. If the parameter is omitted, the namespace tree root is displayed.

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-nsobj>

!nstree

Displays an Advanced Configuration and Power Interface (ACPI) namespace object and its children in the namespace tree.

```
!nstree [<Address>]
```

Address parameter specifies the address of the namespace object. If the parameter is omitted, the entire namespace tree is displayed.

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-nstree>

!rsdt

Displays the Advanced Configuration and Power Interface (ACPI) Root System Description Table.

```
!rsdt
```

See also

<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-rsdt>

Debugging Use Cases

Check the following examples of using the extensions:

Investigating a Frozen System

If the target system has frozen and you suspect it might be an Advanced Configuration and Power Interface (ACPI) problem, use the `!amlci lc` extension to display an active contexts.

If no contexts are displayed, the error is probably not related to ACPI.

If several contexts are displayed, find the one marked with an asterisk. This is the current context (the one that is being executed by the interpreter at the present moment). See an example output below:

```
kd> !amlci lc

Ctxt=fffffe001b8667540, ThID=0000000000000000, Flgs=A--C----, pbOp=0000000000000000,
Obj=\_SB.PCIO.LPCB.H_EC._QFO
```

The abbreviation **pbOp** indicates the instruction pointer (pointer to binary op code). The **Obj** field gives the full path and the name of the method as it appears in the ACPI tables.

For more information, see `!amlci lc`. You can use the `!amlci u` command to disassemble specific method.

Additionally, you can use the `!amlci dl`.

Tracing ACPI BIOS Code

You can use Advanced Configuration and Power Interface (ACPI) tracing to troubleshoot low-level platform issues. Follow the steps below to receive ACPI trace messages.

Important

Run all commands in the Intel(R) Debug Extensions for WinDbg* command prompt.

1. Connect to your target.
2. Set a post-reset breakpoint by running the following command:

```
forensic.setinitbreakpoint()
```

3. Reset your target:

```
itp.resettarget()
```

4. Remove the reset breakpoint:

```
forensic.removeinitbreakpoint()
```

5. Enable ACPI trace messages:

```
acpi.amlitrace()
```

The Intel(R) Debug Extensions for WinDbg* command prompt prints ACPI trace messages. To cancel tracing, press Ctrl + C.

WinDbg* Support for Microsoft* Hyper-V

This chapter describes the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension that provides support for Microsoft* Hyper-V. This extension enables Intel(R) System Debugger functionality with Intel(R) Virtualization Technology (Intel(R) VT) for IA-32, Intel(R) 64 and Intel(R) Architecture within Microsoft* Hyper-V environment with Virtualization-based security (Microsoft* VBS) enabled.

See also

[Intel\(R\) Virtualization Technology \(Intel\(R\) VT\) for IA-32, Intel\(R\) 64 and Intel\(R\) Architecture Product Page](#)

Introducing Microsoft* Hyper-V

Microsoft* Hyper-V virtualization is the type 1 Hypervisor. This virtualization product is part of the Microsoft* Virtualization-based Security (Microsoft* VBS) solution. This WinDbg* extension is designed to provide JTAG system debugger functionality for Microsoft implementation of Virtual Machine Monitors.

For more information on presented Microsoft* solutions, refer to the corresponding documentation:

- [Introduction to Microsoft* Hyper-V on Windows* 10](#)
- [Microsoft* Virtualization-based Security](#)
- [Microsoft Virtualization Based Security and Hypervisor Enforced Code Integrity for Olympia Users](#) - if you seek explanation of how Microsoft* VBS impacts debugging.

For instructions on setting up and using the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension with Microsoft* Hyper-V, see the [Get Started](#) section.

Setting Up the Extension

Follow the steps below to start using Intel(R) Debug Extensions for WinDbg* for Microsoft* Hyper-V support.

Enable Intel(R) Virtualization Technology (Intel(R) VT) for IA-32, Intel(R) 64 and Intel(R) Architecture in BIOS

1. Power on the system and open the BIOS.
2. Open the **Processor** submenu. The processor settings menu may be located under any of **Chipset**, **Advanced CPU configuration**, or **Northbridge** tabs.
3. Enable Intel(R) Virtualization Technology (Intel(R) VT) for IA-32, Intel(R) 64 and Intel(R) Architecture. The settings might depend on the OEM and system BIOS.
4. If the option is available, enable Intel(R) Virtualization Technology (Intel(R) VT) for Directed I/O (Intel(R) VT-d).
5. Click **Save** and exit the BIOS.

Enable Microsoft* Hyper-V on Windows* 10

System Requirements

- Windows* 10 Enterprise, Pro, or Education 64-bit Processor.
- CPU support for Virtual Machine Monitor Node Extension (Intel(R) Virtualization Technology (Intel(R) VT) for Connectivity (Intel(R) VT-c) on Intel CPUs).
- Minimum of 4 GB memory.

Note

Microsoft Hyper-V is built into Windows* OS as an optional feature, no direct download needed.

Enable Microsoft Hyper-V to create virtual machines on Windows* 10 using any way suggested below:

- Enable Microsoft Hyper-V using Windows* PowerShell:

1. Launch the Windows PowerShell console as Administrator.
2. Run the following command:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
```

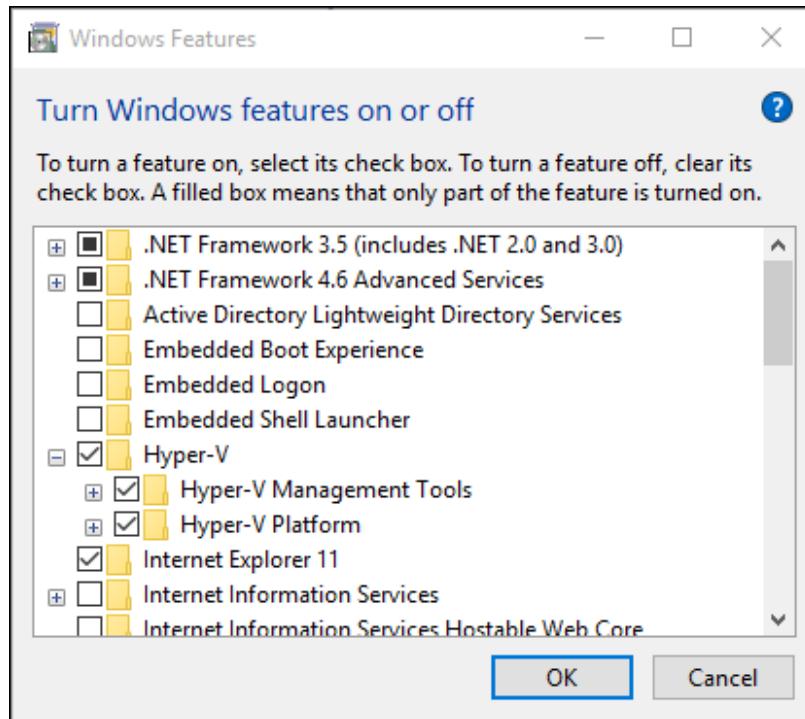
3. When the installation is completed, reboot the system.

- Enable Microsoft Hyper-V through Settings:

! Note

The Hyper-V role cannot be installed on Windows* 10 Home.

1. Open Windows Settings and navigate to Apps > Apps & features.
2. On the right, click Programs and Features under Related settings.
3. Select Turn Windows features on or off.
4. Check the Hyper-V root box:



5. Click OK and exit the settings.

Configure the Target Windows* OS

1. On the target system, execute the following commands from the command line:

```
bcdedit /debug on  
bcdedit /set hypervisordebug on  
bcdedit /set hypervisorlauchtype auto
```

Additionally, if you want to debug the process of loading the hypervisor, run the following command:

```
bcdedit /set bootdebug on
```

2. Reboot the target system.

To support Microsoft Hyper-V environment within the Virtual machine monitor (VMM) context or guest context, Extended Page Tables (EPT) address translation is fully integrated. Thus, all **dbgeng** use cases are transparently supported.

Note

The following virtualization-related symbols are currently available only to the customers with a special NDA contract signed with Microsoft:

- storvsp.pdb
- vhdparsers.pdb
- passthroughparser.pdb
- hvix64.pdb
- hvloader.pdb

Without a **.pdb** file, the debugger cannot resolve the function names or any global objects in the context of the symbols enumerated above. Beside the symbol resolution support, other features are available for all users.

Virtual Machine Architecture

Virtual-machine extensions define processor-level support for virtual machines on Intel(R) processors. Virtual machine monitors (VMM) or Hypervisors can belong to the following two types:

- **Hypervisor Type 1**

Native or BareMetal Hypervisor, which acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents the guest software (virtual machine) with the abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM can retain selective control of processor resources, physical memory, interrupt management, and I/O. Microsoft* Hyper-V belongs to this hypervisor type.

- Hypervisor Type 2

Also known as a hosted Hypervisor. In case of this hypervisor type, each virtual machine is a guest software environment that supports a stack containing the operating system and the application software. The software stack acts as if it were running on a platform without a hypervisor. Each virtual machine operates independently and uses the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software executing in a virtual machine must operate with reduced privilege so that the VMM can retain control of platform resources.

! Note

This hypervisor type is not natively supported by the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension for Microsoft* Hyper-V.

Guest Environment

Every virtual machine is a guest software environment (guest OS). It is an operating system and application software executing above a Virtual Machine Monitor (VMM) that presents the guest software. Each guest environment operates independently from other virtual machines but uses the same interfaces to system resources (memory, graphics, processor interfaces) provided by a physical platform. The behavior of the guest environment does not depend on whether it runs on a VMM or not.

! Note

The guest environment executes at a virtually reduced privilege level so that the VMM has control over system resources. Certain operations trap into the VMM giving it full control of the responses, management of resources, and the platform overall.

Virtual Machine Extension (VMX) operation

Virtual Machine Extension (VMX) operation is a form of processor operation that supports virtualization. TVMX operation belong to the following types:

- **VMX root operation** - executed by the Virtual machine monitor (VMM).

Processor behavior in the VMX root operation differs from the usual one by the following:

- Set of new instructions (VMX instructions) is available
- Values that can be loaded into control registers are limited.

- **VMX non-root operation** - executed by the guest software (virtual machine)

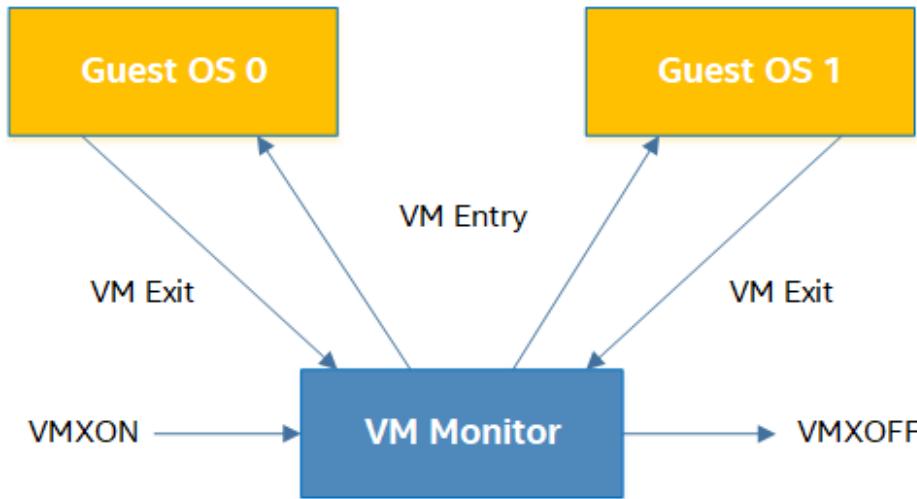
Processor behavior in the VMX non-root operation is modified to facilitate virtualization and the software functionality is limited due to ordinary operations replaced with VM exits (see below). This limitation allows the VMM to retain control of processor resources.

Transitions between VMX root and non-root operations - VMX transitions - are of two types either:

- **VM entry** - transition into the VMX non-root operation.
- **VM exit** - transition from the VMX non-root operation to the VMX root one.

VMM Software Lifecycle

The figure below illustrates the lifecycle of a Virtual Machine Monitor (VMM), its guest software (a virtual machine), and the interactions between them.



1. Software enters the VMX operation by executing the `VMXON` instruction;
2. VM exits transfer control to an entry point specified by the VMM.
3. The VMM might act appropriate to the cause of the VM exit and return to the guest software using a VM entry.
4. Software exits the VMX operation by executing the `VMXOFF` instruction.

VMX Operation Restrictions

In a VMX operation, processors may fix certain bits in CR0 and CR4 to specific values and not support other values. Thus, if any of these bits contain an unsupported value, the `VMXON` instruction fails. Any attempt to set one of these bits to an unsupported value while in VMX operation (including VMX root operation) using any of the CLTS, LMSW, or MOV CR instructions causes a general-protection exception. VM entry or VM exit cannot set any of these bits to an unsupported value.

The first processors to support VMX operation require that the following bits be 1 in VMX operation:

- CR0.PE
- CR0.NE
- CR0.PG
- CR4.VMXE

Note

The restrictions on CR0.PE and CR0.PG imply that VMX operation is supported only in the paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in the unpaged protected mode or in the real-address mode.

Future processors might require a different amount of memory to be reserved. In this case, the VMX capability-reporting mechanism reports this change to the software.

Address Translation in VMX Operation

The architecture for VMX operation includes two features that support address translation:

- **Virtual-processor identifiers (VPIDs)** that manage translations of linear addresses.

Note

This feature is not supported by the present extension.

- **Extended page-table (EPT) mechanism** that supports the virtualization of physical memory.
When the EPT mechanism is used, certain physical addresses used to access memory are treated as guest-software addresses. As a result, guest-software addresses are translated by traversing a set of EPT paging structures to produce physical addresses used to access memory.
This feature allows the hypervisor to gain control over the resources.

VMX Instructions for Intel(R) 64 and IA-32 Architectures

Virtual-machine extensions (VMX) are designed to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

For more information, see [Intel\(R\) 64 and IA-32 Architectures Software Developer Manuals](#).

Instructions for Managing the Intel(R) Virtual Machine Control Structure Shadowing (Intel(R) VMCS Shadowing)

VMPLRD

Takes a single 64-bit source operand that is in memory. The instruction makes the referenced Intel(R) VMCS Shadowing active and current by loading the pointer to the current Intel VMCS Shadowing with the taken operand and establishes the current Intel VMCS Shadowing based on the contents of Intel VMCS Shadowing data area in the referenced Intel VMCS Shadowing

region. As this operation makes the referenced Intel VMCS Shadowing active, a logical processor may start maintaining some Intel VMCS Shadowing data for the Intel VMCS Shadowing.

VMPTRST

Takes a single 64-bit destination operand that is in memory. The pointer to the current Intel(R) VMCS Shadowing is stored in the destination operand.

VMCLEAR

Takes a single 64-bit operand that is in memory. The instruction sets the launch state of the Intel(R) VMCS Shadowing referenced by the operand to “clear”, renders that Intel VMCS Shadowing is inactive, and ensures that data for the Intel VMCS Shadowing have been written to the Intel(R) VMCS Shadowing data area in the referenced Intel VMCS Shadowing region. If the operand is the same as the pointer to the current Intel(R) VMCS Shadowing, that pointer becomes invalid.

VMREAD

Reads a component from the Intel(R) VMCS Shadowing (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or a memory unit.

VMWRITE

Writes a component to the Intel(R) VMCS Shadowing (the encoding of that field is given in a register operand) from a source operand that may be a register or a memory unit.

Instructions for Managing the VMX Operation

VMLAUNCH

Launches a virtual machine managed by the Intel(R) Virtual Machine Control Structure Shadowing (Intel(R) VMCS Shadowing). A virtual machine (VM) entry occurs, transferring control to the VM.

VMRESUME

Resumes a virtual machine managed by the Intel(R) VMCS Shadowing. A virtual machine (VM) entry occurs, transferring control to the VM.

VMXOFF

Causes the processor to leave the VMX operation.

VMXON

Takes a single 64-bit source operand that is in memory. The instruction causes a logical processor to enter the VMX root operation and to use the memory referenced by the operand to support the VMX operation.

Instructions for Managing the VMX-specific Translation Lookaside Buffer (TLB)

INVEPT

Invalidate entries in the TLBs and paging-structure caches that were derived from extended page tables (EPT).

INVVPID

Invalidate entries in the TLBs and paging-structure caches based on a Virtual-Processor Identifier (VPID).

! Note

None of the instructions above can be executed in compatibility mode. Otherwise, they generate invalid-opcode exceptions.

Instructions for Using the Guest Software

VMCALL

Allows the software in the VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.

VMFUNC

Allows the software in VMX non-root operation to invoke a VM function (processor functionality enabled and configured by the software in the VMX root operation) without a VM exit.

WinDbg* Support for pre-NT applications

This chapter describes the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension that provides support for pre-NT applications. It provides support to debug pre-NT environment, where kernel is not available.

Commands

To scan memory looking for modules and load the module symbols if they are available.

```
!scanprent
```

To display information about the processors on the system.

```
0: kd> !sysdbg64.cpuid
CP F/M/S Manufacturer Mhz
0 6,167,0 GenuineIntel 3400
1 6,167,0 GenuineIntel 3400
2 6,167,0 GenuineIntel 3400
3 6,167,0 GenuineIntel 3400
4 6,167,0 GenuineIntel 3400
5 6,167,0 GenuineIntel 3400
6 6,167,0 GenuineIntel 3400
7 6,167,0 GenuineIntel 3400
8 6,167,0 GenuineIntel 3400
9 6,167,0 GenuineIntel 3400
10 6,167,0 GenuineIntel 3400
11 6,167,0 GenuineIntel 3400
12 6,167,0 GenuineIntel 3400
13 6,167,0 GenuineIntel 3400
14 6,167,0 GenuineIntel 3400
15 6,167,0 GenuineIntel 3400
```

To display the contents of the MTRR register.

```
0: kd> !sysdbg64.mtrr
MTRR: var 10, Fixed support: enabled, WC enabled, Default: UC
WB: 00000-0ffff WB: 10000-1ffff WB: 20000-2ffff WB: 30000-3ffff
WB: 40000-4ffff WB: 50000-5ffff WB: 60000-6ffff WB: 70000-7ffff
WB: 80000-83fff WB: 84000-87fff WB: 88000-8bfff WB: 8c000-8ffff
WB: 90000-93fff WB: 94000-97fff WB: 98000-9bfff WB: 9c000-9ffff
UC: a0000-a3fff UC: a4000-a7fff UC: a8000-abfff UC: ac000-affff
UC: b0000-b3fff UC: b4000-b7fff UC: b8000-bbfff UC: bc000-bffff
WP: c0000-c0fff WP: c1000-c1fff WP: c2000-c2fff WP: c3000-c3fff
WP: c4000-c4fff WP: c5000-c5fff WP: c6000-c6fff WP: c7000-c7fff
WP: c8000-c8fff WP: c9000-c9fff WP: ca000-cafff WP: cb000-cbfff
WP: cc000-ccfff WP: cd000-cdff WP: ce000-cefff WP: cf000-cffff
WP: d0000-d0fff WP: d1000-d1fff WP: d2000-d2fff WP: d3000-d3fff
WP: d4000-d4fff WP: d5000-d5fff WP: d6000-d6fff WP: d7000-d7fff
WP: d8000-d8fff WP: d9000-d9fff WP: da000-daff WP: db000-dbfff
WP: dc000-dcff WP: dd000-ddfff WP: de000-defff WP: df000-dffff
WP: e0000-e0fff WP: e1000-e1fff WP: e2000-e2fff WP: e3000-e3fff
WP: e4000-e4fff WP: e5000-e5fff WP: e6000-e6fff WP: e7000-e7fff
WP: e8000-e8fff WP: e9000-e9fff WP: ea000-eaff WP: eb000-ebfff
WP: ec000-ecfff WP: ed000-edfff WP: ee000-eefff WP: ef000-effff
WP: f0000-f0fff WP: f1000-f1fff WP: f2000-f2fff WP: f3000-f3fff
WP: f4000-f4fff WP: f5000-f5fff WP: f6000-f6fff WP: f7000-f7fff
WP: f8000-f8fff WP: f9000-f9fff WP: fa000-faff WP: fb000-fbfff
WP: fc000-fcff WP: fd000-fdff WP: fe000-fefff WP: ff000-fffff

Variable Base Mask Length
0. WB: 0000000080000000 0000007fe0000000 000000020000000
1. UC: 0000000092000000 0000007ffe0000000 000000002000000
2. WB: 0000000000000000 0000007f80000000 000000008000000
3. WC: 00000000facfc000 0000007fffffc000 0000000000004000
4. WB: 0000000100000000 0000007f00000000 0000000100000000
5. WB: 0000000200000000 0000007e00000000 0000000200000000
6. WB: 0000000400000000 0000007c00000000 0000000400000000
7. UC: 0000000000000000 0000000000000000 0000000000000000
8. UC: 0000000000000000 0000000000000000 0000000000000000
9. UC: 0000000000000000 0000000000000000 0000000000000000
```

To display or write data into a specified physical address, see [!db](#), [!dc](#), [!dd](#), [!dp](#), [!dq](#), [!du](#), [!dw](#) and [!eb](#), [!ed](#).

Intel(R) Debug Extensions for WinDbg* for Source Level Debugging

This chapter describes basic debugging operations for source level debugging with Intel(R) Debug Extensions for WinDbg*.

Setting Up Windbg For Source Level Debugging

Complete the following setup steps before performing source-level debugging:

1. Ensure Microsoft SDK version 22000 or windDbg preview is used.
2. Obtain Driver source code Use official MSFT samples from
<https://github.com/microsoft/Windows-driver-samples> to guide your Windows driver development.

Make sure to install Visual Studio 2022 and Windows Driver Kit (WDK) before building your driver.

Windows Driver Frameworks (WDF) is a set of libraries to write high-quality device drivers:
<https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/>

Build driver on host machine, writing down symbol and source file paths.

1. To get target ready for driver testing, execute the following commands in administrator rights to enable test-signed mode and then reboot system. By default, Windows does not load test-signed kernel mode drivers.

```
bcDEDIT /DEBUG ON
```

Enables kernel debugging.

```
bcDEDIT /SET TESTSIGNING ON
```

Enable the loading of test-signed code.

```
bcDEDIT /SET NOINTEGRITYCHECKS ON
```

Enable driver signature enforcement.

Reboot target and install test signed driver:

<https://learn.microsoft.com/en-us/windows-hardware/drivers/install/installing-a-test-signed-driver-package-on-the-test-computer>

1. Configure Windbg over DCI on host machine to use symbols and source files in order to debug on source level driver: Go to Settings > Debugging settings > Debugging path to loading symbols.

- Provide source path to the folder where source file(s) (*.cpp) are located. Separate multiple source folders with a semicolon (;
- Provide symbol path to the folder where symbol file(s) (*.pdb) are located.
- Click OK to save configuration.
- If needed reload can be forced by running the !ld debugger command or !reload <driver module> .

1. Use KMDF debugger extension

- Use .load command from windbg_preview to load the KMDF debugger extension (shipped together with windbg_preview):

```
``.load C:\Program Files\WindowsApps\Microsoft.WinDbg_1.2210.3001.0_x64_8wekyb3d8bbweamd64\winextwdffd.dll``
```

<https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/loading-debugger-extension-dlls>

Debug Operations For Source Level Debugging

Once you set up the debug environment with Intel(R) Debug Extensions for WinDbg*, the following commands and debug view provides capability to debug and analyze target issues:

Breakpoints

The breakpoint can be used in executable code. The following syntax can be used for creating breakpoint that allow you analyze the target and issue debugger command. To manage and display the breakpoints, here are the examples you can used :

- bp module!function : Set a breakpoint on function as soon as module is loaded.
- b1 : List existing breakpoints. Each breakpoint listed has a number in the list.
- bc* : Clear all breakpoints.
- bc number : Clear breakpoint identified by number.
- be number : Enable breakpoint identified by number.
- bd number : Disable breakpoint identified by number.

Run Control, stepping source code

Using flow control to break, go and step In and out of target execution code. Reverse execution is only supported by windbg over SIMICS.

Common KMDF Debugger Extensions

!wdfldr : Displays all loaded KMDF drivers !wfdriverinfo : Displays relevant driver information
!wdfdevicequeues : Displays info related to queue objects that belong to a specified device
!wdfqueue : Displays information about WDFQUEUE object handle !wdfrequest : Displays information about WDFREQUEST object handle !wdffile : Displays information about a specific KMDF handle !wdfiotarget : Displays information about a WDFIOTARGET object handle. !wdfdevice : Displays information associated with a WDFDEVICE object handle !wdfcrashdump : Displays a crash dump that includes WDF log info !wflogdump : Displays WDF log information

Driver source level tracing

- Windbg over DCI supports PT assembly and source level tracing. For more info refer to PT documentation.
- How to use WinDbg to View the KMDF log:

1. Load KMDF extension
2. Locate TMF file for specific WDF version and run !wdftmfile command followed by path to the folder that contains TMF files. !wdftmfile <wdk tmf folder>Wdf01013.tmf
3. Dump the KMDF log for driver (for example osrusbf2) !wflogdump osrusbf2

The screenshot shows the WinDbg interface with several windows open:

- Assembly Window:** Shows assembly code for the `!osrEvtIoDeviceControl` function.
- Command Window:** Shows the command history used to dump the log.
- Breakpoints Window:** Shows a single breakpoint set at line 73 of the assembly code.

Key assembly code from the !osrEvtIoDeviceControl function:

```
51 mov eax, 40h
52 mov eax, 40h
53 mov eax, 40h
54 mov eax, 40h
55 mov eax, 40h
56 mov eax, 40h
57 cmp eax, 5
58 jle osrEvtIoDeviceControl+1
59 mov eax, 40h
60 mov eax, 40h
61 lea rdx, [osrEvtIoDeviceControl+2]
62 mov dx, 31h
63 mov rcx, quore
64 mov rcx, quore
65 mov rcx, quore
66 mov rcx, quore
67 mov dword ptr
68 jmp osrEvtIoDeviceControl+2
69 mov dword ptr
70 mov dword ptr
71 xor rax, rax
72 call osrEvtIoDeviceControl+2
73 mov rax, 98h
74 ret
75 PDEV_CONTEXT device;
76 _PDEV_CONTEXT DevContext;
77 bytesReturned = 0;
78 PBAR_GRAPH_STATE barGraphState = NULL;
79 PSWITCH_STATE switchState = NULL;
80 PUCHAN sevenSegment = NULL;
81 BOOLEAN requestPending = FALSE;
82 NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
83 UNREFERENCED_PARAMETER(InputBufferLength);
84 UNREFERENCED_PARAMETER(OutputBufferLength);
85
86 // If your driver is at the top of its driver stack, EvtIoDeviceControl
87 // at IRQL = PASSIVE_LEVEL.
88
89 // _IRQL_limited_to_(PASSIVE_LEVEL);
90
91 byte ptr
92 PAGE_CODE();
93
```

Key commands entered in the Command window:

```
131: FxPkPnP::PowerPolicyEnterSuspendCompletionRoutine - WDFDEVICE 0xffff8103f1c68f
132: FxPkPnP::PowerPolicyEnterWakeState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f1c68f
133: FxPkPnP::PowerPolicyWaitWakeCompletionRoutine - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f1c68f
134: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
135: FxPkPnP::Dispatch - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d IRP_MJ_PNP
136: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
137: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
138: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
139: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
140: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
141: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
142: FxPkPnP::PowerPolicyEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
143: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
144: FxPkPnP::PowerEnterNewState - WDFDEVICE 0xffff80007efc0c89748 [devobj] 0xffff8103f4e2880d
145: FxPkIo::ResumeProcessingForPower - Power resume all queues of WDFDEVICE 0xffff80007efc0c89748
146: end of log ----
3: kd !wflogdump osrusbf2.sys
Default driver image name: osrusbf2
WDK Library image name: Wdk01009
FxDriverGlobals 0xfffffa103f5f378aa0
WdfBindInfo 0xfffffff8055f57090
Version v1.15
Library module 0xfffff8103ed117e0
ServiceName \Device\Wdf01009
ImagePath \Device\Wdf01009
WDFIVER: 0x00000efc0b0b28d8
Driver logs: Not available
Framework logs: !wflogdump osrusbf2.sys -f
WDFVerifier settings for osrusbf2.sys is OFF
```

Bottom right corner of the screen shows the command `b7: kd>`

WinDbg* Support for RegDB

This chapter describes the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension that provides support for Register Database.

Commands

To set cpu information, introduce the cpu name in specific format. First letter of names in capital letter, without blank spaces.

```
!setcpu <cpuName>
```

To set pch information, we need to introduce the pch name in specific format. First letter of names in capital letter, without blank spaces.

```
!setpch <pchName>
```

To set cpu and pch information, we need to introduce the name in specific format. First letter of names in capital letter, without blank spaces.

```
!settarget <cpuName> <pchName>
```

To load register database xrd file containing the register information.

```
!loadxrd
```

To modify default path to xrd database file.

```
!setxrdpath <xrdPath>
```

To print DML link to base registers.

```
!basereg
```

To get name of MSR by MSR offset.

```
!getmsrname <offset>
```

To enumerate base Architectural registers.

```
!enumarchregs
```

To enumerate base Platform registers.

```
!enumplatfregs
```

To show information about specific Architectural register.

```
!showarchreg <registerName>
```

To read value of Architectural register.

```
!readarchreg <registerName>
```

To write value of Architectural register. Value must be in hexadecimal format.

```
!writearchreg <registerName> <value>
```

To show information about specific Platform register. Parent name is optional. If only register name is specified, all registers that correspond to that name will be printed.

```
!showarchreg <registerName> <parentName>
```

To read value of Platform register. Parent name is optional. If only register name is specified, all registers that correspond to that name will be printed.

```
!readarchreg <registerName> <parentName>
```

To write value of Platform register. Value must be in hexadecimal format.

```
!writearchreg <registerName> <value> <parentName>
```

Using the extension

First, user needs to load the corresponding database. Target platform will be autoconfigured by WinDBG extension.

::

```
0: kd> !loadxrd Intel(R) System Debug extension support for RegDB RegDB: Loaded the
platform registers for AlderLake Successful load of xrd database.
```

Then the architectural and/or platform registers can be printed by using DML links from basereggs.

```
0: kd> !basereggs
Enumeration of base registers:
Architectural Registers
Platform Registers
0: kd> !enumarchregs
Enumeration of thread arch regs:
Name | Offset | Description
Architectural MSRs
IA32_P5_MC_ADDR | 0 | p5 mc addr
IA32_P5_MC_TYPE | 0x1 | p5 mc type
IA32_MONITOR_FILTER_SIZE | 0x6 | monitor filter size
...
0: kd> !enumplatfregs
Enumeration of platform regs:
Name | Description
ADL_Host_Bridge (B0:D0:F0)
RID_CC | Revision ID
PXPEPBAR_0_0_0_PCI | PXPEPBAR PCI
...
```

In order to read or write a platform register value, user should introduce register name and parent name, in order to read a specific register.

```

0: kd> !readplatfreg BCTRL adl_direct_media_interface_bar
Name | Description
Bits | Value
undefined | not defined
0 | 0
SE | SERR# Enable
1 / 0x1
undefined | not defined
7 | 0
0: kd> !showplatfreg BCTRL adl_direct_media_interface_bar
Name | Description
Bits | Write | Read
undefined | not defined
0 | 0 | 1
SE | SERR# Enable
1 / 1 / 1
undefined | not defined
7 | 0 | 1
0: kd> !writeplatfreg BCTRL 0 adl_direct_media_interface_bar
Successful write: 0

```

User can perform the corresponding operations on architectural registers by using the commands for architectural registers.

```

0: kd> !showarchreg STAR
Name | Description | Bits | Write |
Read
SYSCALL_TARGET_EIP | SYSCALL_TARGET_EIP | 0 - 31 | 1 |
1
SYSCALL_CS_SS | SYSCALL_CS_SS | 32 - 47 | 1 |
1
SYSRET_CS_SS | SYSRET_CS_SS | 48 - 63 | 1 |
1
0: kd> !readarchreg STAR
Name | Description | Bits | Value
SYSCALL_TARGET_EIP | SYSCALL_TARGET_EIP | 0 - 31 | 0
SYSCALL_CS_SS | SYSCALL_CS_SS | 32 - 47 | 0
SYSRET_CS_SS | SYSRET_CS_SS | 48 - 63 | 0
0: kd> !writearchreg STAR 0xff
Successful write value 0xff

```

WinDbg* Support for Peripheral Component Interconnect (PCI) Buses

This chapter describes the WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI) extension that provides support for !pci extension.

Commands

To displays the current status of the peripheral component interconnect (PCI) buses.

```
!pci [Flags [Segment] [Bus [Device [Function [MinAddress MaxAddress]]]]]
```

Using the extension

PCI extension support autoconfiguration after launch WinDbg over Intel(R) Direct Connect Interface (Intel(R) DCI) extension.

```
0: kd> !pci
PCI segment 0 Bus 0
00:0 8086:7d10.01 Cmd[0006:.mb...] Sts[0090:c....] Intel Host Bridge SubID:8086:7270

02:0 8086:7d40.00 Cmd[0006:.mb...] Sts[0010:c....] Intel VGA-compatible controller
SubID:8086:2212

04:0 8086:7d03.01 Cmd[0006:.mb...] Sts[0090:c....] Intel Other signal processing controller
SubID:8086:7270

05:0 8086:7d19.01 Cmd[0400:.....] Sts[0010:c....] Intel Other multimedia device SubID:8086:7270

06:0 8086:7e4d.01 Cmd[0506:.mb..s] Sts[0010:c....] Intel PCI-PCI Bridge 0->0x1-0x1

07:0 8086:7eb4.00 Cmd[0400:.....] Sts[0010:c....] Intel PCI-PCI Bridge 0->0x2-0x2b
...
```

You can specific the number of segment, bus or device to display any devices attached to those buses.

See also

[Microsoft WinDbg PCI Buses Instructions Examples](#)

Appendix: Establishing Concurrent Debug Sessions

This section contains instructions on how to set up your systems and debug cables to establish concurrent debug sessions: Kernel* Mode Debugging (KMD) and debugging via Intel(R) Debug Extensions for WinDbg* over Intel(R) Direct Connect Interface (Intel(R) DCI).

Follow these instructions if you need to:

- Debug targets (via KMD sessions) with JTAG via Intel DCI enabled on the UEFI.
- Be able to have concurrent sessions for KMD and JTAG via Intel DCI debugging sessions over one or two cable connections (for example, to be able to use JTAG via Intel DCI debugging when targets do es not react to KDM).

You can configure your system to run concurrent debug sessions with only one debug cable used. If this solution does not work for some reason, you can try two cables. See all instructions below.

1. Choose cables.

To run concurrent debug sessions via one cable, use USB type A-C cable . Connect the C- type end to the target and the A- type end to the host USB3.0 port.

To establish connection via two cables (if the one-cable solution does not work), use the following cables:

- USB type A-C or C-C cable for establishing KMD connection.

Connect the C- type end to the target and the A- type end to the host USB3.0 port.

- USB type A-A cable for establishing JTAG via Intel DCI connection

Connect it to USB3.0 ports. Connect both cables at system boot or plug the type USB A-A cable after booting in the desktop.

2. Configure BIOS. Go to **Intel Advanced Menu > Debug Settings** and set the following:

- Enable **Kernel Debug Patch**
- Set **Platform Debug Consent** to *Manual*
- Inside **Kernel/Platform Debug Support**, enable **DCI Enable** and set **USB DbC Enable Mode** to **USB2**.
- If you use USB type A-C or C-C, enable **USB3 type-C UFP2DFP**.

3. Configure operating system.

Run the following commands:

```
bcdedit /debug on  
bcdedit /dbgsettings usb targetname:test  
bcdedit /set "{dbgsettings}" busparams 0.13.0
```

where 0.13.0 is an example combination of the bus, device, and function number for the USB host controller. For more information, refer to [Microsoft* documentation](#).

Intelligent Debug & Validation Tool (IDV Tool)

Platform designs based on Intel(R) solutions must follow processor power sequence timings, which are currently defined in paper-based electrical specifications. Any specification violation commonly causes platform booting issues, which are commonly solved only by manual analysis and use of expensive tools during first power-on for every platform. As a result, Time to Market for platforms is significantly affected.

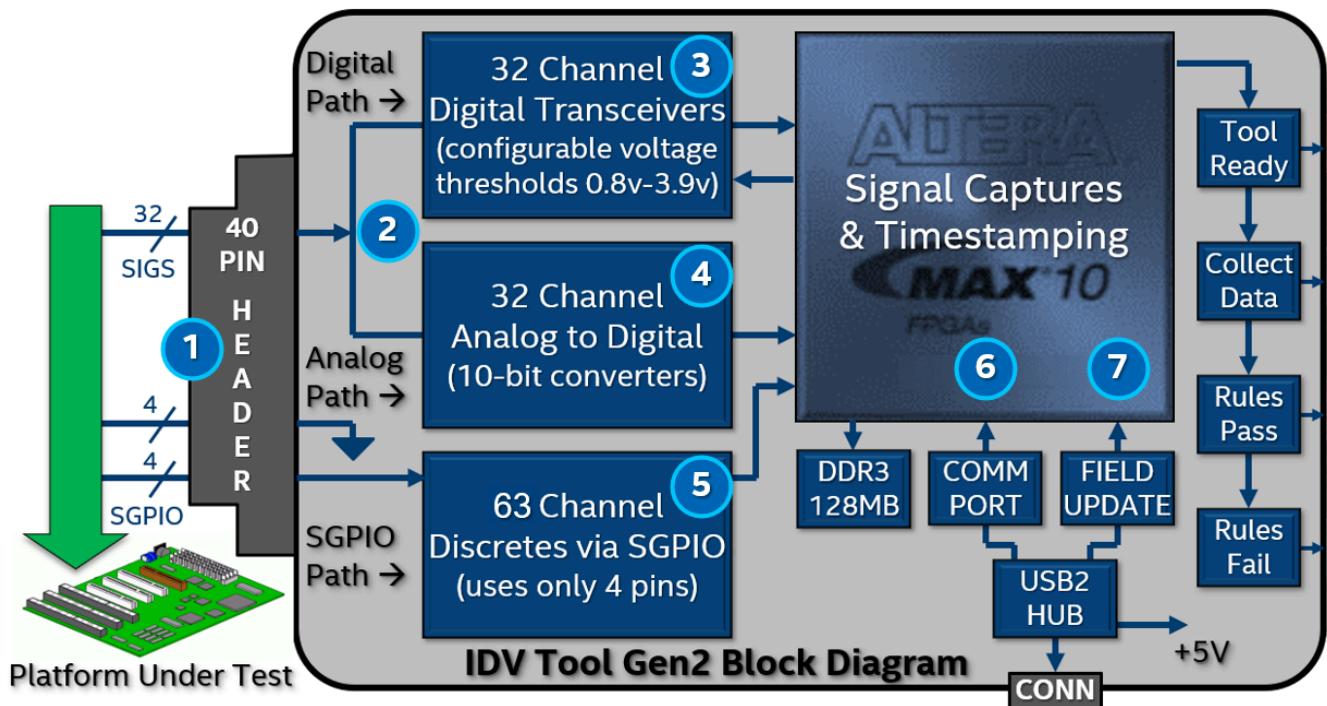
Intelligent Debug & Validation Tool (IDV Tool) provides a solution to resolve power-on issues, reduce Time to Market and development costs, and promote user self-sufficiency. The IDV Tool is fully configurable to automatically collect and timestamp any electrical activity across any number of signals over any time duration and then compare all collections to any set of expected behaviors downloaded to the IDV Tool as “smart” electrical specifications (rule kits). As a result, power-on-reset debug and validation work is automated to seconds and no special experience is required to perform the platform testing.

IDV Tool Hardware Overview

Intelligent Debug & Validation Tool (IDV Tool) setup contains a hardware asset and a software application with GUI. This page provides an overview of the hardware element and interacting with software is covered in other topics.

The IDV Tool hardware is a 32-channel mixed signal oscilloscope and logic analyzer with integrated rule checking. Four LEDs expose the [tool operational status](#).

The setup contains the following elements:



1. 32 signals can be assigned to any header pin as an input (0 to 4.6v max)
2. 32 pins are captured through both digital and analog paths at same time
3. 32 digital transceivers are configured to unique TTL/CMOS voltage thresholds. This is done by the rule kit downloaded to the tool.
4. 32 analog channels are converted to digital using 2 SPI based 10-bit ADCs
5. 63 extra signals can be received serially and captured via 4 SGPIO pins. SGPIO is supported by an increasing number of new Intel(R) platforms.

6. All digital/analog changes are timestamped by FPGA and sent to IDV Tool software via the USB2 interface.

7. IDV Tool hardware supports field FPGA design updates using SW Remote System Update. This allows you to quickly upgrade your IDV Tool hardware with new features under the direct control of IDV Tool software. No JTAG tools are required to accomplish this.

Digital captures occur at 10ns per sample only if logic-levels change. Digital captures are processed by 4 digital rules.

Analog captures occur at 1us per 10-bit sample only if the voltage-levels change. Analog captures are processed by 2 analog rules.

Operational Flow via LEDs

IDV Tool operational flow and status is communicated via the software user interface and four LEDs located on top of the IDV Tool hardware enclosure.

IDV Tool Operational Flow via LEDs



- | | | | |
|-----------|------|------------------------------|--|
| 1. All | OFF | Tool Unpowered | |
| 2. All | ON | Healthy Tool Reset | |
| 3. All | FAST | Factory Image is running | |
| 4. Red | ONLY | Factory Image not loaded | |
| 5. Blue | SLOW | SW/HW communicating | |
| 6. Blue | ON | Tool ready! Power on UUT | |
| 7. Amber | SLOW | Start the trigger data event | |
| 8. Amber | ON | Stop the trigger data event | |
| 9. Green | SLOW | Rule analysis in progress | |
| 10. Green | ON | All rules PASS!!! | |
| 11. Red | ON | Some rules failed | |
| 12. Red | FAST | Tool Issue. Contact Intel | |



1. Tool is unpowered.

2. Tool had a healthy reset.

3. Factory Image is running. This LED state occurs while new FPGA image being installed, or if new FPGA image was not successfully installed.

4. Factory image is not loaded. Contact [Customer Support](#) to update image with JTAG and byte blaster.

5. Software and hardware are communicating

6. Tool hardware is configured and ready capture. Power on the platform under test.

7. Start trigger data event occurred
8. Stop trigger data event occurred
9. Rule analysis is running. This LED state is sometimes not seen if only a few digital and analog captures occurred with only a few rules, but if millions of captures occurred along with 1000's of rules, this state can be seen for a minute or more.
10. All rules have passed.
11. Some rules have passed. Check the console report for exact failure details.
12. Tool has an issue. Contact Customer Support.

Before You Begin

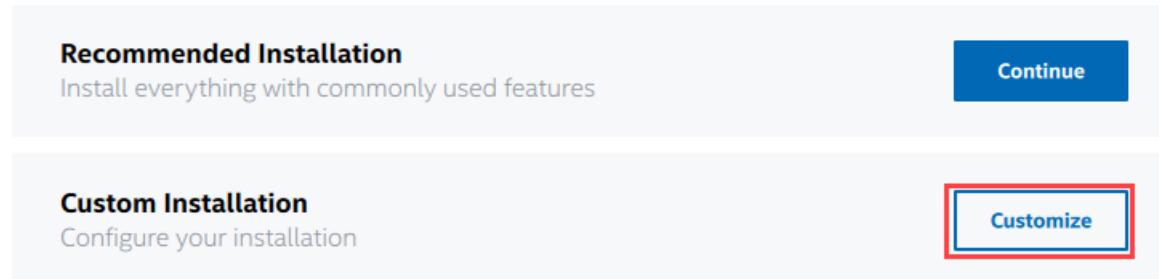
Before you start using Intelligent Debug & Validation Tool (IDV Tool), do the following:

1. Purchase IDV Tool hardware kit via [Design-In Tool Store](#).

1. (Optional) Purchase [Flying Leads Cable Solution](#).
2. (Optional) Purchase [IDV Tool Cable Extender \(Quad 10 pin, 8 inch\)](#).
3. (Optional) Purchase [IDV Tool Cable Extender \(40 pin, 8 inch\)](#).
4. (Optional) Purchase [IDV Tool Cable Extender \(40 pin, 4 inch\)](#).

2. Install IDV Tool as part of Intel(R) System Debugger:

1. [Download Intel\(R\) System Debugger](#).
2. When offered, select **Customize** installation option.



3. Check IDV Tool box (unchecked by default) and proceed with installation.
3. Download the latest FPGA Image for the IDV Tool from the [Resource and Documentation Center](#).
4. Get a rule kit for your target system at [IDV Tool Documentation Library](#) (need to log in).

If you only want to test the IDV Tool, you can download and use an [example rule kit](#).

If no rule kit is available for your target system, you can create one yourself. Reach out to [Customer Support](#) for guidance.

5. Download and unzip the Rule Kit to this exact location: `C:\Users\<user>\.idv\1.0.0\` (to keep all files created and used by IDV Tool stored in one place).

Get Help

To get support or provide feedback, submit a request at [Design-In Tool Store](#).

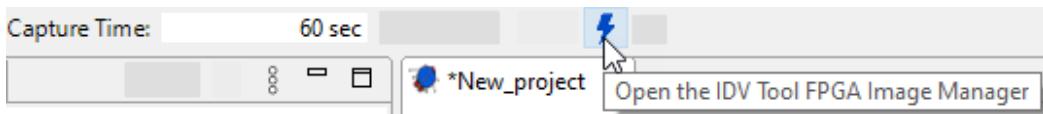
Get Started

1. Connect IDV Tool HW to the system using a USB cable. Four user LED lights go solid.
2. Launch IDV Tool SW: go to `<install-dir>/system_debugger/<version>-nda/idv_tool` and execute `idv_tool.bat`.

`<install-dir>` is the default installation directory of Intel(R) System Debugger (for example, `C:\IntelSWTools` on Windows* OS host).

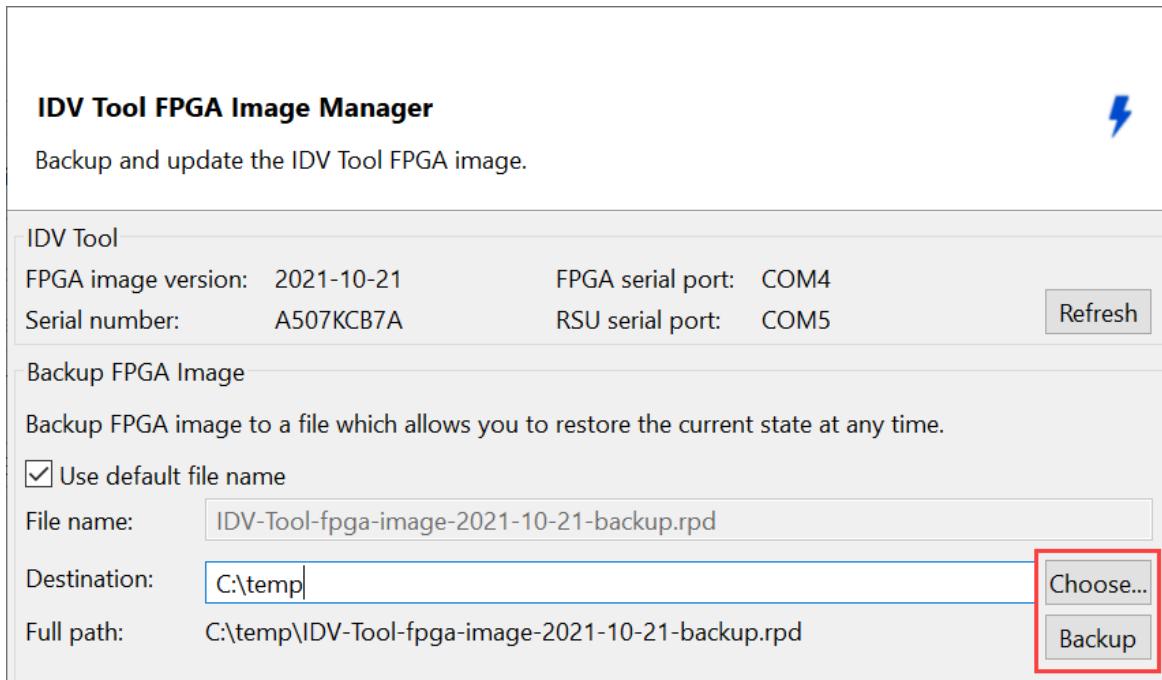
3. Back up and update FPGA Image.

1. Click  to open the IDV Tool FPGA Image Manager and check the FPGA version.

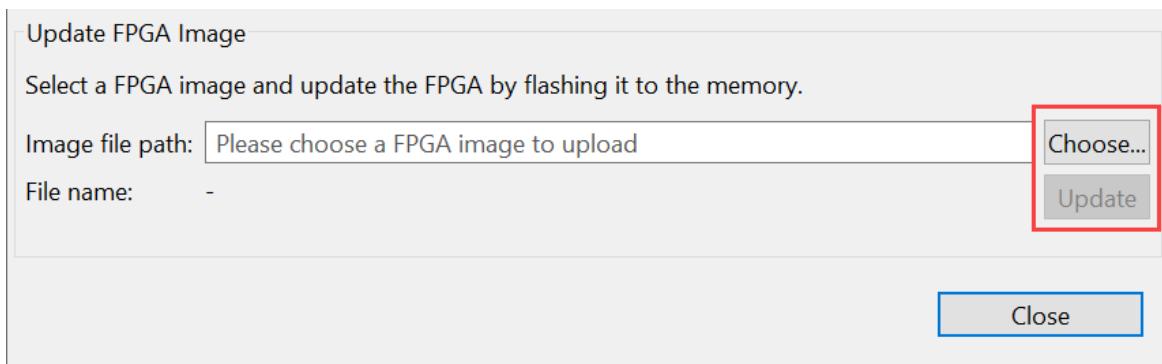


2. Click **Backup** to save the currently installed image.

This step is highly recommended even if you do not need to update the image version.



3. If the installed FPGA image is not of the highest version available, update the image. Click **Choose**, specify the latest FPGA image folder, and click **Update**.



Wait for the update to complete. Four LED lights will blink fast.

4. Click **Close**.

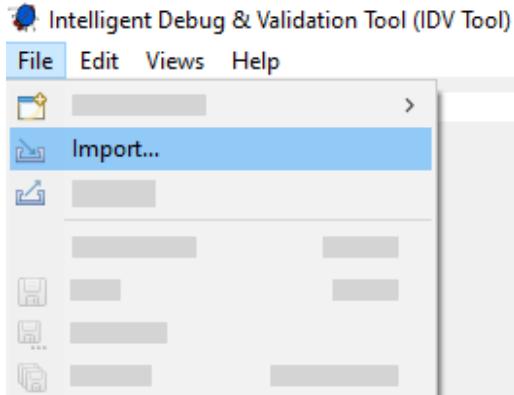
4. If a rule kit for your platform is available at [IDV Tool Documentation Library](#), proceed with **importing** it.

Otherwise, contact [Customer Support](#) for guidelines on how to create one for your target.

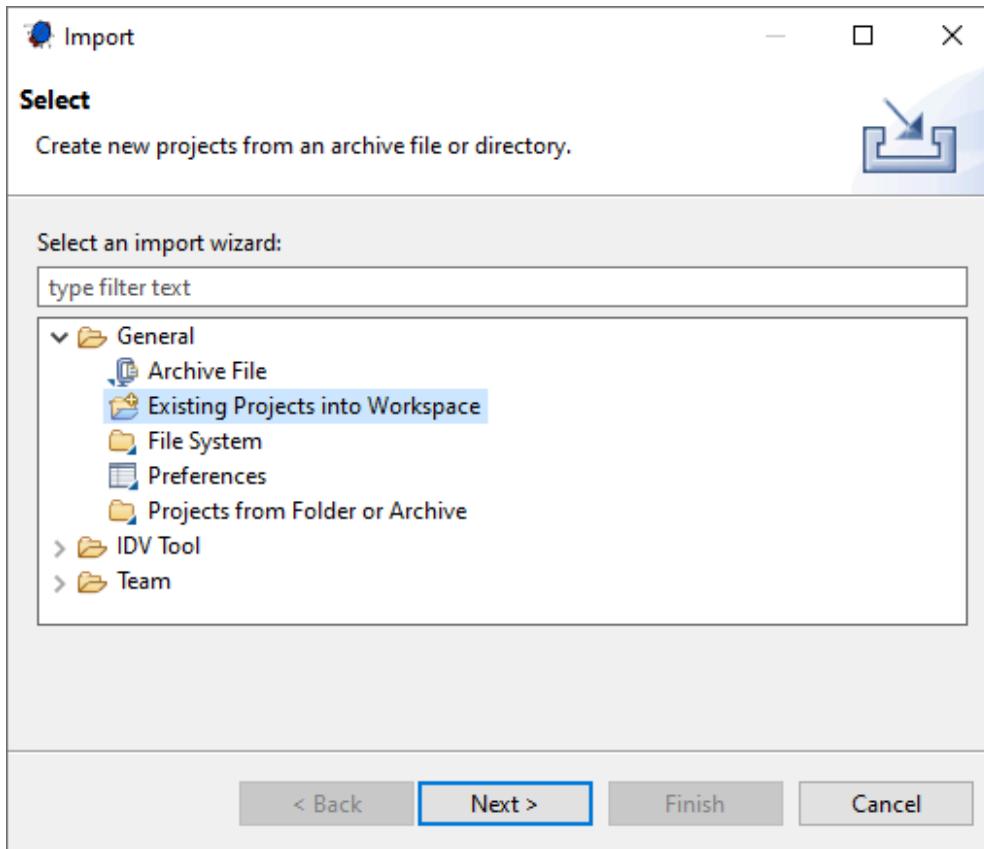
Use Existing Rule Kit

Import a Rule Kit for your target system (or an example Rule Kit for testing):

1. Make sure you have downloaded and unzipped the Rule Kit as described in the [Before You Begin](#) section.
2. On the top-left of Eclipse* GUI, click **File > Import**.

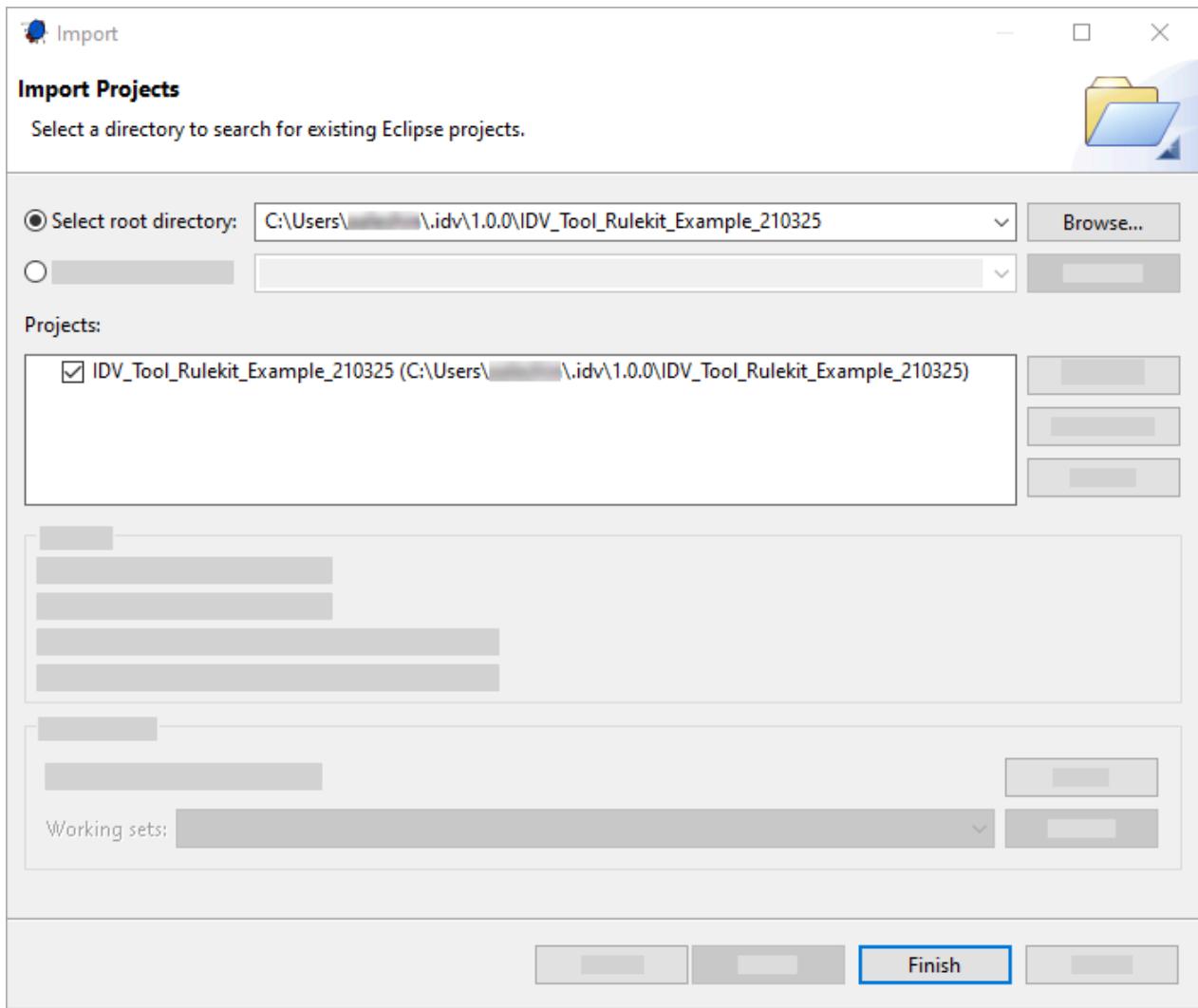


3. In the opened dialog, expand the **General** node and select **Existing Projects into Workspace**.



Click **Next**.

4. Choose **Select root directory** radio button, click **Browse**, and navigate to the *folder* that contains your Rule Kit.
5. Make sure the proper Rule Kit folder is selected in the **Projects** window and click **Finish**.

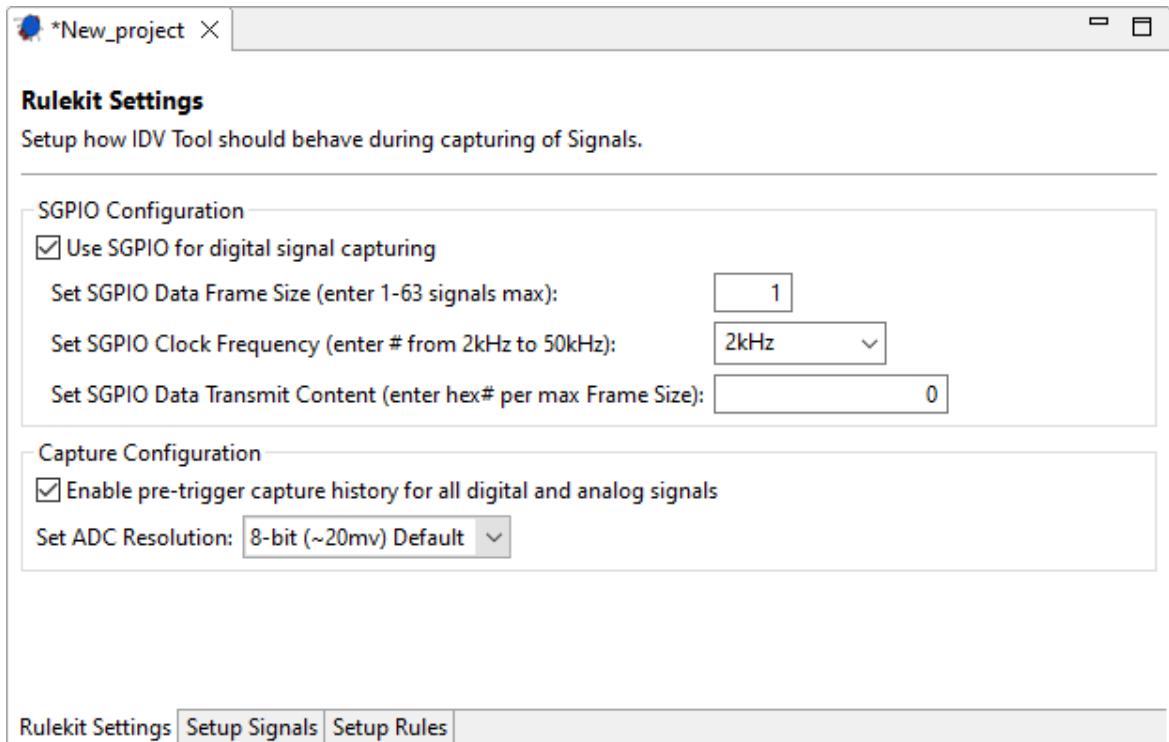


Now you can proceed with capturing and analyzing signals.

Capture Signals

1. Configure IDV Tool for signal capturing. In the project view, go to the **Rulekit Settings** tab (opened by default) and configure the following:

- Check **Use SGPIO for digital signal capturing** and, if needed, change default parameters.
- (Optional) Check **Enable pre-trigger capture history for all digital and analog signals**.
- (Optional) Configure ADC Resolution. Lower values help reduce the noise on the analog channels.



Press Ctrl+S to save the changes.

2. In the main toolbar, specify the capture time and click to start capturing signals.



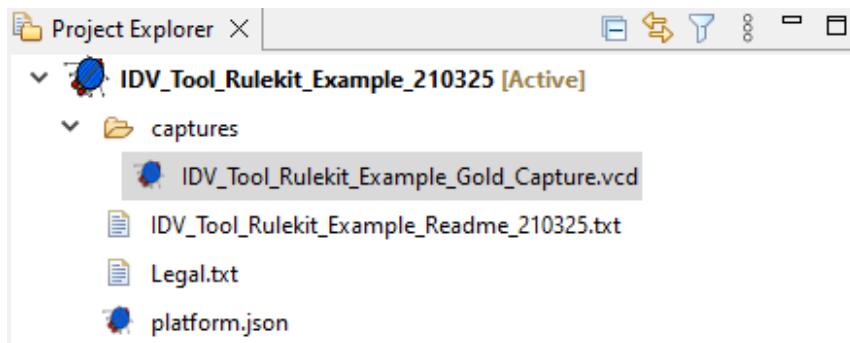
3. Power on the target system.

Upon the first signal change, the IDV Tool triggers and starts collecting all digital logic and analog voltage changes.

After the power-on cycle completes, the tool automatically analyzes all captures against specs with all pass/fail results instantly reported to the console.

4. Once the capture is ready, **Timeline Viewer** opens.

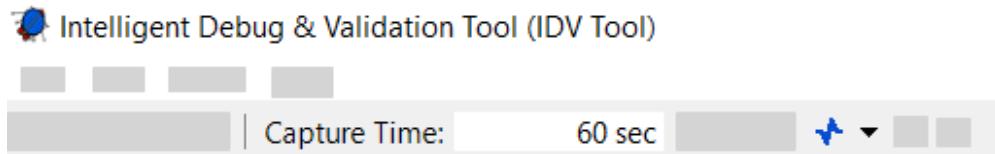
If the view is not opened automatically, go to **Project Explorer**, expand **captures** folder under your project instance, and double-click a **.vcd** file.



Now you can proceed with analyzing captured signals.

Analyze Captured Signals

1. In the main toolbar, click to analyze signal capture.



2. Check analysis results in the **Console** view.

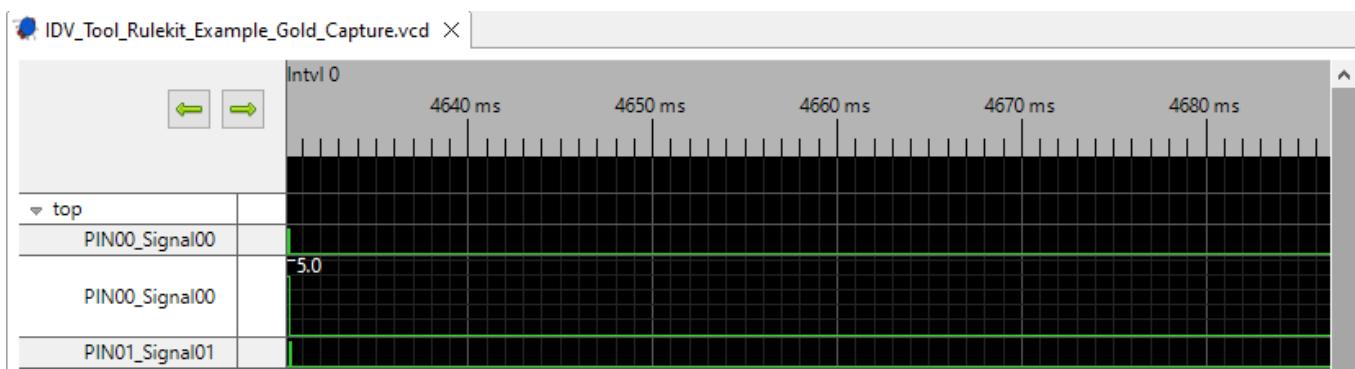
Example of analysis results of a successful capture:

```
Rule checking finished for 441 rules: 183 rules passed, 258 rules were disabled and 0 rules failed!  
Expanded Summary:  
-----  
Total Samples Captured:  
-----  
Digital Data: (Total 33 Samples)  
Analog SPI0 Data: (Total 47 Samples)  
Analog SPI1 Data: (Total 48 Samples)  
SGPIO Data: (Total 128 Samples)  
  
Total Rules Checked 183, NO Failures!
```

For deeper analysis, create rules to analyze specific types of events and work with **Timeline Viewer**.

Use Timeline Viewer

Once the capture is ready for analysis, a **Timeline Viewer** is opened. Alternatively, you can open it from the **Project Explorer**: expand the **captures** subfolder under your project node and double-click a **.vcd** file.



Use the mouse wheel to zoom in or out.

To pan the timeline horizontally, use the horizontal scroll bar or press the SHIFT key and scroll. To pan the timeline vertically, use the vertical scroll bar or press the CTRL key and scroll.

Dragging the mouse cursor anywhere in the timeline area activates the quick measurement tool that displays the time difference between two points on the time axis. To remove the quick measurement, click any place within the timeline area again.

To open the context menu, right-click the **Timeline Viewer** with the following options:

Scale To Fit

Scale the resolution to fit the whole timeline range within the visible timeline area.

Show/Hide Value

Show the value of each lane, if applicable, at the timeline marker.

Show/Hide Details

Show the details box with timestamp details when the mouse is hovered over a message in a lane. The Details box is shown on the right.

Suspend/Resume Details

Freeze the Details box. **Resume Details** to continue updating details box with current mouse position.

Show/Hide Grid

Show or hide grid lines.

Change Theme

Choose between difference themes. The main difference is the background of the timeline area having a default or darker color.

Change Lane Color: Open a dialog box to modify the color of a lane.

Set Alias

Open a dialog box to modify the displayed name of a lane.

 **Hide Lane:** Hide or show lanes.

 **Reload**

Recompute the timeline data.

 **Search:** Search for a certain parameter in the selected lane(s).

 **Go To:** Show timeline marker at the specified timestamp, next trigger, beginning or end of interval:

- **Start:** Go to the beginning of the interval until it reaches the beginning of timeline.
- **End:** Go to the end of the interval until it reaches the end of timeline.
- **Timestamp:** Open a graphical interface to go to the specified timestamp.
- **Trigger:** Go to the next available trigger.
- **Mouse Cursor:** Go to the current position of the mouse cursor.

 **Measure Time Difference**

Open a graphical interface dialog box to measure the time difference between two points of interest, including: timestamp, trigger, beginning or end of interval.

 **Annotation**

Annotate a selected axis or lane for further reference.

 **Enum Preferences**

Open a dialog to edit enumeration of selected signal(s).

 **Pan Right**

Pan to the end of the currently visible range.

 **Pan Left**

Pan to the beginning of the currently visible range.

 **Zoom In:** Zoom in the timeline. You can also use the mouse wheel.

 **Zoom Out:** Zoom out the timeline. You can also use the mouse wheel.

 **Zoom Into:** Zoom into the selected range shown within the quick measurement tool.

Create Rules

To work with rules, go to **Project Explorer**, double-click **platform.json** instance under your project node, and switch to the **Setup Rules** tab.

ID (Alias)	Rule Type	Notes	Signal 1	Transition 1
DHO10	Hard Order	-	PIN11_Signal10	FALL
DHO11	Hard Order	-	PIN12_Signal11	FALL
DHO12	Hard Order	-	PIN13_Signal12	FALL
DHO13	Hard Order	-	PIN14_Signal13	FALL
DHO14	Hard Order	-	PIN16_Signal14	FALL
DHO15	Hard Order	-	PIN17_Signal15	FALL

Rulekit Settings | Setup Signals | **Setup Rules**

To create a new rule, click Add Rule and select a rule type.

Editing and Rearranging Rules

When you create a new rule, you can interact with active fields to configure the rule appropriately.

By default, rules are ranged by ID (first column) and grouped by type (second column) but you can customize the layout as follows:

- Drag and drop a rule to a new location within the given range.
- Create a new rule group: press and hold Crtl, select rules, click Group, and enter a name for the new group. Be default, the group appears at the bottom of the table.
- Do basic operations like enable/disable, copy/paste, or remove rules.

To save all edits, press Ctrl+S.

Rule Types

Digital Sequence Order (DSQ#) Rule

Signals 1 and 2 must sequence to the defined states within a specific time (minimum/maximum time difference) with signal 1 preceding signal 2.

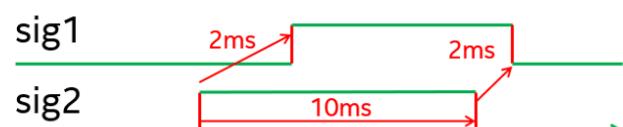
Example

ID	Signal 1	Transition 1	Signal 2	Transition 2	Min Delay	Max Delay	
DSQ00	Signal01	RISE	Signal02	RISE	1ms	2ms	
DSQ01	Signal02	RISE	Signal02	FALL	3ms	4ms	
DSQ02	Signal02	FALL	Signal01	FALL	5ms	6ms	

Passing Rule(s)



Failing Rule(s)



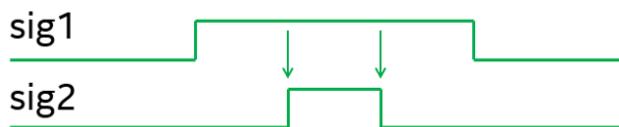
Digital State Order (DST#) Rule

A digital signal must be in a given state when another digital signal changes into a specific state.

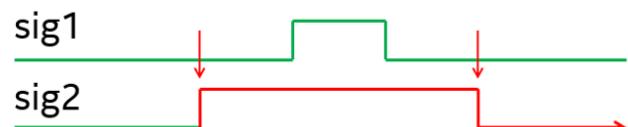
Example

ID	Signal 1	State 1	Signal 2	Transition 2	
DST00	Signal01	HIGH	Signal02	RISE	
DST01	Signal01	HIGH	Signal02	FALL	

Passing Rule(s)



Failing Rule(s)



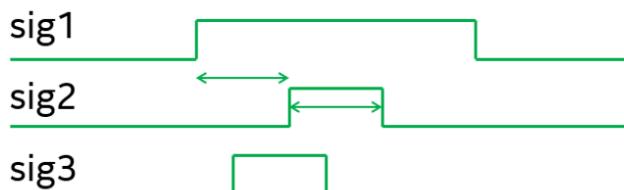
Digital Hard Order (DHO#) Rule

The given digital signals are expected to transition in the given order and no other digital signal is allowed to transition in between these signals, unless it is specified as an exempt signal.

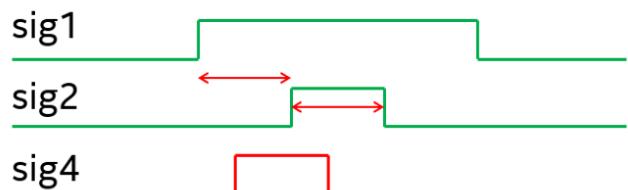
Example

ID	Signal 1	Transition 1	Signal 2	Transition 2	Exception Signal	
DHO00	Signal01	RISE	Signal02	RISE	Signal03	
DHO01	Signal02	RISE	Signal02	FALL	Signal03	

Passing Rule(s)



Failing Rule(s)



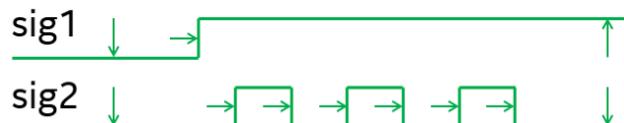
Digital Transition Count (DTC#) Rule

A digital signal is expected to transition a certain number of times. The allowed number of transitions can be a range of values. If specified, the initial and resulting states of the signal are also compared to the expected states.

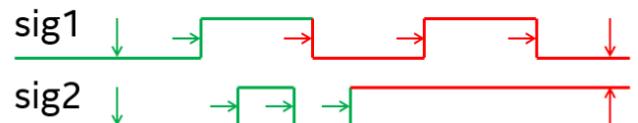
Example

ID	Signal	First State	Final State	Min Count	Max Count	
DTC00	Signal01	LOW	HIGH	1	1	
DTC01	Signal02	LOW	LOW	2	IGNORE	

Passing Rule(s)



Failing Rule(s)



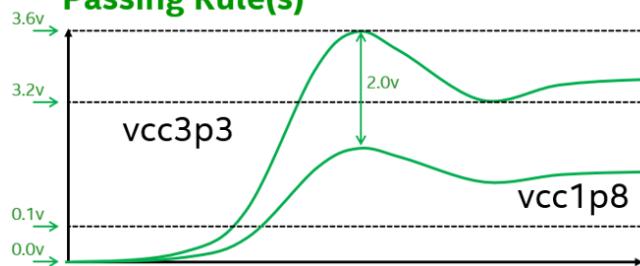
Analog Voltage Difference (AVD#) Rule

Two analog signals are expected to not exceed a given difference in voltage levels at any time.

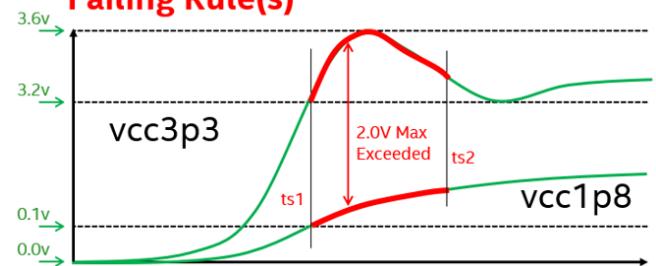
Example

ID	First Signal	Second Signal	Max Difference	
AVD00	Vcc3p3	Vcc1p8	2.0V	

Passing Rule(s)



Failing Rule(s)



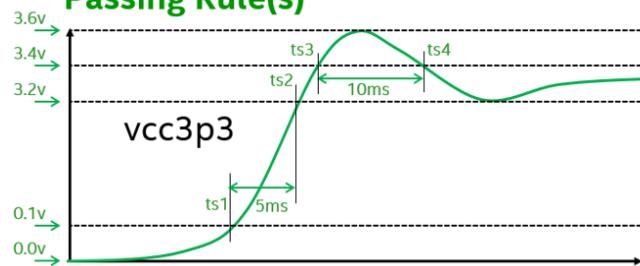
Analog Voltage Level (AVL#) Rule

An analog signal is expected to keep its low and high voltage levels within a given range and the duration of a transition under a certain amount of time. Also, the overshoot duration as well as voltage level is checked to not exceed given values.

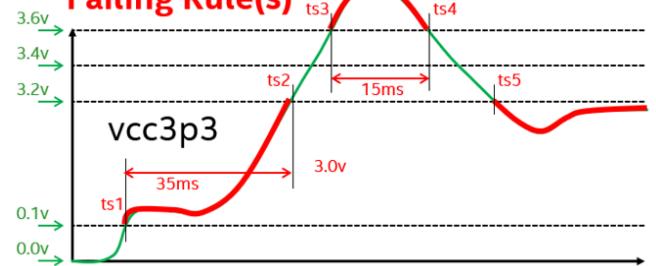
Example

ID	Signal	MinV Low	MaxV Low	MaxT Transition	MinV High	MaxV High	MaxT Overshoot	MaxV Overshoot	
AVL00	Vcc3p3	0V	0.1V	5ms	3.2V	3.4V	10ms	3.6V	

Passing Rule(s)



Failing Rule(s)



Command Line Interface (CLI) Reference

This chapter contains API reference for Intel(R) System Debugger components.

Intel(R) System Debugger - System Debug CLI User Guide

Share Your Feedback With Us

If you have any comments, suggestions, feedback, complaints or feature requests let us know at intelsystemstudio@intel.com.

Contents

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

(C) Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Quick Start (TL;DR)

The Intel(R) System Debug CLI is a Python*-based, scriptable command-line interface to the Intel(R) System Debugger - System Debug.

Launching

Launch `isd_shell` from the Intel(R) System Debugger installation folder.

Then, you can use the `intel_sysdbg` entry point to launch the program.

```
$ intel_sysdbg --help
```

For example, launch an Intel(R) System Debug instance and connect the Intel(R) System Debug CLI to it automatically by running the following command:

```
$ intel_sysdbg --launch_and_connect
```

If a TCA profile is available in the workspace, it will be used by default.

Alternatively, launch `isd_cli`. The Intel(R) System Debug CLI module is imported as “sysdbg”. For example, the previous command can be executed as follows:

```
>>> sysdbg.launch_and_connect()
```

Scripting Mode

Launch or source `isd_env` from the Intel(R) System Debugger installation folder and import the package in a Python* console. For example, connect the Intel(R) System Debug CLI to an Intel(R) System Debug instance running on local port 1534 with the following commands:

```
>>> import intel.sysdbg
>>> session = intel.sysdbg.connect(host="localhost", port="1534")
>>> session.status()
>>> session.halt()
```

In order to launch an Intel(R) System Debug instance and connect the Intel(R) System Debug CLI to it automatically, run the following commands:

```
>>> import intel.sysdbg
>>> session = intel.sysdbg.launch_and_connect()
>>> session.status()
>>> session.halt()
```

Connection Handling

Connection handling is completely managed by Target Connection Assistant (TCA). The most recently used TCA profile is stored in the workspace and automatically loaded on launch. To change the TCA profile, use the following commands:

```
>>> from intel import tcacli as tca
>>> tca.select_profile()
```

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

(C) Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel(R) System Debugger - TraceCLI User Guide

Share Your Feedback With Us

If you have any comments, suggestions, feedback, complaints or feature requests let us know at intelsystemstudio@intel.com.

Contents

TraceAPI

intel_systrace

TraceCLI provides the command line tool **intel_systrace**

TraceCLI Settings

Settings can be used to change the behaviour of TraceCLI. Settings can be used on the command line (`intel_systrace`) and from the Python* API (scripting). For both interaction models (command line, scripting) the settings can either be saved persistent in the workspace or only used temporarily for a specific operation. If the settings are persistently stored in the workspace they will be used for future commands (overwrite default settings).

Settings always consists of two parts: <setting key> and <setting value>

<setting key>

Name of the setting

! Note

kebab-case <setting key> for command line arguments (e.g. `-output-columns`)

snake_case <setting key> for `trace.settings.<setting key>` (e.g. `trace.settings.output_columns`)

<setting value>

Value of the setting. The value can come from different sources:

1. Default value
2. Loaded from workspace
3. User defined

Priority is from top (low) to bottom (high). Sources with higher priority will overwrite the ones with lower priority.

Command Line

See `intel_systrace --help` and `intel_systrace settings --help`.

persistent

```
intel_systrace settings <setting key> <setting value>
```

```
$ intel_systrace settings output-columns catalog.xml
$ intel_systrace settings csme-catalog Summary
```

temporarily

```
intel_systrace --<setting key> <setting value> <cmd>
```

```
$ intel_systrace --output-columns="Summary" --csme-catalog catalog.xml decode ipc_trace.bin
```

Scripting

See `intel.systrace.api.TraceAPI.settings` and `intel.systrace._api.options.settings.TraceCLISetting`.

persistent

```
trace.settings[<setting key>] = [<setting value>]
```

```
>>> from intel.tracecli import TraceAPI
>>> trace = TraceAPI()
>>> trace.settings.output_columns = "Summary"
>>> trace.settings.csme_catalog = "catalog.xml"
>>> trace.decode("ipc_trace.bin")
```

temporarily

```
trace.<cmd>(<args>, <setting key>=<setting value>)
```

```
>>> from intel.tracecli import TraceAPI
>>> trace = TraceAPI()
>>> trace.decode("ipc_trace.bin", output_columns="Summary", csme_catalog="catalog.xml")
```

Custom Python* Environment

Before installing TraceCLI into a custom Python* environment, install the `intel.tcaci` and `ipccli` dependencies first. In this release, only Python* versions 3.6, 3.7 and 3.8 are supported.

```
$ python -m pip install --find-links <isd_install_dir>/etc/python-tca intel.tcaci
$ python -m pip install --find-links <isd_install_dir>/debugger/ipccli ipccli
$ python -m pip install --find-links <isd_install_dir>/system_trace/python intel.systrace
```

Set the `IPC_PATH` environment variable as follows:

- `IPC_PATH = <isd_install_dir>/tools/OpenIPC_<xx>/Bin`

The following environment variables are optional. You can set them, pass as arguments to the `TraceAPI` constructor, or pass as command line arguments when launching `intel_systrace`.

- `NPK_ROOT = <isd_install_dir>/system_trace`

- `TCA_INSTALL_DIR = <isd_install_dir>/tca`

EarlyBoot

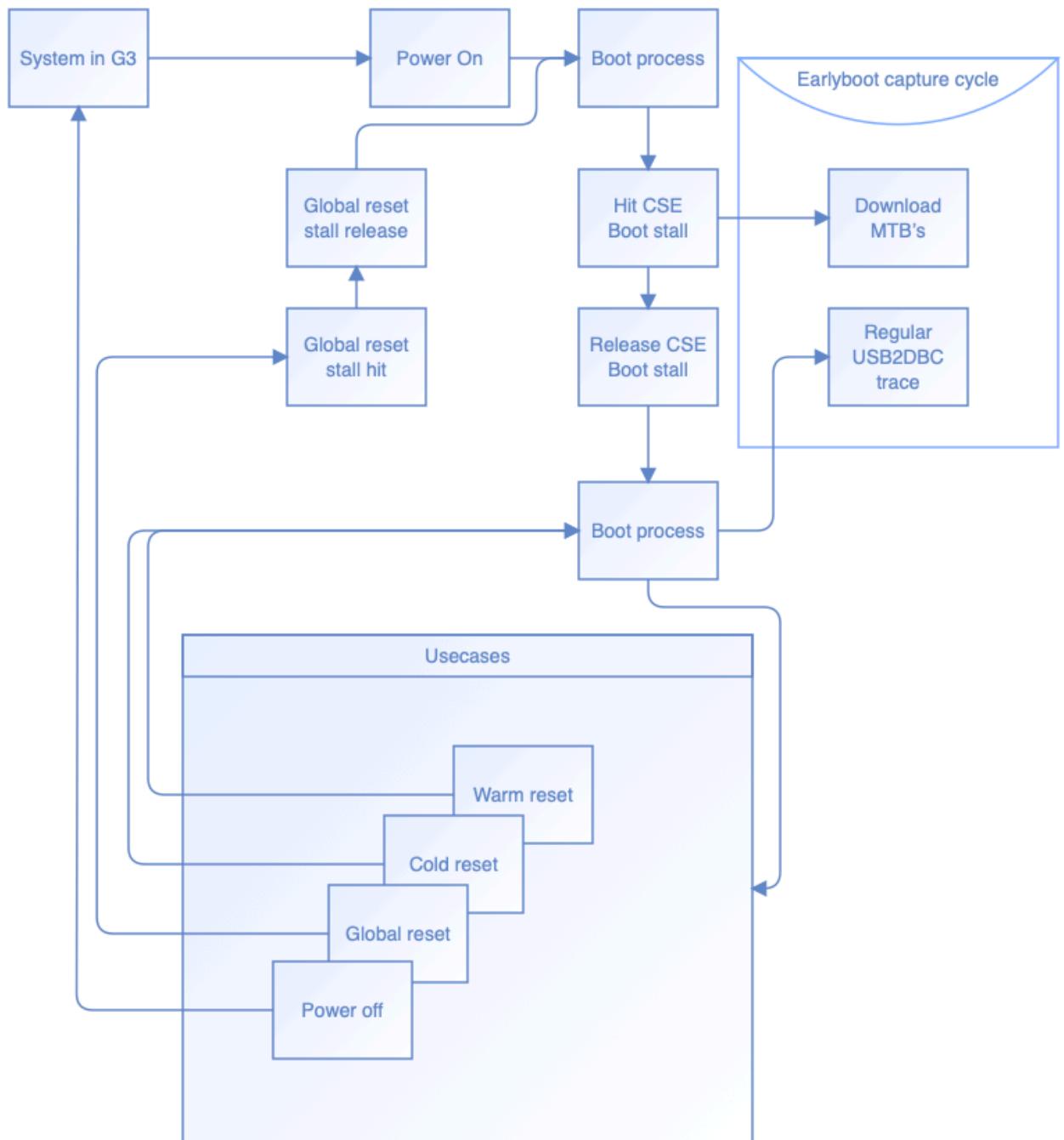
Purpose

This document provides information on how to collect early boot traces over a USB2 Dbc connection.

Normally you would not get trace messages before the USB enumeration. The solution to this problem is to store the messages in the Intel(R) Trace Hub (Intel(R) TH) internal buffer and to extract them from there. After the enumeration, normal USB2 Dbc streaming can be used.

The earlyboot capture can be started directly from G3, thrown into multiple resets (global/cold/warm) and end up in G3 or while target is running. As a result a merged capture containing all G3 and reset cycles is provided.

Typical workflow of earlyboot is shown on the following diagram:



Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

(C) Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Quick Start (TL;DR)

TraceCLI is a Python*-based, scriptable command-line interface to the Intel(R) System Trace Tool chain. All files that are created by TraceCLI, such as configuration and capture files, are stored in a workspace directory. By default, it is located in your home directory `$HOME/isd_workspace` resp.

`%USERPROFILE%\isd_workspace`.

Interactive Mode

Launch `iss_shell` from the Intel(R) System Debugger installation folder or see [here](#) to setup TraceCLI in a custom Python* environment.

```
$ intel_systrace --help
$ intel_systrace console
```

After launching the TraceCLI console mode, a `TraceAPI` instance `trace` is automatically created. To initialize a target connection and set up tracing, run the following command:

```
>>> trace.interactive_setup()
```

Alternatively, you can run the above commands from `isd_cli`.

Scripting Mode

Launch or source `iss_env` from the Intel(R) System Debugger installation folder or see [here](#) to setup TraceCLI in a custom Python* environment.

Work with the TraceAPI object in Python*:

```
>>> from intel.systrace.api import TraceAPI  
>>> trace = TraceAPI()
```

Work with TraceCLI in command line scripts

```
$ intel_systrace --help  
$ intel_systrace decode --help  
$ intel_systrace capture --help
```

Using the TraceAPI Object

Capture streaming trace from a connected target system:

```
>>> result = trace.start_capture()  
>>> print(result.status)  
>>> # check result object for errors then run the desired workload on the target system  
>>> result = trace.stop_capture()
```

```
>>> print(f"decoded data stored in '{result.output_file}'")
```

Work with an existing capture file on disk:

```
>>> result = trace.decode('/path/to/capture_file.bin')  
>>> print(f"decoded data stored in '{result.output_file}'")
```

Connection Handling

Connection handling is completely managed by Target Connection Assistant (TCA). The most recently used TCA profile is stored in the workspace and automatically loaded on launch. To change the TCA profile, use the following commands:

```
>>> from intel import tcacli as tca  
>>> tca.select_profile()
```

When using the TraceCLI interactive mode, a top-level `tca` object is automatically created and available.

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

(C) Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel(R) System Debugger - System Trace - Target Agent User Guide

Share Your Feedback With Us

If you have any comments, suggestions, feedback, complaints or feature requests let us know at intelsystemstudio@intel.com.

Contents

Quick Start

Installation

CLI and API for the Intel(R) System Debugger - System Trace - Target Agent, is a Python* package (version 3.6 or higher) bundled with the installation as a whl file. You can install it with `pip` as follows:

```
pip install intel.systrace.agent -f <nPk-agent-root>/bin/cli
```

Usage

Start the Agent with CLI

After the CLI is installed, you can launch it with the `intel_trace_agent_cli` command. By default, the CLI will try to connect to an agent on localhost or start a new one if none is running. When the CLI has connected to the agent, it opens a Python shell.

If you want to change the default settings, the additional arguments:

```
intel_trace_agent_cli [ip] [agent_port] [ssh_port]
```

where:

- `ip` - IP address of the desired host
- `agent_port` - the port that the agent listens to on localhost
- `ssh_port` - the port the CLI uses to create an SSH tunnel

The CLI tries to create an SSH tunnel to `ip` using port `ssh_port`. When the tunnel is established, the CLI attempts to connect to an agent listening to `agent_port` on localhost.

Use the `-h` flag to list all available options and arguments.

Launch the Agent without CLI

If the CLI is not installed, the `startAgent` script launches an agent with default arguments (localhost).

You can also start the agent by running the script from the command line if you need to set specific options:

```
<nPk-agent-root>/bin/intel_trace_agent
```

- To specify alternative parameters, use `-s` flag and provide `protocol:address:port` combination.
- To enable logging in the console, add `-L-`.

Use the `-h` flag to list all available options and arguments.

Python* API

The Python API is bundled with the same whl file as the CLI. For the full documentation of the API, see <<nplk-agent-root>/cli/doc>.

Trace Agent CLI

Command line tool for interacting with the trace agent, also bundled in the install.

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

IPC CLI Documentation

From a command prompt:

```
C:\> ipccli
```

From a python prompt:

```
In [1]: import ipccli
In [2]: ipc = ipccli.baseaccess()
```

All commands will be found under the ipc object.

DAL-CLI to IPC-CLI Migration

Overriding itp and itpii

The ipccli will be somewhat backwards compatible with the itp. It is recommended that you change “import itp” to “import ipccli”, but this may not always be feasible. In order for the “ipccli” to takeover the itp module, you can run:

```
ipccli.override_itpii()
```

And scripts importing the itp module will now get the ipccli instead.

Below is some assistance with some of the not-supported commands to show how to get the equivalent functionality in the IPC-CLI.

Python Equivalent Commands

- [itp.printf](#)
- [itp.sleep](#)
- [itp.pause](#)
- [itp.exit](#)

IPC-CLI Equivalent Commands

- [itp.irsScan / itp.drScan](#)
- [itp.threads\[#\].isValid](#)
- [itp.threads\[#\].isEnabled](#)
- [BitData not defined](#)
- [Event Subscription](#)
- [Logging](#)

itp.printf

This command provided C-style printf function in Python.

Replacement Use native python print statement. Example:

```
>>> integer = 10
>>> string = "Hello World"
>>> print("%d %s %x" % (integer, string, integer))
10 Hello world 0xa
```

Note that print will by default add a newline. If that is not desired, then you can either add a "" after the print:

```
>>> for i in range(10):
...     print(i, end="")
...
0 1 2 3 4 5 6 7 8 9 10
```

Or use sys.stdout directly:

```
>>> import sys
>>> for i in range(10):
...     sys.stdout.write(i)
...
012345678910
```

itp.sleep

This command provided a timeout/pause for a specified number of seconds.

Replacement

use the *time.sleep* command that does the same thing:

```
>>> import time  
>>> time.sleep(2)
```

itp.pause

This command printed a message to the screen and waited for user input followed by an Enter key press (i.e. unlimited pause until user explicit action). The command returned the string typed by the user before Enter key press.

Replacement

Use native python raw_input command:

```
>>> raw_input("Enter selection and press Enter:")  
>>> Enter selection and press Enter:
```

itp.exit

This command exited the shell with a specified exit status.

Replacement

Use python sys module exit command:

```
>>> import sys  
>>> sys.exit(status)
```

Note

this will exit the current interactive session if it is called from within a script that you are running on the command line.

itp.irscan / itp.drscan

The IPC considers the jtag chain itself a device that you can address. So to get the equivalent of a rawirscan (that will not do serialization), you can target the chain using the device alias of the chain (but be sure to use a lock statement):

```
>>> with ipc.device_lock():
...     ipc.irscan('JtagScanChain0,2,8')
...     print ipc.drscan('JtagScanChain0,32')
...
[32b] 0x00A84013
```

itp.threads[#].isenabled

The IPC CLI does not copy the device information to the node, but both the Intel(R) DFx Abstraction Layer and IPC CLI have the device object available in this case.

Replacement:

```
>>> itp.threads[0].device.isenabled
```

itp.threads[#].isvalid

The Intel(R) DFx Abstraction Layer had nodes that would sometimes not have an actual Intel(R) DFx Abstraction Layer object associated with them. The IPC-CLI does not have this, the nodes are all in the python layer. Therefore it does not have an *isvalid*. The equivalent code that would work in both Intel(R) DFx Abstraction Layer and IPC-CLI is:

```
>>> getattr(itp.threads[0], "isvalid", True)
True
```

BitData not defined

The Intel(R) DFx Abstraction Layer itpii added BitData to the python builtin name space so it was immediately accessible. The IPC CLI does not add anything to the Python global name space. You can still get to the BitData via the *itp* (or *ipc*) object, as well as the *itpii* or *ipclli* modules. Either of these will work:

```
>>> itp.BitData(32,0xa5)
>>> itpii.BitData(32,0xa5)
```

Event Subscription

The Intel(R) DFx Abstraction Layer event subscription subsystem had multiple functions. Each subscribe function allowed you to register a handler, which took a single argument. The OpenIPC + IPC CLI has similar capabilities but the functions for subscribing are slightly different:

- `itpii.subscribeToHardwareEvents => ipc.events.SubscribeTargetEvents`
- `itpii.subscribeToReconfigEvents => ipc.events.SubscribeReconfigurationEvents`
- `itpii.subscribeToRunControlEvents => ipc.events.SubscribeRunControlEvents`
- **`itpii.subscribeToBreakEvents => ipc.events.SubscribeRunControlEvents`**
 - this is not 100% the same, but it is the closest equivalent
- **`itpii.subscribeToErrorEvents => ipc.events.SubscribeMessageEvents`**
 - this is not 100% the same, but it is the closest equivalent

The handlers passed in to support these events take two parameters instead of one now as well. For complete documentation on how IPC CLI event subscriptions work, see the full [Events](#) documentation.

Logging

The IPC logging API is similar to the ITP one, but with some small changes.

- `itp.log == ipc.log`
- `itp.nolog == ipc.nlog`
- `itp.loggercho => ipc.logger.echo`
- `itp.loggerlevel => ipc.logger.level`

Global Commands

Below is a list of commands that are available directly off of the ipc object:

```
ipc.<command>
```

Most of these commands exist to provide legacy compatibility. There is a mix of some commands that expect the *device* as the first parameter and other commands that will iterate through all of the supported devices/nodes and call the corresponding function.

baseaccess (ipc object)

ippcli device commands

The recommended way of accessing a command is via its [node](#). However, the documentation below will almost always show a “device” parameter for all functions. When the functions are accessed via the node, the device parameter does not need to be specified.

The recommended way of calling functions:

```
ipc.threads[0].some_command(arg1)  
ipc.threads[0].msr(msroffset)
```

Due to backwards compatibility, there are some global commands that apply to all devices, and some that require the device id.

Some functions require the device command as the first parameter. They are:

```
autoscandr(), idcode(), irdrscan(), irdrscanreplace(), irdrscanrmw(), irscan(), drscan(),  
i2cscan(), holdhook(), hookstatus(), resettap(), pulsepwrgood(), resettarget(),
```

Additionally, most of the ipcccli commands that are executed on devices can be found on the ipc.cmds variable and all should have device as the first argument (though via the node is the preferred method):

```
ipc.cmds.some_command(device,arg1,arg2)  
ipc.cmds.msr(device,msroffset)
```

Below is a complete list of the commands supported (remember, device is not needed when executing the function from a node).

Command List

Device Actions

Device actions provide the ability to invoke probe or implementation-specific functionality not abstracted by IPC API. Use the following command to enumerate the available device actions:

```
ipc.device_action.show() # Shows the names and descriptions of all device actions for all devices
```

Device actions accept a single string as their argument. Use the following command to invoke a device action:

```
ipc.device_action.<device>.<action>("<arguments>")
```

Or alternatively:

```
ipc.device_action.<device>.execute("<action>", "<arguments>")
```

Device Configs

Device configuration settings (device configs) are named variables scoped to devices to allow advanced users to configure certain behaviors. Use the following commands to enumerate the available device configs:

```
ipc.config.show() # Shows the name and current value of all configs for all devices  
ipc.config.show_help() # Shows the name and description of all configs for all devices  
  
ipc.config.<device>.show() # Shows the name and current value of all configs for a specific device  
ipc.config.<device>.show_help() # Shows the name and description of all configs for a specific device
```

Device config values are strings and can be accessed/modified using the following syntax:

```
ipc.config.<device>.<config> # Returns the current value of a config as a string  
ipc.config.<device>.<config> = '<value>' # Sets the value of a config from a string
```

Device configs that have been modified since the beginning of a debug session can be enumerated using the following syntax:

```
ipc.config.show_modified() # Shows the name, current value, and initial value of all modified configs  
ipc.config.revert_modified() # Restores all modified configuration settings to their initial value
```

The specific configs available depends on IPC API configuration and the target/probe combination. Not all of the device configs listed here will be available for all configurations. If a probe is hot-plugged or new devices are discovered on re-configuration then additional device configs may become available.

Common Examples

Set the JTAG TCLK rate on the first JTAG chain:

```
ipc.config.jtagscanchain0.Jtag.TclkRate = '8000000'
```

Use JTAG for probe mode entry and detection on the first debug port:

```
ipc.config.debugport0.PlatformControl.PreqNotWired = 'true'  
ipc.config.debugport0.PlatformControl.PrdyNotWired = 'true'
```

JTAG Chain Device Configs

Name	Description
Jtag.TclkRate	In hertz, the frequency at which the JTAG pins are driven for issuing JTAG scans.

Debug Port Device Configs

Name	Description
PlatformControl.PlatformType	The type of platform the debug port is connected to; affects how
PlatformControl.PowerOffDelay	In milliseconds, the amount of time to wait after power loss before
PlatformControl.PowerOffTimeout	In milliseconds, the amount of time to wait for the target to power off
PlatformControl.PowerOnDelay	In milliseconds, the amount of time to wait after power restore before
PlatformControl.PowerOnPulseWidth	In nanoseconds, the pulse width of the pin assertion used to power on
PlatformControl.PrdyNotWired	Whether the PRDY pin is not wired. If using OpenRC then PRDY is asserted
PlatformControl.PreqNotWired	Whether the PREQ pin is not wired. If using OpenRC then PREQ is asserted
PlatformControl.ResetDelay	In milliseconds, the amount of time to wait after reset of the target
PlatformControl.ResetPulseWidth	In nanoseconds, the pulse width of the pin assertion used to reset

Core Group Device Configs

Name	Description
RunControl.HaltPrdyTimeout	In milliseconds, the maximum time to wait for a PRDY event before
RunControl.PrdyRunStatusPollingEnabled	Whether the run status of the target is periodically polled
RunControl.PrdyRunStatusPollingInterval	In milliseconds, the interval of time between each TAP status poll
RunControl.ProbeModeDetectMethod	Selects which method is used to detect probe mode entry
RunControl.ProbeModeEntryMethod	Selects which method is used to request probe mode entry
RunControl.ProbeModeExitMethod	Selects which method is used to request probe mode exit.

RunControl.RunStatusPollingInterval	In milliseconds, the minimum time to wait between polling
RunControl.StepMode	Specifies how the non-targeted threads are handled when
RunControl.ThreadHaltTimeout	In milliseconds, the maximum time to wait for threads to e
RunControl.WaitInstructionCompletionMode	Specifies how the non-targeted threads are handled when

Root Device Configs

Name	Description
Jtag.DetectDevicesAtStartupRetries	The maximum number of times to retry detecting taps on a sca
Jtag.EnableLowPowerDebug	When enabled, additional wake up scans may be performed fo
TargetManager.AdjustDevices	When enabled, additional scans are performed after the device
TargetManager.DetectDevices	When enabled, a TAP reset overscan will be performed in orde
TargetManager.UpdateTopologyOnReset	When enabled, a topology update will be triggered in response
TargetOS	This setting controls the amount of time to wait for cores to w
TimeScale	Percentage to scale wall-clock time for time-related operations

ippcli Devices and Nodes

There are two primary objects for interacting with the devices in the silicon:

- **Devices** - The devices primarily hold information about the silicon or portion of silicon it is representing.
- **Nodes** - Hold functions to apply to its corresponding device.

For each Device there is exactly one Node, and for each Node, there is exactly one device. Often it may be useful to apply a function to multiple nodes/devices at the same time, and for that you would use a *NodeGroup*.

- **Node Groups** - Holds multiple nodes for indexing or running a function on

The most common node group used is “ipc.threads”, which exists to be backwards compatible with the DAL/itpii.

Devices

The devices primarily hold information about the silicon or portion of silicon it is representing. A list of the known TAP devices can be found via:

Indx	DID	Alias	Type	Step	Idcode	P/D/	C/T	Enabled
0	0x00003000	ANN_CLTAPC0	ANN_CLTAPC	A0	0x10A84013	0/- / -/-	Yes	
1	0x00004000	ANN_SC0	ANN_SC	A0	0x0202100B	0/- / -/-	Yes	
2	0x00001000	ANN_ARC0	ARC	A0	0x200424B1	0/- / -/0	Yes	
3	0x00001001	ANN_AUDIO00	AUDIO	A0	0x0B284013	0/- / -/0	Yes	
4	0x00001002	ANN_PSH0	LMT	A0	0x18289013	0/- / -/0	Yes	
...								
73	0x00005001	SLM_MODULE1	SLM_MODULE	E0	0x4A68E013	0/- / -/-	Yes	
74	0x00002002	SLM_C2	SLM_C	E0	0x4A68E013	0/- / 0/-	Yes	
75	0x00001008	SLM_C2_T0	SLM	E0	0x4A68E013	0/- / 0/0	Yes	
76	0x00002003	SLM_C3	SLM_C	E0	0x4A68E013	0/- / 1/-	Yes	
77	0x00001009	SLM_C3_T0	SLM	E0	0x4A68E013	0/- / 1/0	Yes	
78	0x00014000	MIPI-PTI	InterfacePort			0/- / -/-	Yes	

You can also print the device to see some of the known information about that device:

```
>>> print ipc.devicelist[0]
alias      = ANN_CLTAPC0
devicetype = ANN_CLTAPC
stepping   = A0
did        = 0x00003000
idcode     = 0x10a84013
irlength   = 8
isenabled  = True
bustype    =
```

In some cases there may be additional information about the device that is available. To see all that is available on a device, use the built-in python dir command:

```
>>> print dir(ipc.devicelist[0])
```

The Device, the Device's DID, and the Device's alias are the primary methods of specifying a device when calling in to the IPC CLI functions. However, if the Node is being used (more on that in a bit), then the device does not need to be specified.

You can see from the example above that the Device's DID does not begin with 0. However, the itpii/DAL started with 0. For backwards compatibility the "Indx" can also be specified when calling in to a function but it is **not recommended**. It is *preferable* to use the DID, or even better, the Alias.

The Device List behaves as a list and can be indexed in to (see example above). There are two additional functions that may be useful to many users:

Nodes

Nodes are objects that help map the devices to the functions that those devices support. Currently there is a *NodeContainer* object that holds all the nodes and it is found here in the ipclli:

```
>>> ipc.devs.<tab>
ipc.devs.ann_arco          ipc.devs.ann_iopti0        ipc.devs.ann_pnd0
ipc.devs.ann_usb2phy0       ipc.devs.ann_iosdio0      ipc.devs.ann_pnd_dft0
ipc.devs.ann_atpg0          ipc.devs.ann_iossp0        ipc.devs.ann_pnd_mbist0
ipc.devs.ann_usb2pll0       ipc.devs.ann_iosspa0       ipc.devs.ann_pnd_tap2iosf0
ipc.devs.ann_audio0         ipc.devs.ann_vsp0
ipc.devs.ann_usb3phy0
```

Each Node will have the functions that are supported by that device:

```
>>> ipc.devs.slm_c0_t0.<tab>
ipc.devs.slm_c0_t0.asm()      ipc.devs.slm_c0_t0.cv        ipc.devs.slm_c0_t0.mem()
ipc.devs.slm_c0_t0.asmmode()   ipc.devs.slm_c0_t0.device
ipc.devs.slm_c0_t0.memblock()
ipc.devs.slm_c0_t0.br()        ipc.devs.slm_c0_t0.did
ipc.devs.slm_c0_t0.memdump()
ipc.devs.slm_c0_t0.brchange()  ipc.devs.slm_c0_t0.dport()    ipc.devs.slm_c0_t0.msr()
ipc.devs.slm_c0_t0.breaks     ipc.devs.slm_c0_t0.drscan()   ipc.devs.slm_c0_t0.name
ipc.devs.slm_c0_t0.brget()     ipc.devs.slm_c0_t0.go()      ipc.devs.slm_c0_t0.nodetype
ipc.devs.slm_c0_t0.brnew()     ipc.devs.slm_c0_t0.halt()    ipc.devs.slm_c0_t0.port()
ipc.devs.slm_c0_t0.brremove()  ipc.devs.slm_c0_t0.idcode()   ipc.devs.slm_c0_t0.state
ipc.devs.slm_c0_t0.cpuid()     ipc.devs.slm_c0_t0.invd()    ipc.devs.slm_c0_t0.step()
ipc.devs.slm_c0_t0.cpuid_eax() ipc.devs.slm_c0_t0.irdrscan()
ipc.devs.slm_c0_t0.thread_status()
```

NodeContainer

The various nodes are attributes off of something called a *NodeContainer*, but really you most likely just care that it is located here:

```
>>> ipc.devs
<ipclli.ipc_env.ipc_node_container.NodeContainer object at 0x00000000002C7ECC0>
```

The node container currently has the following functions for use by users:

Node Groups

A node group is a derivative of a list that will allow you to perform operations across all the nodes in that list. *ipc.threads* is an example of one such node group and here is an example of reading from all the MSRs on that node group:

```
>>> ipc.threads.msr(0x400)
GLM_C0_T0.msr - [64b] 0x0000000000000000
GLM_C1_T0.msr - [64b] 0x0000000000000000
GLM_C2_T0.msr - [64b] 0x0000000000000000
GLM_C3_T0.msr - [64b] 0x0000000000000000
```

The display is showing each node and the value that was received for each node. This is actually a list of values being returned though, so you can index in to that return data if you need to:

```
>>> data = ipc.threads.msr(0x400)
>>> data[0]
[64b] 0x0000000000000000
>>> data[1]
[64b] 0x0000000000000000
>>> len(data)
0x4
```

Now, suppose you need to check the value of an MSR across all the threads. The Node group helps shortcut that check by letting you do this:

```
>>> ipc.threads.msr(0x400) == 0
True
```

Keep in mind, **ALL** the nodes must have the same value for that to be true. So if a single thread reports non-zero, then the expression evaluates to False.

StatePort Object

The IPC API has a notion of building up multiple operations to build a higher level function called

Device Actions

Some nodes support device specific actions. You can find those actions via show actions:

```
>>> ipc.device_actions.show_actions()
debugport0 - LogInput - ScanEngine procedure that writes the specified input
                    to the OpenIPC log file (iff ScanEngine logger is
                    enabled to level Debug) and also returns the input
root       - LogInput - ScanEngine procedure that writes the specified input
                    to the OpenIPC log file (iff ScanEngine logger is
                    enabled to level Debug) and also returns the input
```

You can execute those actions off the same device_actions object:

```
>>> ipc.device_actions.root.LogInput("message for log file")
```

Or if you have the device-node already, the actions are there as well:

```
>>> ipc.devs.root.device_actions.LogInput("message for log file")
```

JTAG and I2C Locking

The new locker objects are recommended as a replacement for some of the itp type of control variables. Specifically, the jtag locker is recommended instead of trying to use the old "itp.cv.manualscans". The with syntax provided by Python will make sure that an unlock is always called even when there is an exception. The IPC API handles the actual lock and unlock count so that recursive lock/unlock calls are safe.

JTAG Locking

I2C Locking

IPC API Breaks and Breakpoints

IPC Breaks

Events

There are currently three types of Events represented here: Target Events, Run Control Events, and Message Events.

There are default events here for you to use, or you can create your own.

To unsubscribe from the default event, use the ignore_run_control, ignore_target, ignore_message, and ignore_all commands. These commands are capable of both turning off and on the events listed here: True turns them off, False turns them on.

Example:

```
>> ipc.events.ignore_all(True) #This turns off all the events listed here  
>> ipc.events.ignore_all(False) #This turns on all the events listed here
```

! Warning

The ignore_all command will only ignore the events listed here and not any other events the user may have registered.

To use your own event, create a function you want to use and use SubscribeMessageEvents, SubscribeRunControlEvents, and SubscribeTargetEvents to subscribe. To unsubscribe from your own event, use UnsubscribeMessageEvents, UnsubscribeRunControlEvents, and UnsubscribeTargetEvents. See below for examples and details.

The wait command is useful for waiting for a Run Control event to occur. Use the wait command following a go command to prevent the IPC CLI from accepting input until after the specified number of seconds, or a break in execution, whichever occurs first.

The wait command is required immediately after a go command in a debug procedure or imported file if the commands that follow are not to be executed until execution breaks. If there is no wait command after a go command in a debug procedure or imported file and a command tries to access the target processor, the procedure or import file execution terminates.

Please note that this is for the default run control events listed here and not any other run control events automatically.

Example:

```
>>? ipc.events.wait() # wait forever for a break event
>>? if ipc.events.wait(3): # wait 3 seconds for a break event
>>?     print("A break was encountered"); #wait returns true if break is seen
```

Target Events

Target events are those events that originate in the target, possibly due to user-interaction with the target, that may be of interest to a listener. These events include Power, Reset, and Sleep, among others. These events do not include any kind of break events, which are handled as Run Control events.

Each Target event will provide the event type (a value from the IPC_TargetEvent_Types enumeration) and the current state for that event. For example, a Power event would be identified with the enumeration label _IPC_TargetEventArgs and have the status “Off” or “On”, indicating the power is now off or the power is now on, respectively. If the power for an individual device has changed state, then that device is provided in the structure that provides the event arguments.

Use events.SubscribeTargetEvents to subscribe to Target events. Use events.UnsubscribeTargetEvents to unsubscribe from Target events.

When it returns, it returns an info structure. Items in that structure are:

- structureSize
- **eventType** - values returned will be one of:
 - Power
 - Reset
 - Sleep
 - BCLK
- **currentState**
 - Will be “On” or “Off” for power events
 - Will be “Started” or “Completed”
- debugportId
- logicalDeviceIds

Example:

```
def show_target_events(info, arg0):  
    """  
    Args:  
        info - event information  
        arg0 - unused in this example, but is required that this function have 2 parameters  
    """  
  
    if info.eventType == "Power" and info.currentState == "On":  
        print "System Powered Up"  
    elif info.eventType == "Power" and info.currentState == "Off":  
        print "System Powered Down"  
    elif info.eventType == "Reset" and info.currentState == "Started":  
        print "System is starting Reset"  
    elif info.eventType == "Reset" and info.currentState == "Completed":  
        print "System finished Resetting"
```

Then to subscribe you use:

You can subscribe to and unsubscribe from your own target events using the `SubscribeTargetEvents` and `UnsubscribeTargetEvents` functions.

```
>>> ipc.events.SubscribeTargetEvents(show_target_events, None)
```

Example:

```

>>> def my_target_event(info, unused):
...     print(info.eventType)
...
>>> ipc.events.SubscribeTargetEvents(my_target_event, None)

Power #Two Target Events called
Power
>>> ipc.events.UnsubscribeTargetEvents(my_target_event)
>>> def my_target_event(info, self):
...     print(info.currentState)
...
>>> ipc.events.SubscribeTargetEvents(my_target_event, None)

Off #Two Target Events called
On

```

Run Control Events

Run Control events are those events that occur whenever the target enters or exits probe mode. All break events, from resetbreak to breakpoints, are handled as Run Control events.

The Run Control Events are strings and are:

- Halt: Indicates a user requested a halt to occur
- Break: Indicates a break has occurred, halting one or more CPUs.
- Stop: Indicates a CPU has been put into or taken out of a stop state. A stopped CPU can no longer be accessed.
- Resume: Indicates a CPU has resumed execution.

Use events.SubscribeRunControlEvents to subscribe to Run Control events. Use events.UnsubscribeRunControlEvents to unsubscribe from Run Control events.

When it returns, it returns an info structure. Items in that structure are:

- structureSize
- deviceld
- breakAddress
- eventType
- breakpointId
- breakName
- breakType
- breakSubtype
- details

Example:

```
@staticmethod
def show_break_events(info, self):
    devobj = _ipc.deviceList.findByDID( info.deviceId )
    _log.result(IaThreadBreak(_ipc, devobj, info.breakType))
```

You can subscribe to and unsubscribe from your own target events using the `SubscribeRunControlEvents` and `UnsubscribeRunControlEvents` functions.

Example:

```
>>> def my_run_control_event(info, self):
...     print info.breakAddress
...
>>> ipc.events.SubscribeRunControlEvents(my_run_control_event,0)

933621 #Four Run Control Events called
796375
796375
796375
>>>
>>> ipc.events.UnsubscribeRunControlEvents(my_run_control_event)
>>> def my_run_control_event(info, self):
...     print info.structureSize
...
>>> ipc.events.SubscribeRunControlEvents(my_run_control_event,0)

56 #Four Run Control Events called
56
56
56
```

Message Events

Message events come from the IPC and represent warning conditions and general notes of interest. In general, error conditions in the IPC are communicated through exceptions, which are translated to error codes in the IPC API. Warnings and other notes are sent as message events that can generally be safely ignored or displayed to the user as they occur.

Messages are identified by strings, with two types currently defined:

- Warning: Indicates an operation might not have the desired effect.
- Info: Indicates a note of possible interest.

Use `events.SubscribeMessageEvents` to subscribe to Message events. Use `events.UnsubscribeMessageEvents` to unsubscribe from Message events.

When it returns, it returns an info structure. Items in that structure are:

- `structureSize`

- eventType
- message

Example:

```
@staticmethod
def show_message_events(info, self):
    print ""
    _log.result("{0} - {1}".format(info.eventType, info.message))
    print ""
```

You can subscribe to and unsubscribe from your own target events using the SubscribeMessageEvents and UnsubscribeMessageEvents functions.

Example:

```
>>> def my_message_event(info, self):
...     print info.structureSize
...
>>> ipc.events.SubscribeMessageEvents(my_Message_event,0)

56 #One Message Event called
>>>
>>> ipc.events.UnsubscribeMessageEvents(my_message_event)
>>> def my_message_event(info, self):
...     print info.Message
...
>>> ipc.events.SubscribeMessageEvents(my_message_event,0)

Successfully Called Message Event #One Message Event called
```

Reconfiguration Events

The IPC manages a logical device tree of all devices available in a target. This logical device tree is dynamic, depending on settings in the target. For example, the cores could be removed under certain circumstances or the cores could be added to the device tree.

Whenever the device tree might be changed, a reconfiguration started event is sent out. During this time, the device tree is considered to be in flux and no operations should be taken during this reconfiguration cycle. When the reconfiguration is complete, a reconfiguration complete event is sent out. The IPC API listens for this event.

The IPC API exposes the devices in the IPC device tree as individual device IDs. If the device tree changes, it is likely the device IDs from the IPC API will also likely change. This is why the IPC API listens for the reconfiguration events.

When it returns, it returns an info structure. Items in that structure are:

- eventType
- devicesHaveChanged

Example:

```
>>> config_event = threading.Event()
>>> def my_config_handler(info, self):
...     if info.eventType == "Completed":
...         print("Reconfig completed")
...         config_event.set()
...
>>> ipc.events.SubscribeReconfigurationEvents(my_config_handler, None)
>>> config_event.clear()
>>> ipc.forcereconfig()
>>> config_event.wait() # this blocks waiting for config event
>>> # all done clear our handler
>>> ipc.events.UnsubscribeReconfigurationEvents(my_config_handler)
```

System Stall Events

System Stall events are generated when a system stall is encountered after being set using the Break service.

When it returns, it returns an info structure. Items in that structure are:

- structureSize
- eventType
- stallName
- debugportId

Example:

```
>>> def my_stall_handler(info, self):
...     print("%s - %s"%(info.stallName, info.eventType))
...
>>> ipc.events.SubscribeSystemStallEvents(my_stall_handler, None)
>>> ipc.events.UnsubscribeSystemStallEvents(my_stall_handler)
```

Functions

Diagnostics

IPC CLI Diagnostics

The following functions are documented as being a part of the `DiagnosticsManager` class and are available via:

```
ipc.diagnostic.<function>
```

A common use of the diagnostic commands could be:

```
ipc.diagnostic.markExperiment()  
    *do something interesting*  
ipc.diagnostic.collect()  
>>>...Collecting Log Results  
...Logging Preset Restored to *previous preset*  
...Compiling Payload  
...Diagnostic Payload can be found at: *PATH to payload*
```

Logging

IPC CLI Debug Logging

The following functions are documented as being a part of the LoggerManager class. However, all of these functions are available via:

```
ipc.logger.<function>
```

The most frequently used debug configuration commands will be:

```
ipc.logger.level("ipc","debugall") # to turn on ALL debug information for the IPC CLI  
ipc.logger.echo(True) # to echo the output to the screen  
ipc.logger.setFile("path_to_log.log") # to send out put to the file  
ipc.logger.openipc_echo(True) # to echo the output to the Open IPC Log files  
... do something interesting  
ipc.logger.reset() # put all logging back to defaults
```

For the CLI logging with debugall you will typically see at least three log messages. CALLER, ENTER, and EXIT.

- The CALLER should show the first call outside of the ipccli that called the function in the ipccli.
- The ENTER will show the value of the parameters passed in to the ipccli function.
- The EXIT will show the code leaving the ipccli function and the value of any data that was created.

For Example:

```
>>> ipc.threads[0].irdrscan(2,32)
CALLER: File "<stdin>", line 1, in <module>
ENTER: irdrscan(device=0x10003e8L,instruction=0x02, bitCount=32, data=None, writeData=None)
EXIT : irdrscan result = [32b] 0x0A3CA013
[32b] 0x0A3CA013
```

Here is the complete list of functions available via ipc.logger.<function_name>:

OpenIPC Debug Logging

The following ipccli commands are intended to allow a user or script access to the logging capabilities implemented within the OpenIPC application. Intended for diagnostic and debug activities. These command are only available when the ipccli is running on top of OpenIPC. Which almost always the case.

In order to gain more insight into internal operations, the OpenIPC implements a logging infrastructure to enable recording of events and operations that occur. These logs may not be useful to end-users, but an OpenIPC developer(s) may ask you to turn on these logs if you encounter a problem.

OpenIPC has the concept of Presets which allows for a group of loggers to be enabled using a Preset Name. Presets include “GeneralDebug”, “All”, “Off”, and “Custom”. The names are fairly self explanatory as to what sort of logging output will be provided from each Preset type.

The “GeneralDebug” preset should cover most logging needs and should be selected by default.

The “All” preset is appropriate to set when low level driver, probe hardware or debug transport issues are suspected.

Use the “Off” preset to turn logging off.

If at first you are not sure which Preset to use, select the “GeneralDebug” Preset.

The Preset setting will be persistent across restart and reconnect. OpenIPC application performance will be impacted by logging. Be sure to turn logging “Off” when no longer needed.

The logging output file(s) will be in a .OpenIPC/Log/Session<#> directory within your user directory. The latest logs will be in the directory with the largest <#>.

By default, the logging output file(s) will be stored in an .OpenIPC directory within your OS specific home directory:

Windows:

C:Users<user>.OpenIPC

Linux:

~/.OpenIPC

macOS:

~/.OpenIPC

Here is the complete list of functions available via ipc.logger.<function_name> for OpenIPC:

`ipccli.cli_logging.openipc_echo(echoOn=None)`

Enable logging to Open IPC log files when `echoOn` is `True` (you must use `level()` to set which loggers are enabled for debug). Disables logging to Open IPC log files when `echoOn` is `False` and returns the current state if `echoOn` is `None`.

Open IPC uses a `logging.xml` file to configure its own logging. The Open IPC's `IpcInterface` logger must be set to a log level of "Info" or above for log messages from `ipccli` to be recorded. See Open IPC documentation for further details on configuring Open IPC logging.

Args: `echoOn` (boolean) - see above.

Returns: None

`ipccli.cli_logging.openipc_loadpreset(preset)`

Load a known logging preset setting that controls the level of detail that OpenIPC will log (logging verbosity).

Call the `openipc_presets()` command to get a list of available logging presets.

- The "GeneralDebug" preset should cover most logging needs and should be selected by default.
- The "All" preset is appropriate to set when low level driver, probe hw or debug transport issues are suspected.
- Use the "Off" preset to turn logging off

Args: `preset` (str) - name of an acceptable logging preset

Returns: None

`ipccli.cli_logging.openipc_flush()`

Write any pending log messages to disk. Then create a new rollback log file for further messages. This typically happens after the session closes, but this can be used to force a flush and new file.

Args: None

Returns: None

`ipccli.cli_logging.openipc_clear()`

Delete the log files in the standard logging directory (free up disk space).

Args: None

Returns: None

ipccli.cli_logging.openipc_archive()

Creates an archive zip file containing every log file in the standard logging directory and then removes them from the directory. Intended to easily package log files to hand off to another person.

Args: None

Returns: None

ipccli.cli_logging.openipc_presetnames()

Returns a list of acceptable logging preset names that can be loaded when calling openipc_loadpreset()

Args: None

Returns: list of strings

ipccli.cli_logging.openipc_writemessage(*message*)

Writes the specified user message to the OpenIPC log file. Makes it easy to place a marker in the log file to pass specific debug information or to tag the beginning and end of a section of commands of interest.

Args: message (str) - message to write to the log

Returns: None

ipccli.cli_logging.openipc_getloadedpreset()

Returns the name of the current loaded logging preset. Some preset will always be loaded.

Args: None

Returns: a preset name (str)

IPC CLI STDIO Logging

The following can be used to log all of the output that is being sent to stdout/stderr:

IPC CLI Logging - for Developers

The IPC-CLI logging is built on top of the standard Python logging module and is compatible with (and similar to) the PythonSv logging framework. The primary difference is that IPC-CLI logging is built so that all logging using its logging goes through a parent logger. The parent logger then controls the screen or file output. To get a logger that ties in to the CLI logging, a new logger is requested via:

```
>>> from ipc import ipc_logging  
>>> log = ipc_logging.getLogger("my script")
```

This will give the caller a logger that is controlled via the ipc.logger functions, such as echo, setFile, and level. The logger will show up in the ipc.logger.show() function as well.

If a script wants its own logging with its own file, then it is recommended to use the python logging module directly or to use the PythonSv logging.

How does the cli_logging work? It makes all of the logging objects child loggers such that the *ipc* logger is really named *ipccli.ipc*. The bitdata logger is really *ipccli.bitdata*. When we set *ipccli.ipc* to *debugall*, then that allows the messages to get to the *ipccli* logger. Then *ipccli* will log to the screen or file depending on any *echo()* or *setFile()* calls that have been made.

Dynamic Initialization

Below is a list of commands that are available directly off of the *ipc* object:

```
ipc.init.<functions>
```

IPC Datatypes for Users

There are a number of classes used within the IPC-CLI, but there are only a few that are recommended/needed by users of the IPC-CLI. Those classes/types will be documented here

- [BitData\(\)](#)
- [Address\(\)](#)

BitData

Address

Settings

The IPC CLI settings file is the file to change behavior in the IPC CLI. Most of these must be set before the first call to *ipccli.baseaccess*. Options need to be in all caps and free of typos.

Example:

```
>>> from ipccli import settings
>>> settings.PROMPT_DISPLAY = False
>>> import ipccli
>>> ipc = ipccli.baseaccess()
```

OR you can set using environment variable by preceding the setting with IPCCLI:

```
IPCCLI_PROMPT_DISPLAY=True
```

OUT_OF_PROCESS If OUT_OF_PROCESS is set to false, you can't use NorthPeak, cannot have another CLI or any other IPC client open.

IPC_LAUNCH_SERVER Attempt to launch a new server instead of connecting to an existing one. The port for the new server can optionally be specified in the IPC_API_SERVER variable.

IPC_API_SERVER The URI for the IPC API server to connect to (e.g. "domain.corp.com:1234", "10.2.14.13:1234", "[::1]:1234"). Assumes that the server is already started and listening on that address. This option takes precedence over OUT_OF_PROCESS.

IPC_CONFIG_FILE The configuration file for configuring the IPC API server. When set to None, the server's default configuration is used.

VERSION_CHECK checks the version for implemented functions.

IPC_PATH is the path to the OpenIPC. Typically it's C:\Intel\OpenIPC\Bin, but sometimes C:\Intel\DAL.

EVENT_DISPLAY turns on the default event displays for Target Events, Run Control Events, and Message Events.

EVENT_TIMESTAMP turns on or off whether events show a timestamp.

DISPLAY_WAIT_TIME specifies the number of seconds to wait for a run control event before displaying. This only works for the default run control event. This allows time to queue the printout for a condensed output. Default is 0.1 seconds.

PROMPT_PREFIX allows the user to specify a prefix in to the python prompt (like an IP address).

Or for command line usage:

```
usage: ipccli [-h] [--in-process] [--ipc-launch-server]
              [--ipc-api-server IPC_API_SERVER]
              [--ipc-config-file IPC_CONFIG_FILE]
              [--ipc-config-params IPC_CONFIG_PARAMS] [--ipc-path IPC_PATH]
              [--no-version-check] [--no-event-display]
              [--no-event-timestamp] [--display-wait-time DISPLAY_WAIT_TIME]
              [--event-wait-time EVENT_WAIT_TIME] [--developer]
              [--prompt-display] [--prompt-prefix PROMPT_PREFIX]
```

Launches an interactive IPC CLI session.

optional arguments:

-h, --help	show this help message and exit
--in-process	Connect to IPC API in the same process space.
--ipc-launch-server	Force launch of IPC API server (instead of attaching to an existing server). Use --ipc-api-server to specify a port for the server.
--ipc-api-server IPC_API_SERVER	URI for the IPC API server to connect to (e.g. "domain.corp.com:1234", "10.2.14.13:1234", "[::1]:1234"). Assumes that the server is already started and listening on that address. This option takes precedence over --in-process.
--ipc-config-file IPC_CONFIG_FILE	Path to the configuration file for configuring the IPC API server. When not specified, the server's default configuration is used.
--ipc-config-params IPC_CONFIG_PARAMS	Configuration parameter values to apply to the selected configuration as key/value pairs separated by commas (e.g. "TargetIpAddress=127.0.0.1,TargetPort=987").
--ipc-path IPC_PATH	Path to the IPC API implementation to connect to.
--no-version-check	Disable checking IPC API and implementation version before invoking newer APIs.
--no-event-display	Disable display of IPC API events.
--no-event-timestamp	Disable display of IPC API event timestamps.
--display-wait-time DISPLAY_WAIT_TIME	Time to wait before displaying queued events.
--event-wait-time EVENT_WAIT_TIME	Time to delay event display before checking for another event.
--developer	Enable developer mode.
--prompt-display	Enable display of current status on the prompt.
--prompt-prefix PROMPT_PREFIX	Prefix before the prompt.

Acknowledgements

The IPC-CLI release includes third party components covered by the following licenses:

- pyparsing, covered by the MIT License - [pyparsing](#)
- Sphinx, covered by the BSD 4-clause “Original” or “Old” License - [Sphinx](#)

All Commands

Categories

[JTAG](#)
[RunControl](#)
[HardwareFeatures](#)
[Breakpoints](#)
[TapNetwork](#)
[Memory](#)
[DMA](#)
[EnDebug](#)
[IO](#)
[Troubleshooting](#)

JTAG

`autoscandr()`
`idcode()`
`irdrscan()`
`irdrscanreplace()`
`irdrscanrmw()`
`irdrscanverify()`
`irscan()`
`drscan()`
`tapresetidcodes()`
`resettap()`
`idcode()`
`jtag_goto_state()`
`jtag_locker()` - Deprecated
`jtag_shift()`
`rawirdrscan()`

RunControl

`go()`
`halt()`
`ishalted()`
`isrunning()`
`status()`
`runstatusall()`
`polling()`
`crb()`
`crb64()`

```
msr()  
apicid()  
arch_register()  
asm()  
break_address()  
break_subtype()  
break_type()  
cpuid()  
cpuid_eax()  
cpuid_ebx()  
cpuid_ecx()  
cpuid_edx()  
eval()  
interestingthreads()  
invd()  
isruncontrolenabled()  
keepprobemoderedirectioncleared()  
keepprobemoderedirectionset()  
operating_modes()  
pcudata()  
processor_mode()  
readpdr()  
runcontroldisable()  
runcontrolenable()  
wbinvd()  
wrsubpir()  
step()  
step_mode()  
stepintoexception()  
thread_status()
```

Dynamic Initialization

```
finish()  
get_available_scenarios()  
select_scenarios()  
get_connected_probes()  
select_probe()  
initialize_probe()  
get_supported_tap_controllers()  
prespecify_tap_controllers()  
prespecify_device_configs()  
save_config()
```

```
load_config()
```

HardwareFeatures

```
holdhook()  
hookstatus()  
pulsehook()  
getpin()  
setpin()  
pulsepin()  
obspins()  
hookpins()  
power_status()  
pulsepwrgood()  
resettarget()  
i2c_locker() - Deprecated  
i2c_raw_read()  
i2c_raw_write()  
i2cscan()
```

Breakpoints

```
brchange()  
brdisable()  
brenable()  
brget()  
brnew()  
brremove()
```

TapNetwork

devicelist documentation - [ippcli Devices and Nodes](#)

```
forcereconfig()  
getplatformtype()  
getsupportedplatformtypes()  
setplatformtype()  
device_locker()  
specifytopology()
```

Memory

```
mem()  
memblock()
```

`memdump()``memload()``memsave()`

DMA

`dma()``dmaclear()``dmaconfigmethods()``dmaload()``dmasave()`

EnDebug

`encrypteddebugstatus()``startencrypteddebug()``stopencrypteddebug()`

IO

`interfaceport_read()``interfaceport_read_to_shared_memory()``interfaceport_window()``interfaceport_write()``interfaceport_write_to_shared_memory()``io()``port()``dport()``wport()`

Troubleshooting

`help()``version()``markExperiment()``collect()``log()``nolog()``clients()``isconnected()``reconnect()``block_transport_from_leaving()``perfreport()`

```
startremotedebugagent()
```

```
stopremotedebugagent()
```

Python* API Reference

Troubleshooting

This chapter provides you potential ways of resolving issues that you face during debugging. The unexpected behavior of the tool and unsatisfactory debugging experience can be caused by issues in the target system or the Intel® System Debugger functionality.

Search for a solution within this chapter first. If you do not find a solution for your problem, refer to the following resources:

- Platform Closed Chassis Debug User Guide. This document is useful for understanding the capabilities of Intel® Direct Connect Interface (Intel® DCI) probes, firmware and BIOS settings required for Intel DCI enabling.

Sign in to [Resource & Documentation Center](#) and search by document number [626623](#).

- Intel® System Debugger Release Notes. This document contains most up-to-date information on known issues (and recommended solutions) for a particular release.

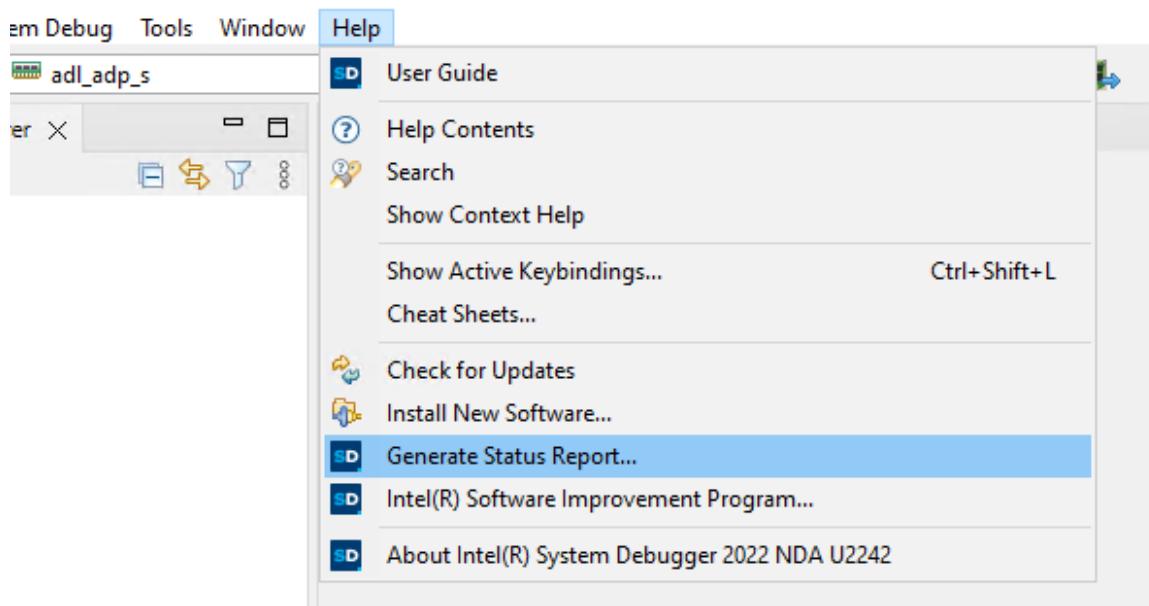
Find a corresponding file at <https://registrationcenter.intel.com>: download buttons for Release Notes are available in the Intel® System Debugger record that you used to download the latest product installer.

- Customer Support. Request assistance through [Online Service Center](#) or [Intel® Premier Support](#).

Automatic Status Report

After a connection has been created through the connection wizard, manual or automated, click on the Help / Generate Status Report... button to generate a status report.

This status report contains all log files which have been created or modified within the last two days. It also contains detailed information on the current host / debug setup.



In CLI this functionality can be executed via `tcacli`.

```
[10:44:58] [INFO] [IPC] [ADL_C7_T1] Resuming -- 10:44:58.34  
In [2]: tca.generate_report("C:\\report.zip")
```

How to Get Log Files manually

Log Files

code-block:: none

Warning - Log File Location: The location of the different log files will change in the future.

- **Installer**

- Windows: `%TMP%\intel_oneapi_installer` and `C:\IntelSWTools\logs\` (take the latest)
- Linux: `/opt/intel/oneapi/logs/installer.install.intel.oneapi.lin.sys_dbg_ndu.product*.log` (take the latest)

- **Eclipse* IDE**

- `<eclipse_workspace>\.metadata\.log` e.g., `C:\Users\<name>\system_debugger\workspace\.metadata\.log`

- **Crash Log**

- Use the `-vv` parameter to enable verbose mode for the terminal output, see **Terminal content as text** (CLI issue)

- **TCA**

- Windows: `%TMP%/tca-*.*.log` (take the latest)
- Linux: `/tmp/isd_env_<user_name>-<date>/tca-*log` (take the latest)

- System Debug

- You need to enable logging first in the launch configuration: Debugger Diagnostics > Enable Debugger Logs
- `<eclipse_workspace>/isysdbg-* .log` (take the latest) e.g. `C:\Users\<name>\system_debugger\workspace\isysdbg-20210804_170743.log`

- Python* Shell / CLIs

- Log file(s) are located at `<isd_cli_workspace>/logs/*.log`
- Logs from previous sessions are located at `<isd_cli_workspace>/logs/archive/*.zip`
- Type `isd.settings.workspace` to see the location of `<isd_cli_workspace>`
- For the GUI tool `<isd_cli_workspace>=<eclipse_workspace>/isd_cli/<tca_profile_id>` (GUI issue) e.g., `C:\Users\<name>\system_debugger\workspace\isd_cli\61404122-2c59-4025-937c-50619426ed61`
- For CLI tool, the default `<isd_cli_workspace>=<user_home_dir>/isd_workspace` (CLI issue) e.g., `C:\Users\<name>\isd_workspace`

- System Trace

- File > Export > System Trace > Trace session (GUI issue)
- `<isd_cli_workspace>/trace/<session_id>` (take the latest), needs to be zipped manually (CLI issue)
- Server logs (GUI issue)
 - Windows: `%LocalAppData%\Intel\serverLog*.txt` (take the latest) e.g., `C:\Users\<name>\AppData\Local\Intel\serverLog.2021-08-02_15.21.52.txt`
 - Linux: `~\serverLog*.txt` (take the latest)

- OpenIPC

- Right click on OpenIPC tray icon, click *Show Diagnostics*.
- Follow instructions within *Log Configuration* section to configure logging.

- Intel® Debug Extensions for WinDbg

- `%TMP%/intel_debug_logs/windbg_dci/*.log` (take the latest)
- Python terminal: copy content as text.

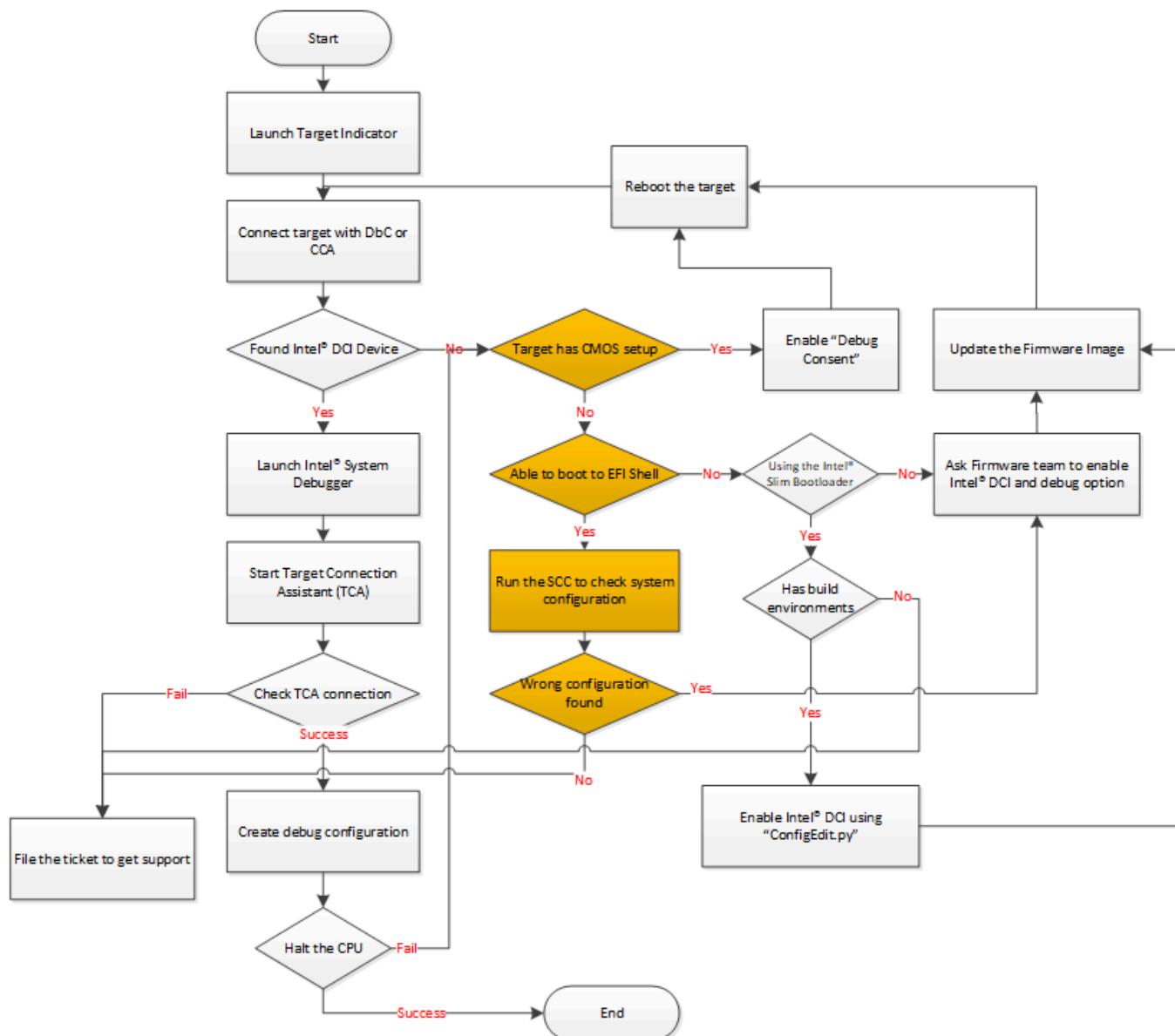
Check Debug Connection with System Configuration Checker

System Configuration Checker is a diagnostic utility that can help you resolve debug connection issues and enable debugging of an IA platform using Intel® System Debugger. The System Configuration Checker EFI application reads the debug enabling Memory I/O (MMIO) registers and Machine Specific Register (MSR) from the target hardware platform and diagnoses whether those are correctly configured.

System Configuration Checker key goal is to ensure that the target platform is ready for being debugged. For example, the tool checks if CPU run control is enabled and Intel® Direct Connect Interface (Intel® DCI) is enabled for platform debugging. The System Configuration Checker diagnoses debug-related registers and informs you whether they are configured correctly or not.

Set Up and Diagnose Debug Connection

The following diagram shows the typical debug connection setup flow in Intel® System Debugger. You can launch the System Configuration Checker EFI application when the Intel® Direct Connect Interface (Intel® DCI) device cannot be shown in device manger of Windows* OS (or the output of `lsub` in generic Linux* OS), which is illustrated by the green flow in the diagram:



Launch System Configuration Checker on the Target

UEFI BIOS usually has an option to boot to EFI shell and copy the `<install-dir>\system_debugger\<version>\samples\bin\SystemConfigurationChecker.efi` file to any thumb drive that is formatted in FAT filesystem. If the UEFI BIOS does not have EFI shell boot option, you can prepare an empty USB

thumb drive and format it with GPT partition. The following output shows an SCC example report, which contains diagnostics results and suggested ways of fixing issues:

```
> SystemConfigurationChecker.efi

System Configuration Checker 1.0
Found the matched configurations for system checking
Current Status:
  Phase-1: Platform Debug Consent checking
    Platform Debug Enable bit is set,
    Intel® DCI is enabled as USB2/3 DbC,
    then a debug connection is not established.
  Phase-2: CPU Run-Control Enabling checking
    Run-Control is enabled
  Phase-3: Intel® Processor Trace Enabling checking
    Intel® Processor Trace is disabled
  Phase-4: Crash Log configuration checking
    Three Strike is disabled
Next Steps:
  Platform is configured for system debugging using Intel® DCI,
  but Intel® DCI connection is not established. Check USB DbC or OOB connection.
```

Please share this diagnostic output with Intel when you report any issues related to debug connection.

Issues and Solutions

This section lists common (see below) and component-specific (see child pages) known issues and possible solutions. If you do not find an answer here, refer to the [Release Notes](#) for a list of short-term Known Issues and [Customer Support](#).

Target Run Control Issues

Unable to Halt the CPU

Possible issues and solutions:

Cause
CPU run control is not enabled in BIOS settings.
The Intel® Boot Guard technology is enabled. It blocks CPU run control even if the latter is enabled in BIOS

Warning Returned after Halting and Resuming the CPU

After you establish the target connection and suspend and resume threads, the following message is returned in the console:

Warning - Error occurred while determining break reason for thread...

Solution

Reconfigure the target connection by specifying that the target runs an OS. See Connecting to the Target procedure, step 7 (Platform Debug Configuration).

Unable to Identify why Threads Got Suspended

After resuming the target, threads got suspended but no breakpoint indicator is visible (all threads are simply marked as "Suspended").

Possible cause

If [thread filtering](#) is enabled, a breakpoint might be hit in a filtered-out (hidden) thread. That is why displayed thread show no indication of a breakpoint hit.

Unable to Use Buttons for Target Run Control

Cause

Buttons for target run control (like Suspend or Reset) are enabled only when a proper debug context is selected.

Solution

Refer to the [Target Run Control](#) section to identify enabling conditions for each button.

Target Simulation Issues

This chapter contains troubleshooting cases that are specific to virtual target simulation by Simics® simulator.

Cannot Connect to Agent

The following error message is displayed:

```
TCF Connect: Error Connecting to Agent TCP:localhost:1534. No connection could be made because the target machine actively refused it
```

Solution

Review the [startup instructions](#) and do the following:

- Check if Simics® simulation is running on the specified IP address/hostname.
- Check if you have enabled the Target Connection Framework (TCF) interface for the Simics simulation.

- If you are running several Simics simulation instances on one system, check if you use the right port.
- Check the Simics simulation console for any startup errors.

Cannot Connect to Stream VS0

The following error message is displayed:

```
TCF Connect: Error Connecting to Stream VS0. Invalid context
```

Cause

Simics® simulation TCF interface is enabled but no North Peak data stream available (required for tracing).

Solution

- Check if you are using a compatible version of the Simics Base package. See [Requirements](#).
- Check if you have enabled North Peak in the Simics simulation.
- Check if you have enabled the TCF interface for the Simics simulation (even so you can connect).
- Check the Simics simulation console for any startup errors.

Cannot Read Data

The following error message is displayed:

```
TCF Connect: Error Reading Data. Channel closed
```

Cause

Simics® simulation has stopped or crashed.

Solution

Check the Simics simulation console for any startup errors.

System Debug Issues

Unable to Load Symbols Using “Load the Current Module”

Pressing the  Load the Current Module (formerly named **Load This**) button does not result in locating proper symbols: the debugger cannot find the file to load.

Cause

BIOS was built without debug symbols.

Your local file system paths do not match the paths included in the debug information. This might happen if you have moved or renamed files since they were built.

The module to be loaded misses the proper page alignment. This is common for early UEFI stages.

The module size is bigger than 4KB.

Unable to Load Symbols from the .elf Binary File

Cause The `.elf` file does not contain debug symbols.

Solution First, you should analyze the `.elf` file to determine if the suggested cause is true in your case. To do analyze the file, you can use the `readelf` tool, which is a part of GNU* development utilities:

1. Check the individual file information by running the following command:

```
readelf -a <filename>
```

In the command output, check if the `.debug_str` and `.debug_info` sections are present.

1. Dump the `.debug_str` header and identify the memory address of the function you want to debug by running the following command:

```
readelf --string-dump=.debug_str <elf_file>
```

Important

Make sure the system instruction pointer refers to the same memory address as defined in the output above.

If debug symbols are indeed missing in the `.elf` file, refer to the documentation for your toolchain and find how to enable debug symbols.

Unable to Break at Reset Vector

You should check if the Integrated Firmware Image (IFWI) uses the latest firmware ingredients, especially the Power Management Controller (PMC) firmware.

Unable to Display the Variable Value

The debugger determine a variable from a particular address during the debug session. When you hover over a variable in the source code or select it in the [Variables view](#), the value is show as N/A and the following message is displayed :

```
Exception: Cannot retrieve <parameter>. The parameter is incorrect.
```

Cause

A certain form of optimisation is turned on, which has affected the debug information.

Solution

Ensure that any form of optimization is disabled and symbol files are generated properly.

Collecting Execution Trace Timed Out

If the process of [collecting execution trace](#) has not been completed within a default time range (10 seconds), the following message is displayed in the console:

```
ITrace: Processor Trace decoding timed out
```

To resolve the issue, try the following options:

- Extend the timeout. This might help slow probes to complete the capture.

Open the `<install-dir>\system_debugger\<version>\env.d\win32*-isysdbg-env.bat` file for editing and uncomment the following line:

```
set ISYSDBG_PT_TIMEOUT=<seconds>
```

where `<seconds>` must be an integer.

! Tip

Try to extend the timeout by 600 seconds. If the issue remains, check more recommendations below.

- Verify the probe used for target connection and switch to USB3 DbC if possible.
- Collect detailed logs of the trace capture process, analyze it yourself or send to Intel.

1. Configure logging:

1. Open the `<install-dir>\system_debugger\<version>\env.d\win32*-isysdbg-env.bat` file for editing, uncomment the line for setting the `ISYSDBG_LOG_LEVELS` environment variable, and edit it as follows:

```
set ISYSDBG_LOG_LEVELS=itrace,tde
```

where `itrace` and `tde` are the names of log levels required for analysis.

Save and close the file.

2. Create a new [debug configuration](#), enable log collection (step 3), and launch the debug session.
2. Try to [collect execution trace](#) again.
3. Find the saved log file in the Eclipse* workspace directory. For example, `C:\Users\<username>\system_debugger\workspace\.metadata\.log`.
4. Send the log file to Intel for further analysis or study it yourself:

- Look for the following messages:

- `ITrace: PT buffer of size=XXXX, rotation=XXXX, PSBFreq=XX read from address=XXXX`

This message reports on an active Intel® Processor Trace (Intel® PT) configuration on the target system.

Possible issue: the `PSBFreq` parameter indicates the size of a single buffer segment. If the parameter value is too high (for example, 16K), processing a segment takes long.

- `TDE: read_memory(ip=XXXX, size=XX); X page cache misses; X milliseconds or TDE: XXXX more memory reads on second X`

Possible issue: if reading the memory takes too long, the caching system is not working properly. Using the USB3 DbC probe might resolve the issue.

- Check if the debugger is scanning the proper region of interest. Try to change or reduce the area of scanning using the breakpoints. Contact Customer Support for assistance.

No Power Events Displayed in the Instruction Trace View

If you want to capture Power Event Trace and/or `PTWRITE` instruction via the [Instruction Trace view](#) and see no expected data in the view output, check the following:

1. Check if your target supports Power Event Trace and/or `PTWRITE` instruction:

1. Configure logging:

1. Open the `<install-dir>\system_debugger\<version>\env.d\win32*-isysdbg-env.bat` file for editing, uncomment the line for setting the `ISYSDBG_LOG_LEVELS` environment variable, and edit it as follows:

```
set ISYSDBG_LOG_LEVELS=itrace
```

Save and close the file.

2. Create a new **debug configuration**, enable log collection (step 3), and launch the debug session.

2. Try to **collect execution trace** again.

3. Find the saved log file in the Eclipse* workspace directory. For example, `C:\Users\<username>\system_debugger\workspace\.metadata\.log`.

Search for messages starting with `ITrace:` tag that define whether you target system supports Power Event Trace and/or `PTWRITE` instruction.

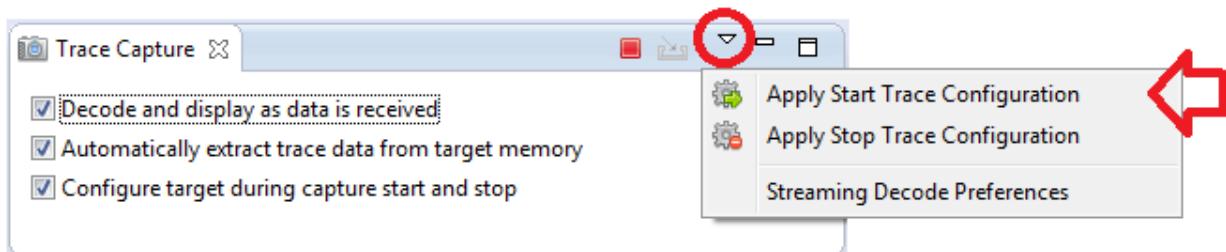
2. If you target supports these trace types, relaunch trace capturing and/or target connection.

System Trace Issues

No Trace Events Appear

Solution

- Check if the configuration has the necessary trace sources enabled.
- Check if the firmware to trace has tracing to North Peak (NPK) enabled.
- Ensure that **Configure target during capture start and stop** box in the **Trace Capture** view is checked and restart trace capture. If this option is disabled, the viewer skips any trace configuration.
- Select the **Apply Start Trace Configuration** option from the **Trace Capture** view context menu and restart trace capture:



- If you are using the target simulation, you need to determine if a problem lies with Simics® simulator or System Trace. Check the `*-npk-bin.log` (for example, `tgl-npk-bin.log`) file in the workspace folder of the Simics® simulator:

- If the size of the file is 0 bytes, the problem is most likely caused by the simulation or the setup of the Simics simulator.
- If the size of the file is larger than 0 bytes, the problem is most likely caused by the Simics simulator setup or System Trace.

Event Trace for Windows* (ETW): Error in Start Trace Capture

Solution

Make sure that the Intel® Trace Hub device is displayed in Windows Device Manager. If it is not displayed, set the configuration in BIOS to enable the Trace Hub device.

Memory Pool Configuration is not Valid for Current OS

The following message is returned in the console:

```
[ERROR] [TRACE] Memory pool configuration is not valid for current OS
```

Reason

On Linux* OS, sysfs entry (`/sys/module/usbcore/parameters/usbfs_memory_mb`) controls the maximal chunk size, which can be scheduled for USB bulk transfer by kernel. Decoder will adjust size of chunks that exceed value to maximum possible (in this case, value from sysfs) and continue execution. This might result in significant performance degradation and data losses or back pressure.

Solution

As decoder cannot adjust sysfs value without root permissions, you must have compatible OS and memory pool settings. For example, set maximal chuck size to 128Mb by executing `echo 128 | sudo tee /sys/module/usbcore/parameters/usbfs_memory_mb`.

EarlyBoot: Corrupt Message (too short)

The following message is shown in the “Summary” field after decoding:

```
Corrupt message (too short) raw:
```

Reason

EarlyBoot usecase is running through multiple target power cycles. There is no way to tell target to flush the trace buffer and though last messages up to alignment bound might be lost or corrupted.

Solution

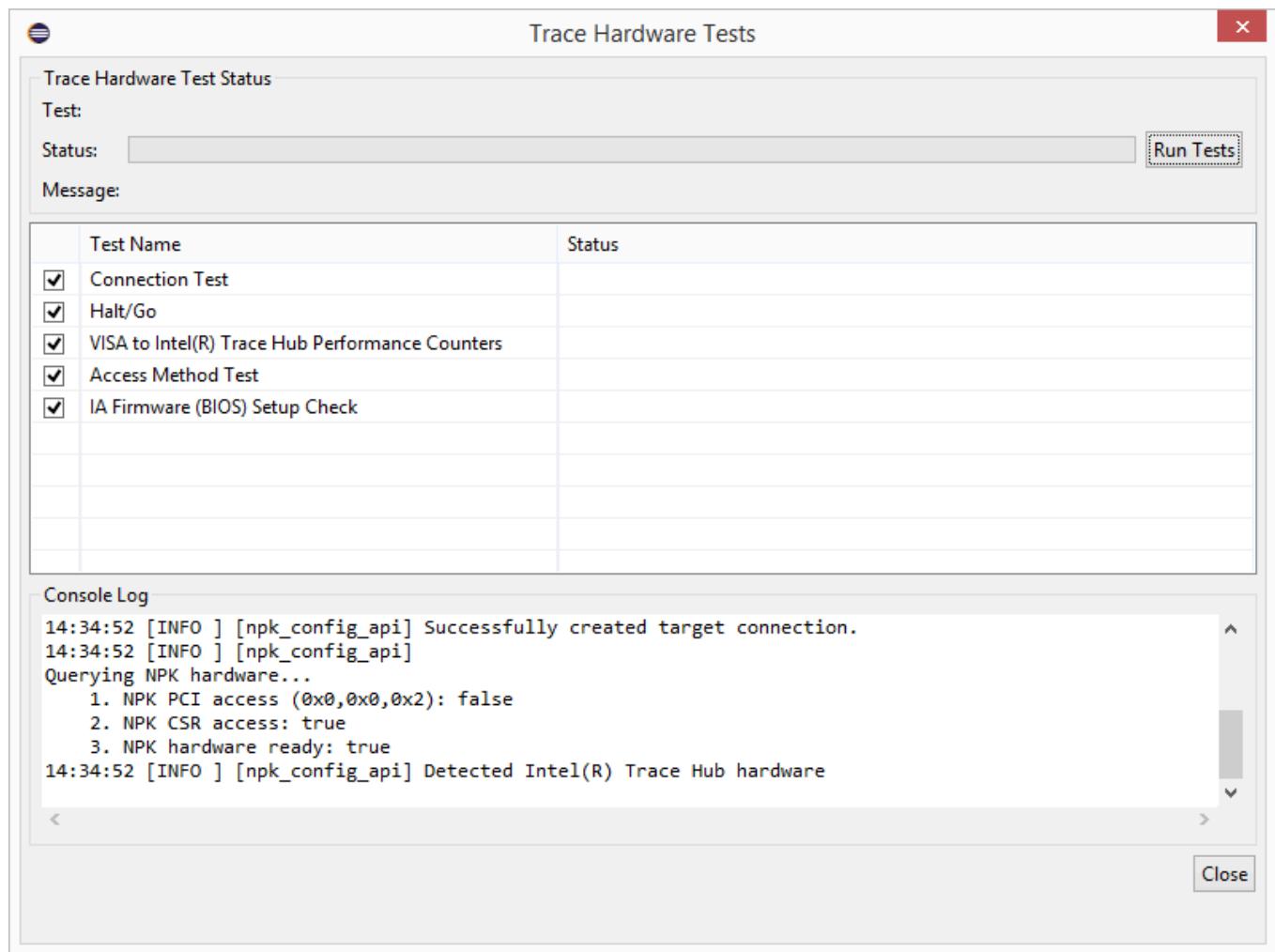
This is an EarlyBoot data collection limitation; it should not affect user experience. You can ignore this error.

Advanced: Test Your Target

Before starting trace, you can verify that basic features on the target are working correctly and the target is ready for tracing.

Once the target connection is established, open the **Trace Capture** view, click  View Menu and select **Trace Hardware Tests**.

The following dialog opens:



Select the tests you want to run from the list:

Connection Test

Checks that the connection to the probe is available.

Halt/Go

Checks that the platform can halt/go correctly. An error here indicates that some sources, e.g. AET, may not be configurable.

VISA to Intel® Trace Hub Performance Counters

This is the Intel® Trace Hub internal check that verifies the validity of the hardware VISA observation paths to Intel® Trace Hub are able to be configured and that performance counters are usable.

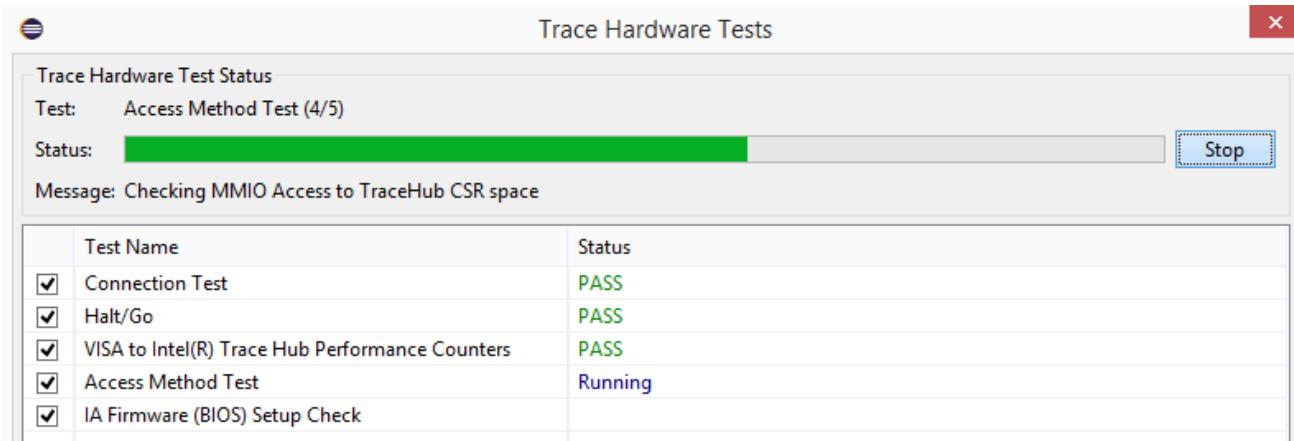
Access Method Test

Tests the various access methods that are supported. If an access method fails, it could be that some trace sources may not be configurable.

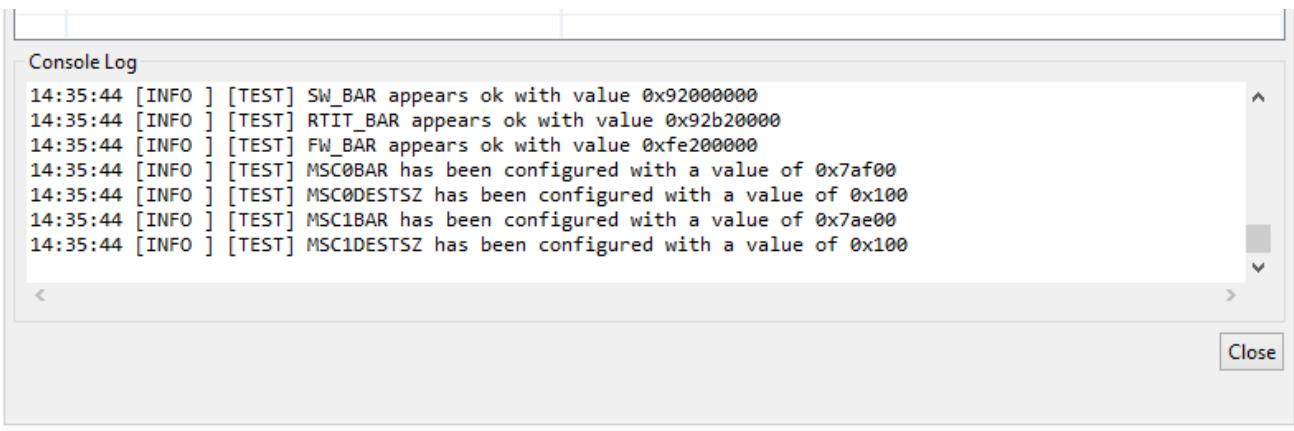
IA FW BIOS check

Check to see that BIOS configured the Intel® Trace Hub hardware correctly. If this test fails, it means that BIOS did not setup Intel® Trace Hub and the user should enable Intel® Trace Hub in the BIOS menus.

To start the tests click the **Run Tests** button on the upper-right. The status of the test run is indicated by a progress bar:



The **Status** column indicates the test results. For more descriptive status messages, check the **Console Log**:



Lost XX packets over YY (traffic exceeded Intel® Trace Hub bandwidth limit)

The following message is shown in the "Summary" field after decoding:

Lost 242 raw MIPI STP packets over 295.573ns (traffic exceeded Intel® Trace Hub bandwidth limit)

Reason

Bandwidth of collected trace data exceeds bandwidth limit supported by DCI to USB trace technology

Solution

- Try streaming with USB3 instead of USB2 if applicable
- Try to reduce amount of streaming data: for example, deselect unneeded cores for Architectural Event Traces (AET) if applicable

Intel® Debug Extensions for WinDbg Issues

If the connection to the target via the IA JTAG fails, WinDbg issues the 'Server execution failed' message.

Possible issues and solutions are:

ERROR: Could not start kernel debugging using exdi: Class not registered

Cause

The ExdiIPC.dll file was not registered correctly.

Solution

Register the .dll file as follows:

1. Go to the `<install-dir>\system_debugger\<version>\windbg-ext\iajtagserver\intel64` directory.
2. Run the `regsvr32 ExdiIPC.dll` command.

ERROR: Could not start kernel debugging using exdi: Specified module could not be found

Cause

The `ExdiIPC.dll` file was removed after the registration.

Solution

Reinstall the add-on and make sure that it is available at `<install-`

`dir>\system_debugger_<version>_nda\windbg-ext\iajtagserver\intel64\ExdiIPC.dll`

ERROR: Could not start kernel debugging using exdi: Win32 error 0x87. The parameter is incorrect

Cause

The `dllhost.exe` process froze after an abnormal termination of a debug session.

The Intel® Debug Extensions for WinDbg eXDI COM server was not registered with Windows* OS due to a

ERROR: No probe found. Please check probe connection and power

Cause

Connection between the host and the probe is not correctly established. The probe is powered off or out of order.

Solution

Check the host connection to the probe and the probe power. Also, check if the probe drivers are correctly installed and working.

ERROR: No active threads detected

Cause

- Connection between the target and the probe is not correctly established.
- The probe is powered off or out of order.

Solution

- Check the target connection to the probe and the target power.
- Reset the target if locked.

ERROR: could not import runpy module / ModuleNotFoundError: No module named 'runpy'

Cause

The system with Intel® System Debugger installed has not been cleaned.

Solution

Remove the folder `C:\Users\<username>\AppData\Local\Intel\windbg-ext-venv` and restart WinDbg.

WinDbg* is frozen

Cause

WinDbg is scanning memory to establish the connection. This can take several minutes. If the WinDbg break was issued at the wrong time, the scanning process takes about 10 minutes.

Solution

Wait until the scanning is done.

Cannot start the Target Indicator on Linux* OS

The screen is empty or the following message is returned:

```
Error: Unable to initialize GTK+, is DISPLAY set properly?
```

Solution

Launch the Target Indicator with the **-w** parameter. See an example below:

```
/opt/intel/oneapi/system_debugger/latest/target_indicator/bin$ ./TargetIndicator -w
```

Cannot Connect the Closed Chassis Adapter (CCA) to Target Using USB3

Connection attempt runs into a timeout.

Solutions

- Use a USB2 (non-SuperSpeed) Type A to Type B cable to connect the CCA to the host.
- Use the cable shipped with the Closed Chassis Adapter (CCA)

Closed Chassis Adapter (CCA) is not Recognized by Windows* OS

Cause

Intel® Direct Connect Interface (Intel® DCI) drivers are not properly installed.

Solution

In the Windows Device Manager, search for a device called *Intel USB CCA Debug Class Devices*. If it is not set up, contact customer support (see details in Intel® System Debugger Release Notes).

Connection to Target Takes Long and Fails with a Timeout Message

Cause

If OpenIPC detects an old firmware on the CCA, it updates the firmware automatically during the first connection to the target. In some cases, connection procedure may run into a timeout and display error messages on the Eclipse* console. The old firmware is only updated at the first connection. All subsequent connections should run smoothly.

Solution

If you ran into a timeout, do the following:

1. Close Eclipse* and kill OpenIPC connections (if any).
2. Unplug the CCA adapter.
3. Plug-in the CCA adapter.
4. Start Eclipse* and follow the connection procedure again.

Eclipse Console Shows an IPC API Error Message

For example:

- IPC API library is not initialized
- Error, IPC is NOT ready - failed to connect

Solution

Make sure the target is connected correctly using an Intel® Direct Connect Interface (Intel® DCI) USB 3.x Debug Class debug cable (yellow or orange connectors).

Board is not Booting Completely to UEFI Shell

Solution:

Try restarting (power off/power on) the board several times until it boots to UEFI.

Advanced: Using IPC CLI to Configure OpenIPC Connection

The default and recommended way of establishing target connection using the Target Connection Assistant (TCA) is described in the [corresponding section](#).

Note

In the default way of establishing target connection via GUI, the IPC CLI session is launched in the Shell view when Intel® System Debugger connects to OpenIPC.

However, you can use the IPC command line interface (IPC CLI) to run and configure OpenIPC sessions if:

- TCA does not detect your running OpenIPC session as expected
- You are used to working with the IPC CLI.

In any of the cases above, follow the steps below to establish target connection:

1. Launch IPC CLI:

- a. Navigate to the installation root directory and run `isd_shell.bat` (for Windows* OS host) or `isd_shell.sh` (for Linux* OS host).
- b. In the opened command line, run `ipccli` and configure the OpenIPC session as required.

For instructions on using the IPC CLI, call `help()` from the command line or refer to the IPC CLI documentation located at:

- For Windows* OS host: `<install_dir>\system_debugger\<version>\tools\python3\Lib\site-packages\ipccli\html`
- For Linux* OS host:
`<install_dir>/system_debugger/<version>/tools/python3/lib/python3.x/sitepackages/ipccli/html`

2. When your OpenIPC instance is running, launch the Intel® System Debugger. From the installation root directory, run `iss_ide_eclipse-launcher.bat` (for Windows* OS host) or `iss_ide_eclipse-launcher.sh` (for Linux* OS host) and proceed with [remaining steps](#).

Eclipse Console shows a problem with locales packet

When trying to connect to local target with IDE buttons, Eclipse launches an error window with the suggestion to install locales package

Solution: sudo locale-gen en_US en_US.UTF-8 sudo dpkg-reconfigure locales

export LC_ALL=en_US.UTF-8

Samples

This section contains instructions on using sample materials shipped with the product. Most samples require a real target system to be connected but some (the one for System Trace) can be used without it.

Work with the sample if you are a novice user of the product or one of its components and want to learn how to complete basic tasks using Intel® System Debugger functionality.

IMPORTANT NOTICE: This software is sample software. It is not designed or intended for use in any medical, life-saving or life-sustaining systems, transportation systems, nuclear systems, or for any other mission-critical application in which the failure of the system could lead to critical injury or death. The software may not be fully tested and may contain bugs or errors; it may not be intended or suitable for commercial release. No regulatory approvals for the software have been obtained, and therefore software may not be certified for use in certain countries or environments.

EFI Application

Prerequisites

- Operating system
 - Windows* 10
 - Windows* 11
 - 64-bit Linux*
- Hardware
 - Any machine which has an EFI based BIOS (UpXtreme detailed here)
- Software
 - Cmake* version 3.7 or higher
 - Microsoft Visual Studio* 2019 or higher (for Windows*)
 - Clang-9 or higher (for Linux*)

File Locations

After installing the Intel® System Debugger on the system, you can find sample sources and binaries in the following directory:

- Sample sources: `<isd-install-dir>/samples/intel`
- Sample binaries: `<isd-install-dir>/samples/bin`

Next steps

Read more about how to build the sample and write it to an USB drive

- [Build and flash to a USB drive EFI Application on Windows*](#)
- [Build and flash to a USB drive EFI Application on Linux*](#)

Build and flash to a USB drive EFI Application on Windows*

- [Install EDK II](#)
- [Build the EFI Sample Application](#)
- [Flashing the EFI Application to a USB drive](#)

Install EDK II

1. Open the Windows Command Prompt (cmd)
2. Create a workspace folder for the EDK II
3. Change to the workspace folder
4. Clone the EDK II repository and update the submodules

```
git clone https://github.com/tianocore/edk2.git  
cd edk2  
git submodule update --init
```

5. Build BaseTools

```
.\edksetup.bat Rebuild
```

Build the EFI Sample Application

1. Open the x64 Native Tools Command Prompt for your Visual Studio
2. Set the environment variables

```
set EDK2_ROOT=<edk2-workspace-dir>\edk2  
set EDK2_TOOLS=%EDK2_ROOT%\BaseTools\Bin\Win32
```

3. Create the build folder

Note

The build folder should be created in user writable space, outside of the ISD installation directory

4. Change to the build folder
5. Run the CMake generator

```
cmake <isd-install-dir>\samples
```

6. Build the application

```
cmake --build .
```

7. Find debug symbols and EFI file in `<build-dir>\Debug`

- `SystemConfigurationChecker.efi` - EFI Application sample
- `SystemConfigurationChecker.pdb` - debug symbols file

Flashing the EFI Application to a USB drive

Format the USB drive with a GPT partition table

1. Open the Windows Command Prompt (cmd) and run the `diskpart` tool

2. Enter the following commands (where `x` is the disk number of the USB drive from the output of `list disk`)

```
list disk
select disk x
clean
convert gpt
```

In the case of successful conversion, similar message should be displayed

```
DiskPart successfully converted the selected disk to GPT
```

3. Enter the `exit` command to close the DiskPart tool

Write the EFI Application to an USB drive

1. Run the script `<isd-install-dir>\samples\intel\tools\make_boot_media.ps1` with PowerShell*

If required, accept the access request for administrator privileges

2. Using the file browser that appears, select the file `SystemConfigurationChecker.efi`, located in the `<build-dir>\Debug` directory

3. Using the file browser that appears, select the prepared USB drive

Note

If you receive Execution Errors, you might have to change your powershell execution policy. This can be done with the following command in PowerShell*: `Set-ExecutionPolicy Unrestricted`

Build and flash to a USB drive EFI Application on Linux*

- [Install EDK II](#)
- [Build the EFI Application sample](#)
- [Writing the EFI Application to an USB drive](#)

Install EDK II

1. Create a workspace directory for the EDK II
2. Change to the workspace directory
3. Clone the EDK II repository and update the submodules

```
git clone https://github.com/tianocore/edk2.git
cd edk2
git submodule update --init
```

4. Build BaseTools

```
make -C BaseTools -jN  
"N" - is the number of processes to run in parallel
```

Build the EFI Application sample

1. Set the environment variables

```
cd <edk2-workspace-dir>/edk2  
source edksetup.sh
```

! Note

Environment variables can also be set manually

- `export EDK2_ROOT=<edk2-workspace-dir>/edk2`
- `export EDK2_TOOLS=${EDK2_ROOT}/BaseTools`

2. Create build directory

! Note

Build directory should be created in user writable space, outside the ISD installed directory

3. Change to the build directory

4. Run the CMake generator

```
cmake <isd-install-dir>/samples
```

5. Build the application

```
cmake --build .
```

6. Find debug symbols and EFI file in `build` directory

- `SystemConfigurationChecker.efi` - EFI Application sample
- `SystemConfigurationChecker.so` - ELF file that also contains the debug symbols

Writing the EFI Application to an USB drive

To write the EFI Application to an USB drive `make_boot_media.sh` script is used. This script is located in `<isd-install-dir>/samples/intel/tools`

1. Make the script executable

```
chmod +x <isd-install-dir>/samples/intel/tools/make_boot_media.sh
```

2. Run script to flash the EFI Application to the USB drive

```
<isd-install-dir>/samples/intel/tools/make_boot_media.sh <build-dir>/SystemConfigurationChecker.efi /dev/sdb
```

Note

/dev/sdb - is the name of USB drive

System Debug Usage Sample

This project demonstrates how you would use the Intel® System Debugger for system debug to find a bug in an example EFI application.

Creating a Bug

Let's make a simple "bug". This, of course, is not really a bug - but will be a write to an IO port, that we will use the debugger to "catch".

You can do this by using the EDI application created in a [corresponding sample](#).

To write to this IO port - you need to build the EFI app with the Cmake option `BAD=ON`.

```
cmake .. -DBAD=ON
```

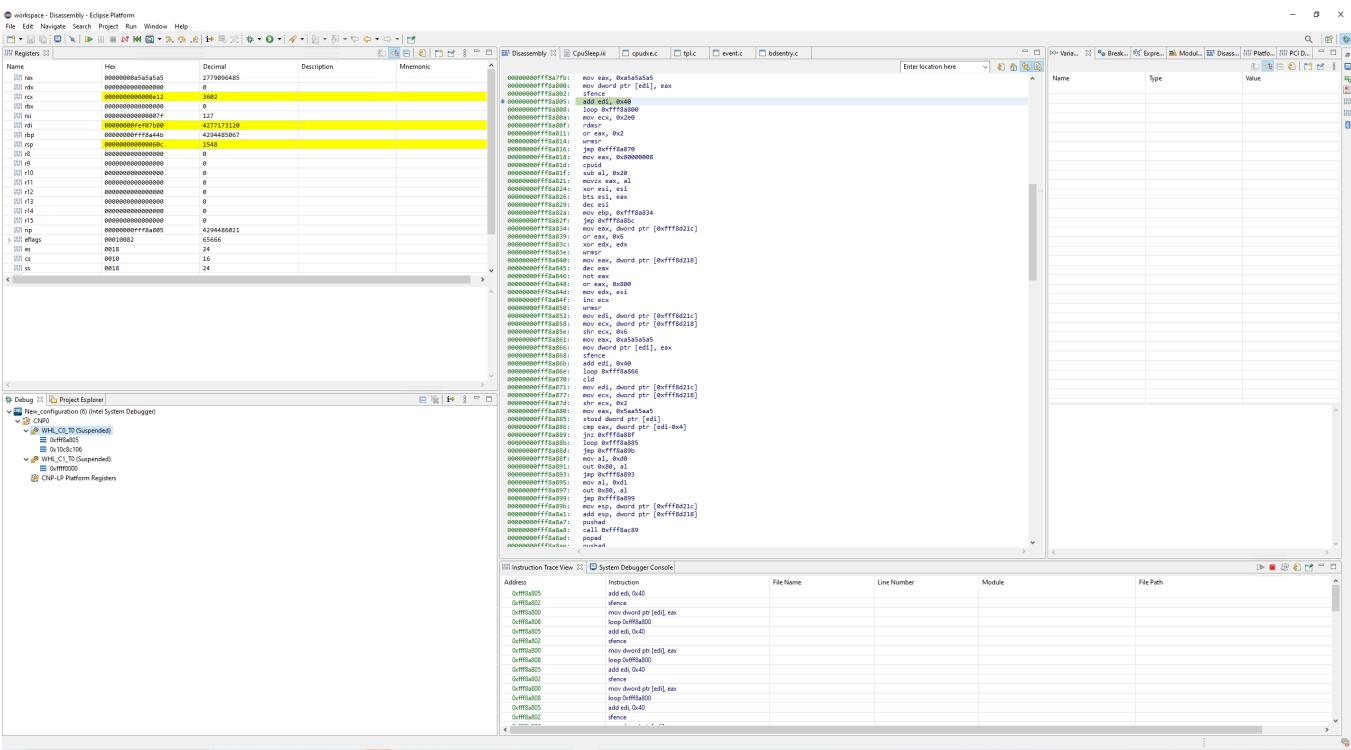
Flash the resulting `uxdbgapp.efi` to a USB stick, using the provided tools (described [here](#)) and boot from this on the UpXtreme target.

Basics of System Debug

So, lets use Target Connection Assistant to connect to the target. This is explained in section Connection to the Target.

```
intel.sysdbg.connect()
```

Once we are connected, we should see the following:



Now, we can halt the target using the button or `target.suspend()`.

We can then load the sources of the binary that is currently running on the target. This is done using the button or `threads[0].symbols.load_this()`

We can then evaluate the value of a symbol

Name	Type	Value
TempString	short *	0x00000000888baa18
OldConIn	_EFI_SIMPLE_TEXT_INPUT_PROTOCOL *	0x0000000000000000
ConInHandle	<Void> *	0x0000000000000000
Size	unsigned long long	0x0000000000000064
Status	unsigned long long	0x0000000000000000
Split	SPLIT_LIST*	N/A
y	int	-1287684615
x	int	N/A
z	int	12
mProfileList	short *	N/A
gEfiShellProtocol	_EFI_SHELL_PROTOCOL *	N/A
gST	EFI_SYSTEM_TABLE *	N/A
ShellInfoObject	SHELL_INFO	N/A

```
threads[0].frames[0].symbols.find("x")[0].value().string()
```

To go to the next instruction, we can perform step operations. There are three types of step:

Step into  continue execution into the function that is currently shown (not applicable in disassembly).

```
threads[0].runcontrol.step()
```

Step over  continue execution until the next line.

Source Lines:

```
threads[0].runcontrol.step(stepType=tcf.services.runcontrol.RM_STEP_OVER_LINE)
```

Instructions:

```
threads[0].runcontrol.step(stepType=tcf.services.runcontrol.RM_STEP_OVER)
```

Note

If we want to step in assembly, we should use instruction stepping mode - this is denoted with

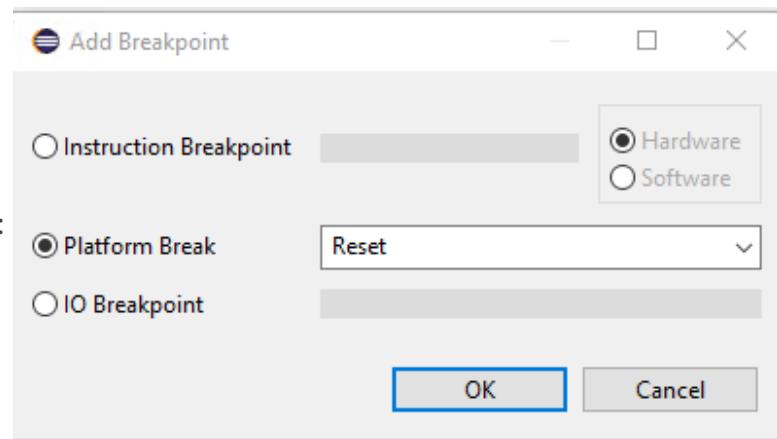


Step out:  continue execution until we leave the current function.

```
threads[0].runcontrol.step(stepType=tcf.services.runcontrol.RM_STEP_OUT)
```

Using System Debug to Find the Bug

Let's ensure the target is in the state we want it to be.



We do this by setting a reset breakpoint:



```
threads[0].breakpoints.add(condition="Reset")
```

Then we reset the target using . The target should then halt at the reset vector.

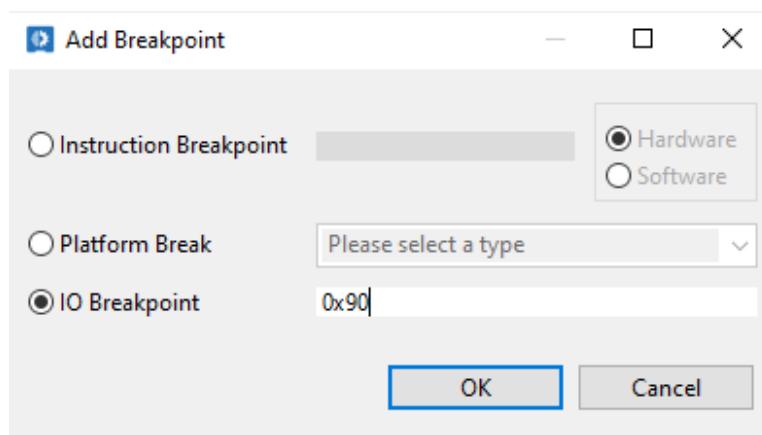
```
target.platform.reset()
```

Now, lets set the IO breakpoint. The **BAD** Cmake option in the efi application writes to port **0x90**.

This breakpoint can be set like this:



Add breakpoint

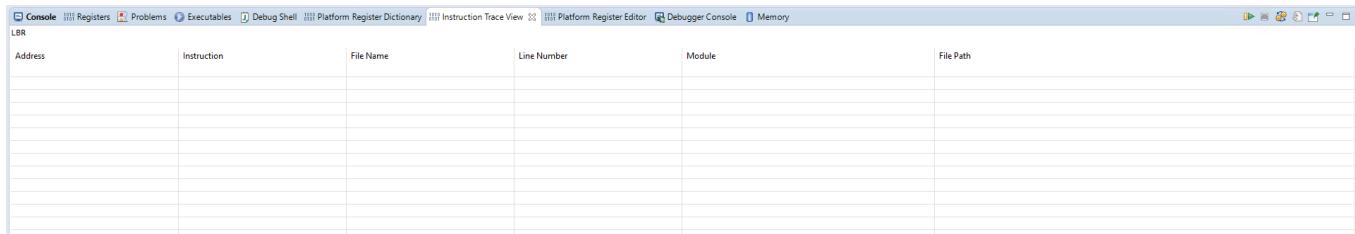


```
threads[0].breakpoints.add(condition="io_port:0x90")
```

Lets use Instruction Tracing to track the targets execution.

We do this by going to *Window-> Show View -> Instruction Trace View

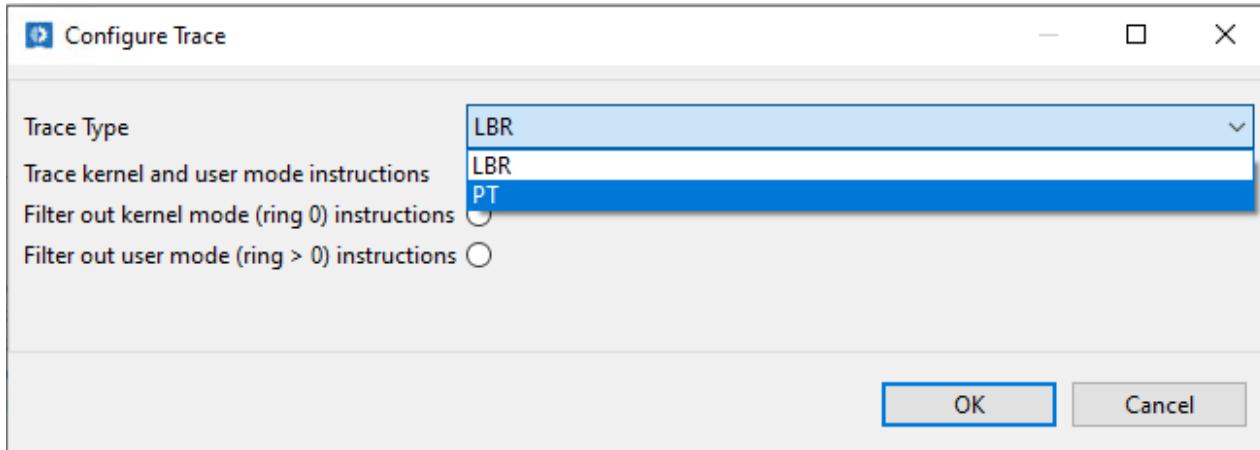
You should see the following window.



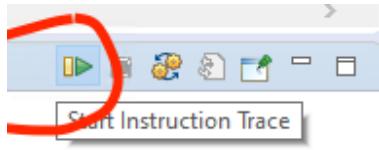
To enable trace, please click the configure button



Now, select *LBR* from the configuration menu.



Enable the trace with



Now, lets resume the target using the button. The target should then stop where the IO write happens.

```
target.resume()
```

After the target halts, the instruction trace should be shown in the instruction trace view.

```
threads[0].lbr.read(0, 100)
```

From here we can use the debugger features to figure out what went wrong in our code.

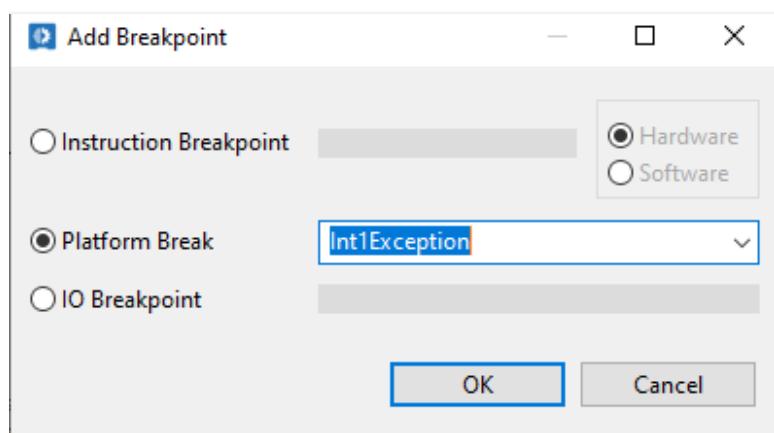
Using an Int1 to Communicate with the Debugger

Instead of using the IO port write, we can also use an Int1 exception to communicate with the debugger.

This is what the `notify_debugger_int(int value)` function does in the `main.c` file in the `uxdbgapp`.

This is the default method to communicate with the debugger.

This can be set using the following



```
threads[0].breakpoints.add(condition="Int1")
```

Note

Some targets don't behave well with the Int1 instruction, while other targets don't behave properly with the IO port write. Knowing which one to use depends on the machine.

```
#!/usr/bin/env python3
...
=====
Copyright © 2019 Intel Corporation

SPDX-License-Identifier: MIT
=====
...
OneApi System Debug example. Use System debugger for source Level debugging
...
import intel.sysdbg

intel.sysdbg.connect()

threads = intel.sysdbg.threads
target = intel.sysdbg.target

target.suspend()
threads[0].symbols.load_this()
threads[0].symbols.load_dxe()

threads[0].frames[0].symbols.find("x")[0].value().string()

threads[0].breakpoints.add(condition="Reset")

target.platform.reset()

target.resume()
```

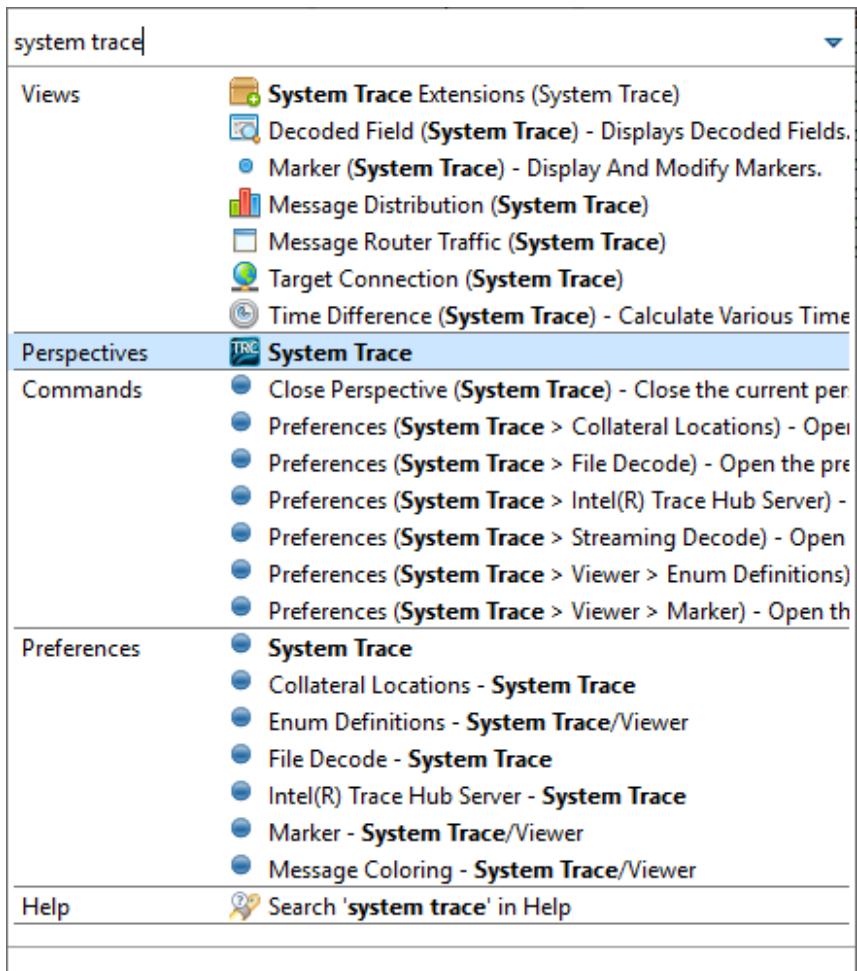
System Trace Usage Sample

Intel® System Debugger includes a Python*-based command line interface for capturing and decoding system trace called TraceCLI. Developers can use the TraceCLI Python package for automating trace capture and the System Trace UI for development and manual trace capture.

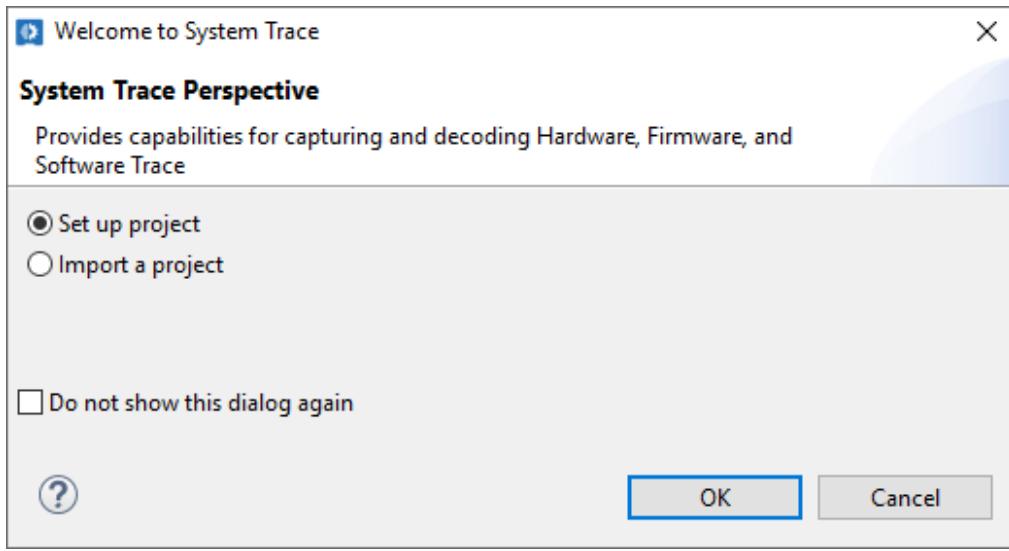
This project demonstrates decoding a trace file using the two methods mentioned above (trace UI and TraceCLI).

Using the System Trace UI

1. Launch Intel® System Debugger
2. Using the Eclipse Quick Access control (Ctrl+3), open the System Trace perspective:



3. In the System Trace wizard, select Set up project.



4. Enter a project name

New Trace Project

Create a new System Trace project.

Project name:

Use default location

Location:

Choose file system:



< Back

Next >

Finish

Cancel

5. Choose **Manually select target**. Select “Comet Lake U” and click Next.

Set Up System Trace Project

Select Or Detect Target

Click "Next" to continue manual target configuration.

Connect and detect target

Manually select target

Search: e.g. target name, target code name, component name, component code name

2nd Generation Intel® Xeon® Scalable Processors with Intel® C620 Series Chipsets (Purley Refresh)
Intel Xeon Processor (Cascade Lake), Intel C620 Series Chipset (Lewisburg)

Cedar Island
3rd Gen Intel® Xeon® Scalable Processor (Cooper Lake), Intel C620 Series Chipset (Lewisburg)

Generic 10th Gen Intel Core Processor (Comet Lake H) platform
10th Gen Intel Core Processor (Comet Lake H), Comet Lake PCH-H

Generic 10th Gen Intel Core Processor (Comet Lake S) platform
10th Gen Intel Core Processor (Comet Lake S), Comet Lake PCH-H

Generic 10th Gen Intel Core Processor (Comet Lake U) platform
10th Gen Intel Core Processor (Comet Lake U), Comet Lake PCH-LP

Generic 10th Gen Intel Core Processor (Ice Lake U) platform
10th Gen Intel Core Processor (Ice Lake U), Ice Lake PCH-LP

Generic 6th Gen Intel Core Processor (Skylake U) platform
6th Gen Intel Core Processor (Skylake U), 6th Gen Intel Core Platform I/O (Sunrise Point PCH-LP)

Components

CPU: 10th Gen Intel Core Processor (Comet Lake U) Stepping: K1

PCH: Comet Lake PCH-LP Stepping: A0

Note: Filtering for System Trace.

< Back Next > Finish Cancel

6. Select the Intel® Silicon View Technology (Intel® SVT) Debug Class (DbC) USB Debug Cable and click Next.

**Select Connection Configuration**

Please choose a connection configuration, then click on the "Next" button to continue to the connection summary page.

Physical Target**Intel SVT DbC USB Debug Cable**

Provider Type: OpenIPC (1.2026.4774.100)

Configuration Name: CML_CMP_DCI_DbC

Intel SVT DbC USB Debug Cable

This USB debug cable enables debug of platforms supporting Silicon View Technology, using DCI-USB2 or DCI-USB3. This is similar to a standard USB A male to A male cable, except that the VBus signal is removed. USB 3.0 compliant cable.

Links:

- * [Buy 1 meter cable](#)
- * [Buy 1.8 meter cable](#)

Intel(R) DCI OOB via Intel SVT Closed Chassis Adapter (CCA)

Provider Type: OpenIPC (1.2026.4774.100)

Configuration Name: CML_CMP_DCI_CCA



< Back

Next >

Finish

Cancel

7. Since this is a file decode, uncheck 'Configure provider' as well as 'Connect to target on finish'.

 Set Up System Trace Project

Connection Summary

Click on the "Verify Connected Target" button to check your connection configuration or click on the "Finish" button to create the connection.

Generic 10th Gen Intel Core Processor (Comet Lake U) platform

10th Gen Intel Core Processor (Comet Lake U), Comet Lake PCH-LP

Components

CPU: 10th Gen Intel Core Processor (Comet Lake U) Stepping: K1

PCH: Comet Lake PCH-LP Stepping: A0

Connection Configuration

Intel SVT DbC USB Debug Cable

Provider: OpenIPC (1.2026.4774.100) [Product]

Path: C:\Program Files (x86)\Intel\oneAPI\system_debugger\2021.1-beta09\tools\OpenIPC_1.2026.4774.100\

Configure Provider

▶ Provider Configuration

▶ Platform Debug Configuration

Supported Tools

System Debug

System Debug - Legacy (does not support current configuration)

System Trace

Connect to target on finish

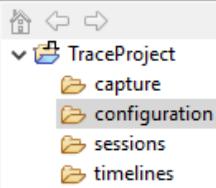
8. Provide a trace configuration name

Trace Configuration File

Create a new trace configuration file.

Enter or select the parent folder:

TraceProject/configuration



File name:

[Advanced >>](#)



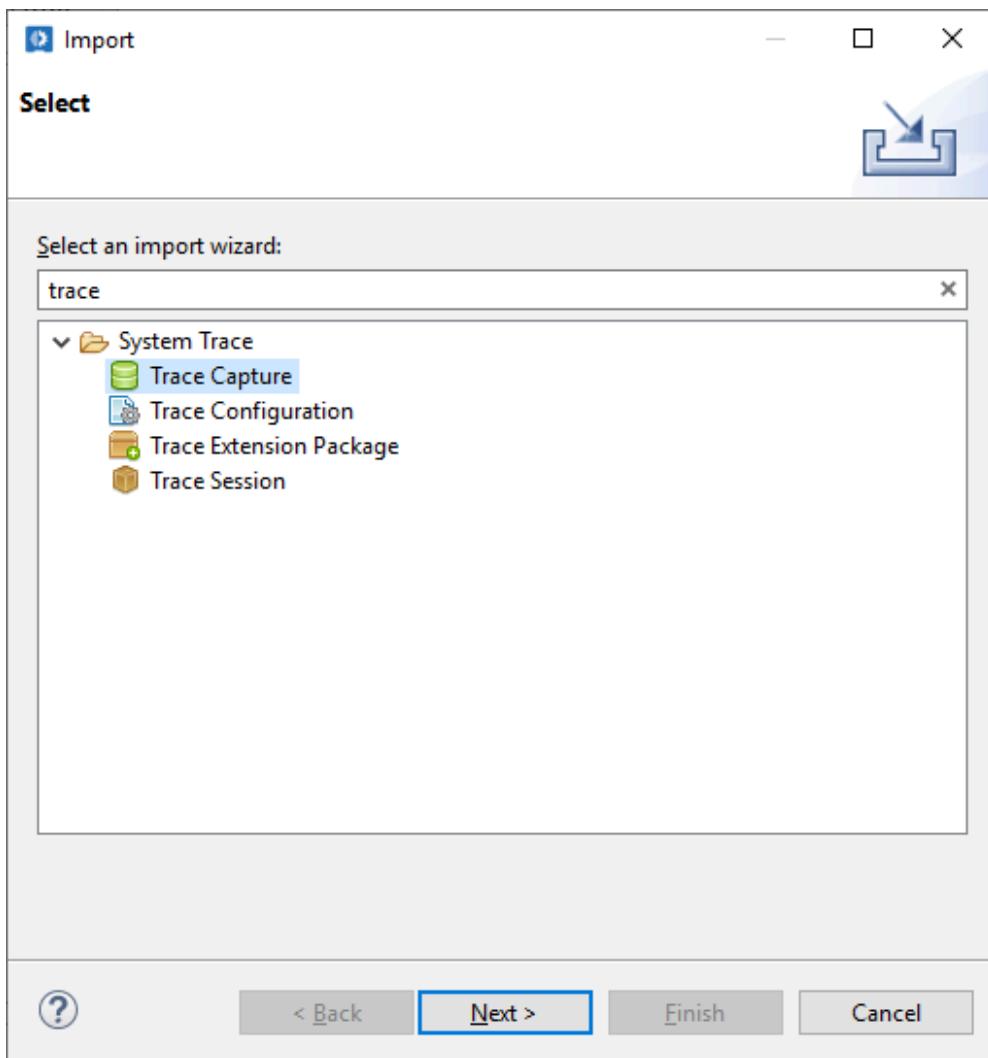
< Back

Next >

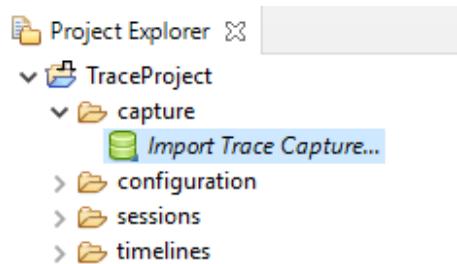
Finish

Cancel

9. Import Trace Capture.

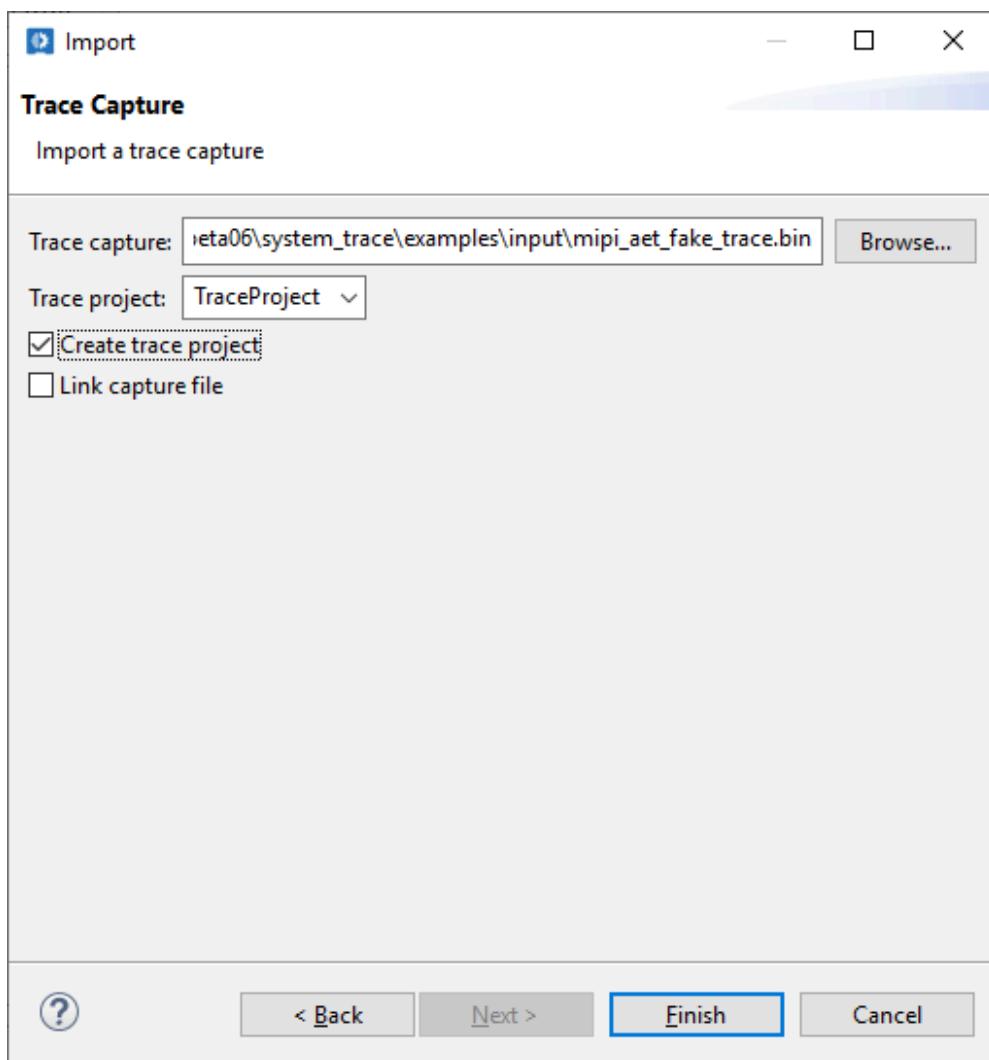


Alternatively, in the Project Explorer, expand the 'capture' node and click **Import Trace Capture**.

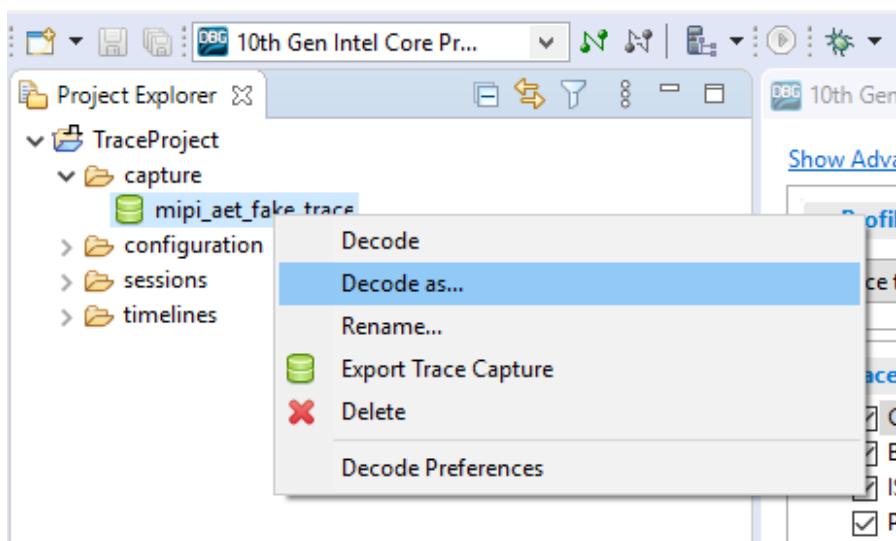


10. Select an example file at `<isd install`

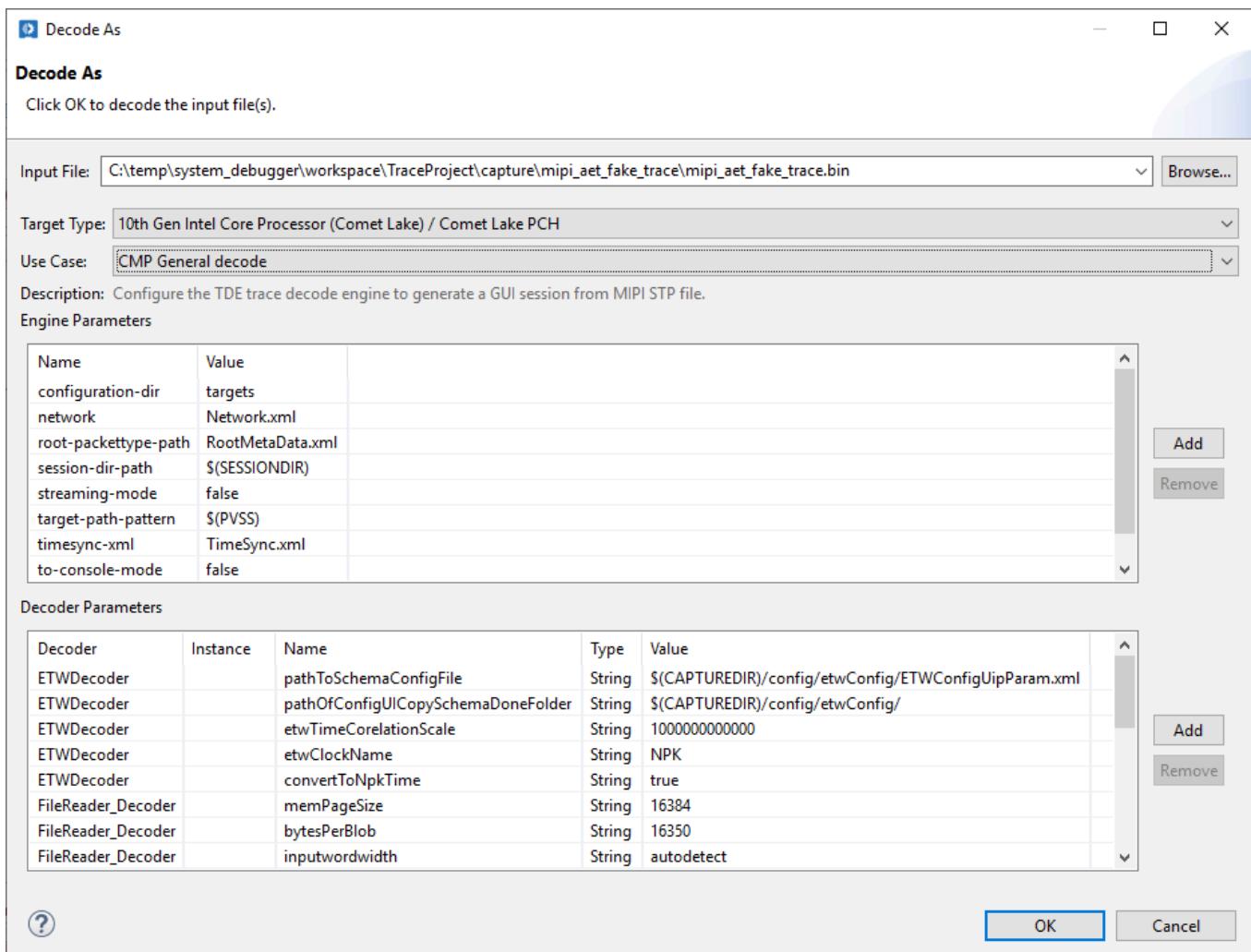
`folder>/system_trace/examples/input/mipi_aet_fake_trace.bin.bin`



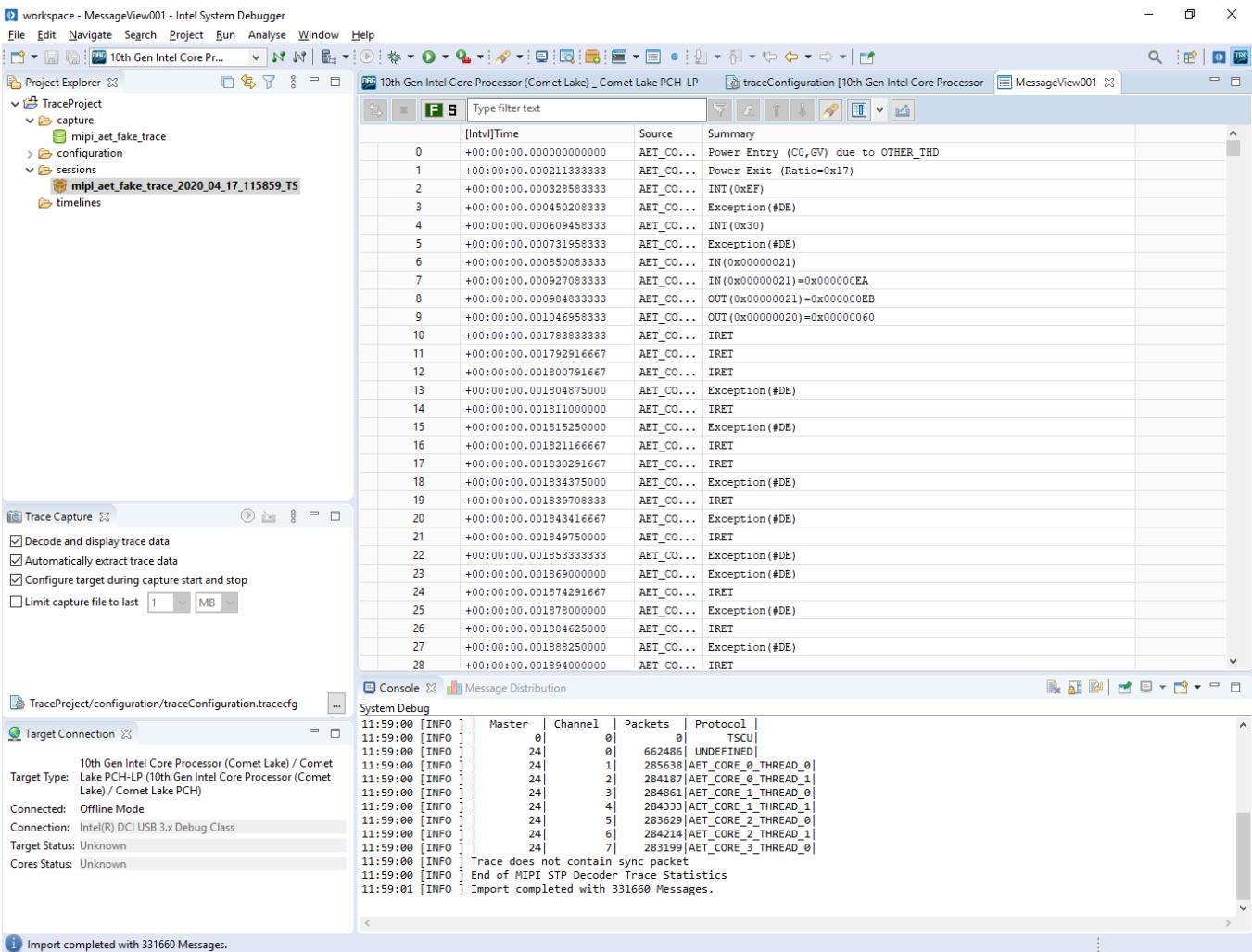
11. Select the imported capture for decoding



12. Ensure that **CMP General decode** use case is selected and click OK.



'MessageView001' opens showing decoded trace



Using the TraceCLI

TraceCLI has three usage models (console, file decode, and streaming)

```
> intel_tracecli --help
usage: intel_tracecli [-h] [-v] [--pvss PVSS] [--target TARGET]
                      [--usecase USECASE] [--transport TRANSPORT]
                      {console,decode,stream} ...
```

```
Intel TraceCLI Version 1.2003.826.200
Copyright Intel Corporation All rights reserved
```

```
positional arguments:
{console,decode,stream}
  console          Run interactive mode
  decode           Decode a trace capture file
  stream           Capture and decode traces
```

Running the Example

```
%ISS_PYTHON3_BIN% tracecli_example.py

> $ISS_PYTHON3_BIN/tracecli_example.py
Intel TraceCLI Version 1.2015.469.100
Copyright Intel Corporation All rights reserved

Using installation at C:\Program Files (x86)\inteloneapi\system_debugger\2021.1-beta06\System_Trace

Basic usage guideline for file decode:
1. session = trace.filedecode_session()
2. session.interactive_setup()
3. session.decode_file('ipc_trace_test.bin')

Basic usage guideline for streaming:
1. session = trace.stream_session()
2. session.interactive_setup()
3. session.start_stream_capture()
4. session.enable_trace()
5. session.disable_trace()
6. session.stop_stream_capture()

Other options (Examples):
- session.set_decoder_parameter('MIPI_Decoder', 'startAtAsync', 'false')
- session.csv_columns.extend(['MasterID','ChannelID','payload','Summary','PacketType'])

Info: MIPI STP Decoder Trace Statistics [instance: mipi]
| Master | Channel | Packets | Protocol |
| 0 | 0 | 0 | TSCU |
| 24 | 0 | 662486 | UNDEFINED |
| 24 | 1 | 285638 | AET_CORE_0_THREAD_0 |
| 24 | 2 | 284187 | AET_CORE_0_THREAD_1 |
| 24 | 3 | 284861 | AET_CORE_1_THREAD_0 |
| 24 | 4 | 284333 | AET_CORE_1_THREAD_1 |
| 24 | 5 | 283629 | AET_CORE_2_THREAD_0 |
| 24 | 6 | 284214 | AET_CORE_2_THREAD_1 |
| 24 | 7 | 283199 | AET_CORE_3_THREAD_0 |

Trace does not contain sync packet
End of MIPI STP Decoder Trace Statistics

"Time","Source","Summary"
"[000]0000:00:00.000000000000","AET_CORE_0_THREAD_1","Power Entry (C0, GV) due to OTHER_THD"
"[000]0000:00:00.000211333333","AET_CORE_1_THREAD_1","Power Exit (Ratio=0x17)"
"[000]0000:00:00.000328583333","AET_CORE_1_THREAD_0","INT(0xEF)"
"[000]0000:00:00.000450208333","AET_CORE_2_THREAD_0","Exception(#DE)"
"[000]0000:00:00.000609458333","AET_CORE_0_THREAD_0","INT(0x30)"
"[000]0000:00:00.000731958333","AET_CORE_0_THREAD_0","Exception(#DE)"
"[000]0000:00:00.000850083333","AET_CORE_1_THREAD_1","IN(0x00000021)"
"[000]0000:00:00.000927083333","AET_CORE_0_THREAD_1","IN(0x00000021)=0x000000EA"
"[000]0000:00:00.000984833333","AET_CORE_3_THREAD_0","OUT(0x00000021)=0x000000EB"
"[000]0000:00:00.001046958333","AET_CORE_0_THREAD_1","OUT(0x00000020)=0x00000060"
"[000]0000:00:00.001783833333","AET_CORE_1_THREAD_0","IRET"
"[000]0000:00:00.001792916667","AET_CORE_1_THREAD_0","IRET"
"[000]0000:00:00.001800791667","AET_CORE_0_THREAD_0","IRET"
"[000]0000:00:00.001804875000","AET_CORE_2_THREAD_1","Exception(#DE)"
"[000]0000:00:00.001811000000","AET_CORE_2_THREAD_0","IRET"
"[000]0000:00:00.001815250000","AET_CORE_2_THREAD_0","Exception(#DE)"

...
```

Intel® Debug Extensions for WinDbg Usage Sample

This project demonstrates how you would use Intel® Debug Extensions for WinDbg over Intel® Direct Connect Interface (Intel® DCI) to connect to a target and loading kernel symbols.

This sample works with Windows* 10 Host/Target Systems only. For a list of system requirements matching your configuration, see the Intel® System Debugger Release Notes.

Starting the Intel® Debug Extensions for WinDbg for IA JTAG Debugging

WinDbg must have access to kernel symbols for the connection to succeed. Thus, you must set the symbol file path beforehand.

Launch WinDbg (not by using the batch script, as this will perform a connection) and set the symbol file path by selecting **File > Symbol File Path** and adding:

```
srv*C:Symbols*http://msdl.microsoft.com/download/symbols
```

to the path or by setting the `_NT_SYMBOL_PATH` global environment variable with a corresponding value. Save the workspace by selecting **File > Save Workspace**, and then closing WinDbg.

! Note

You need to do this only once before the first use. Connect the host system to the target one with the **Intel® In-Target Probe (Intel® ITP)** or the **Intel® Direct Connect Interface (Intel® DCI)**.

Launching WinDbg

Intel® Debug Extensions for WinDbg can be launched as follows:

Click the desktop icon or open the Start Menu and click Intel® Debug Extensions for WinDbg under **Intel® System Debugger NDA**

Run `windbg_iajtag_console.bat` located at:

```
C:\IntelSWTools\oneapi_ndu\system_debugger\<version>\windbg_iajtag_console.bat
```

Another way to launch WinDbg over Intel DCI would be to use the `isd_shell`. Launch `isd_shell.bat`, that is located in the root installation directory:

```
C:\IntelSWTools\oneapi_ndu\system_debugger\<version>\isd_shell.bat
```

From `isd_shell` run `windbg_dci` to invoke WinDbg.

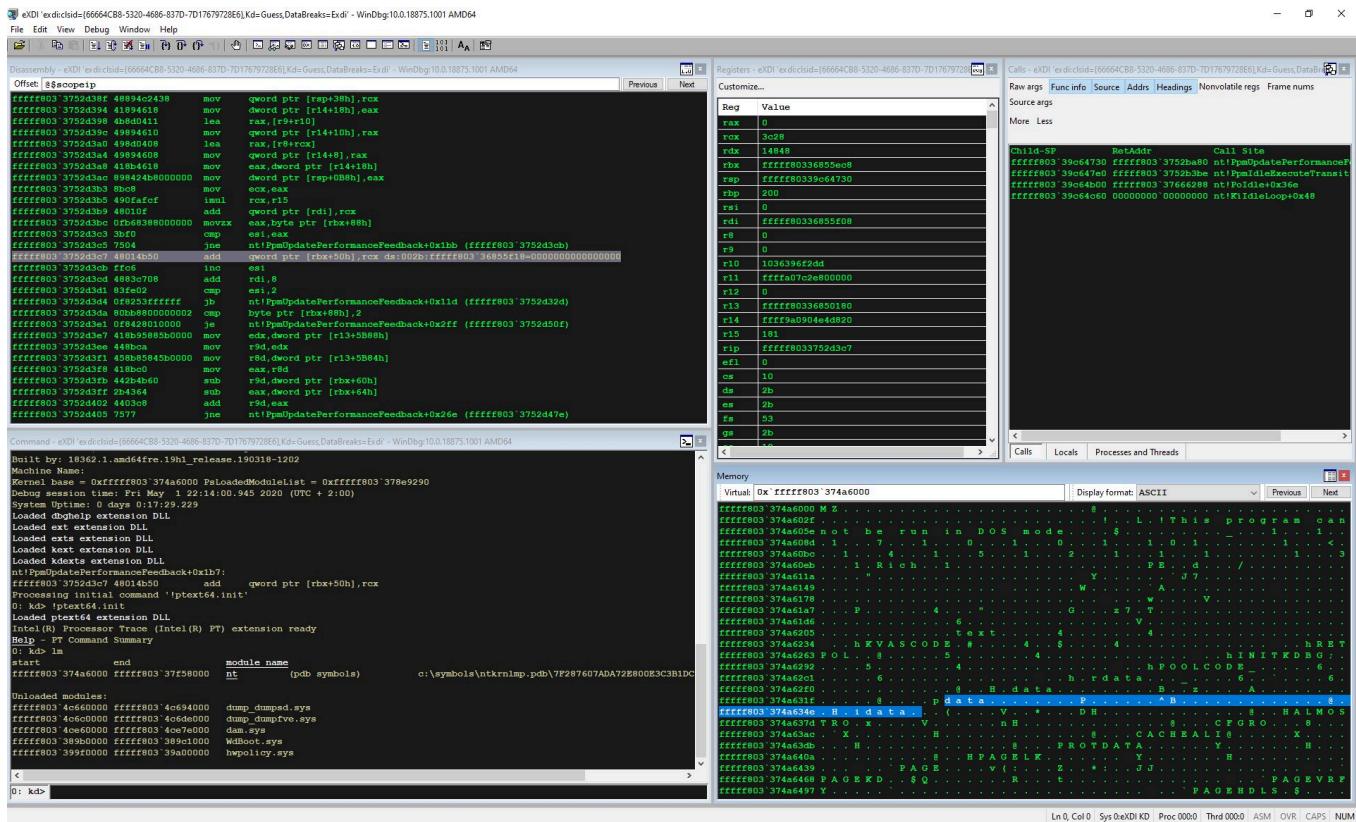
At this point, two Python objects are available for debugging:

`itp` - Intel® ITP interface

`itpkd` - wrapper over WinDbg and kernel debug console

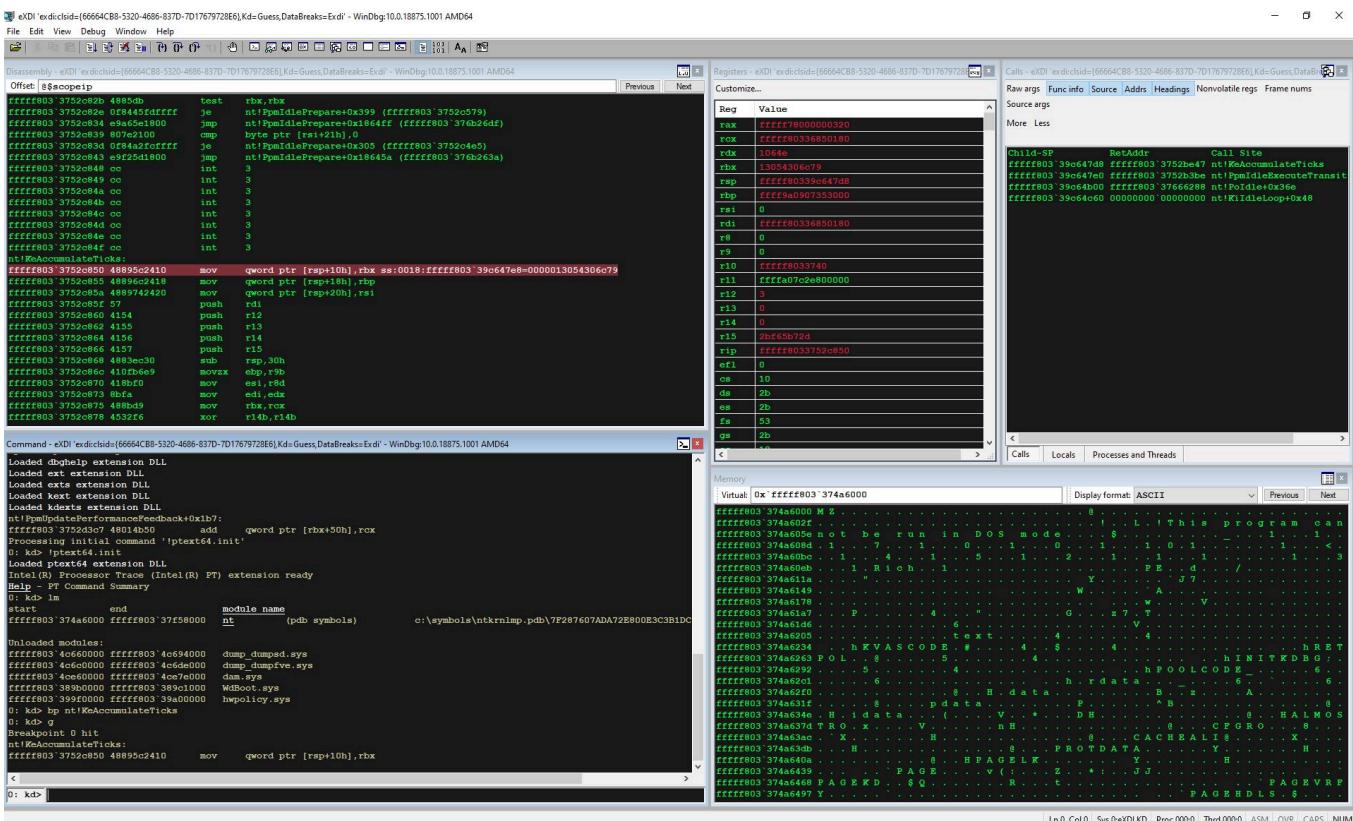


Execute `windbg()` to halt the target and run a WinDbg session. After that, WinDbg starts connecting to the target.

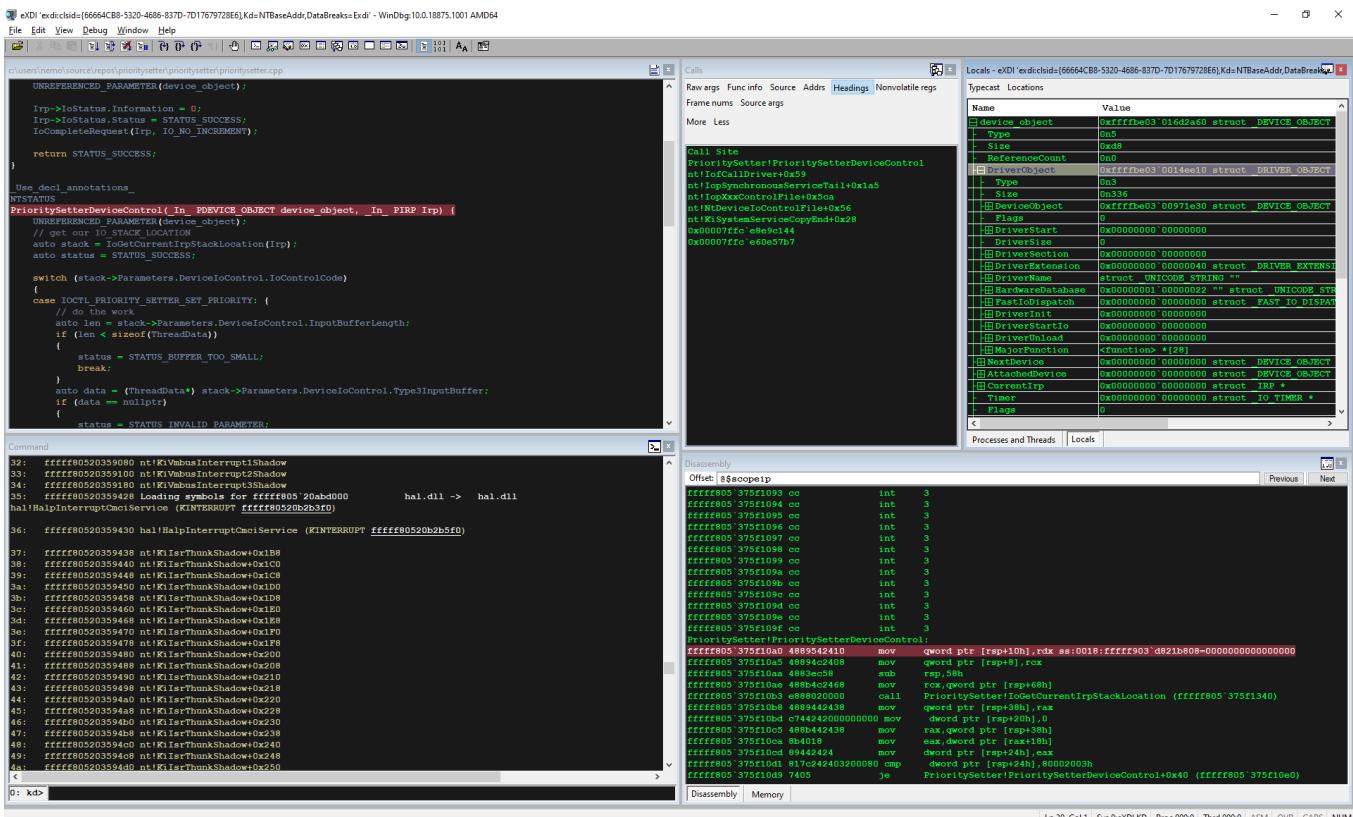


From now on, you can use WinDbg commands to interact with target, for example:

Set a breakpoint:



Debugging drivers:



Get [pcitree](#):

```

x64 WinDbg - WinDbg:10.18875.1001 AMD64
File Edit View Debug Window Help
Locals : x64\!edcida[6664CB8-5320-4686-837D-7D17679728E6]\Kd=NTBaseAddr,DataBreaks=x64 - WinDbg:10.18875.1001 AMD64
Locals
Name Value
Call Site
PrioritySetter!PrioritySetterDeviceControl
CallSite!IoCallDriver+0x5
nt!IoSynchExecutive+0x1a5
nt!IoGetThreadStackLocation+0x56
nt!Nt!SystemServiceCopyEnd+0x28
0x00007ffc e8e9c144
0x00007ffc e8e9c57b

Registers
Rawargs Funcinfo Source Addr Headings Nonvolatile regs
Frame num Source args
More Less

Processes and Threads
Locals

```

Command

```

32: fffff80520359080 nt!KiVmbusInterruptShadow
33: fffff80520359100 nt!KiVmbusInterrupt2Shadow
34: fffff80520359180 nt!KiVmbusInterrupt3Shadow
35: fffff80520359428 Loading symbols for fffff805 20ab4000 hal.dll -> hal.dll
hal!HalpInterruptCmService (KINTERRUPT fffff80520b2b5f0)
36: fffff80520359430 hal!HalpInterruptCmService (KINTERRUPT fffff80520b2b5f0)

37: fffff80520359438 nt!KiListThunkShadow+0x1B8
38: fffff80520359440 nt!KiListThunkShadow+0x1C0
39: fffff80520359448 nt!KiListThunkShadow+0x1C8
3a: fffff80520359450 nt!KiListThunkShadow+0x1D0
3b: fffff80520359458 nt!KiListThunkShadow+0x1D8
3c: fffff80520359460 nt!KiListThunkShadow+0x1E0
3d: fffff80520359468 nt!KiListThunkShadow+0x1E8
3e: fffff80520359470 nt!KiListThunkShadow+0x1F0
3f: fffff80520359478 nt!KiListThunkShadow+0x1F8
40: fffff80520359480 nt!KiListThunkShadow+0x200
41: fffff80520359490 nt!KiListThunkShadow+0x208
42: fffff80520359498 nt!KiListThunkShadow+0x210
43: fffff80520359498 nt!KiListThunkShadow+0x218
44: fffff805203594a0 nt!KiListThunkShadow+0x220
45: fffff805203594a8 nt!KiListThunkShadow+0x228
46: fffff805203594b0 nt!KiListThunkShadow+0x230
47: fffff805203594b8 nt!KiListThunkShadow+0x238
48: fffff805203594c0 nt!KiListThunkShadow+0x240
49: fffff805203594c8 nt!KiListThunkShadow+0x248
4a: fffff805203594d0 nt!KiListThunkShadow+0x250

```

0: x->

Note

Connecting to the target can take several minutes. Do not enter any commands until the connection is fully established and the connection confirmation message is displayed (Target initialization succeeded). If the connection fails, you see an exception in the console. Check the network connection if `KDVersionBlock` is not found. Run the target for a while if the kernel is not found.

Using Intel® Debug Extensions for WinDbg with WinDbg Preview*

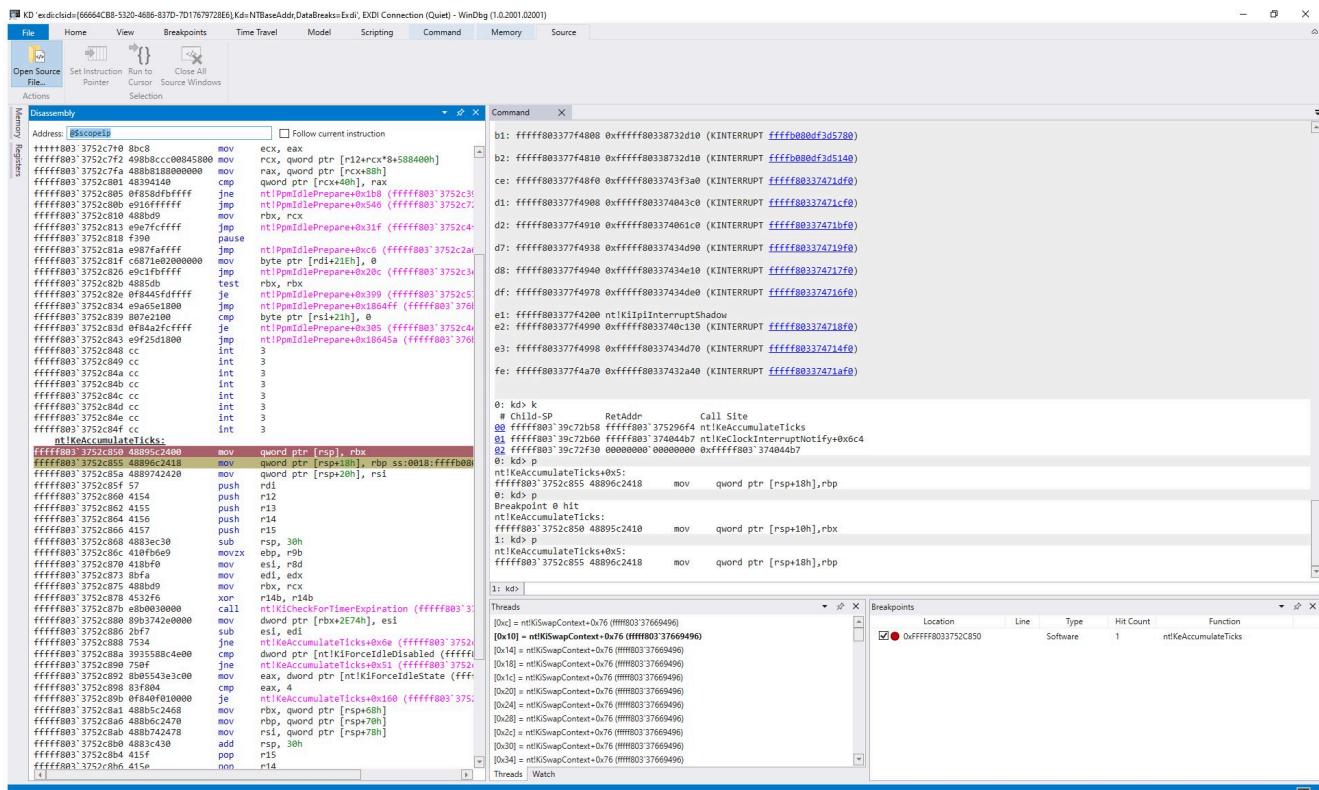
System requirements:

- Microsoft WinDbg Preview*. For download and installation instructions, see the Windows* Debugging Tools documentation.
- Windows 10 version 14257.0 or higher

Starting WinDbg Preview

To start using Intel® Debug Extensions for WinDbg with WinDbg Preview* follow the steps below:

1. Connect to the target.
2. Launch `isd_shell.bat` located in the installation root directory.
3. Execute the `windbg_preview` command to start WinDbg Preview*.



Cookbook

The cookbook is a collection of recipes that provide instructions to analyze a real-world issue using one or multiple components of Intel® System Debugger. Each recipe contains the following structural parts:

- Introduction to the concept or an issue type
- Ingredients required to reproduce the recipe
- Step-by-step scenario of achieving a goal.

The cookbook is regularly extended with new recipes.

If you need assistance or want to request a new recipe, contact Customer Support through [Online Service Center](#) or [Intel® Premier Support](#).

! Tip

To see a full-size version of an inline image, right-click it and select **Open image in new tab**. This method is especially useful for checking big screenshots.

Debug System Management Mode (SMM)

System Management Mode (SMM) is a special-purpose operating mode in CPUs based on Intel® architecture. SMM was designed for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

The following SMM mechanisms make it transparent to application programs and operating systems:

- The only way to enter SMM is by signaling a System Management Interrupt (SMI) through the SMI# pin on the processor or through an SMI message received through the Advanced Programmable Interrupt Controller (APIC) bus. The SMI is a non-maskable external interrupt and takes precedence over an Non-maskable Interrupt (NMI) and a maskable interrupt.
- The processor executes SMM code in a separate address space that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.
- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

Upon entering SMM, the processor switches to a new address space. The critical code and data of the SMI handler reside in a memory region referred to as *system-management RAM* (SMRAM). The processor uses a pre-defined region within SMRAM to save the processor's pre-SMI context. SMRAM can also be used to store system management information (such as system configuration and specific information about powered-down devices) and OEM-specific information.

Challenges in Debugging Code Running in SMM

SMI can occur at irregular intervals and detecting an SMI is a challenge by itself. Furthermore, on the Intel architecture, the debug features are controlled through a set of debug registers. The Debug Control Register (DR7) defines how the breakpoints set in Debug Address Registers should be interpreted by the processor. During SMM entry, the Debug Register DR7 is cleared. This disables software and hardware breakpoints that were set before SMM entry. Intel® System Debugger offers special features to overcome these challenges during SMM debugging.

Ingredients

- Software Tools: [Intel® System Debugger NDA](#)

- Debug Probe: [Intel® Silicon View Technology \(Intel® SVT\) Closed Chassis Adapter \(CCA\)](#) or [Intel® SVT Debug Class \(DbC\) USB Debug Cable](#)
- Hardware: Whiskey Lake U platform running [TianoCore EDK II BIOS](#) with Intel® Direct Connect Interface (Intel® DCI) debug interfaces enabled.

Connect Intel® System Debugger to the Target

Follow instructions to connect to the target. Ensure that a target connection has been successfully established.

Note

Refer to the troubleshooting section of this recipe for instructions on troubleshooting target connection issues.

Allow the target to boot up to a state where SMM debugging is needed.

Set up Breakpoints for Debugging SMM

Once the platform is ready for SMM debugging, the next step is to set up breakpoints in the code running within SMM. However, if you set breakpoints setup before entering SMM, they will be cleared at the SMM entry. To restore such breakpoints using Intel System Debugger, you can either:

- Intercept the SMM entry point
- Force SMI

See [instructions on both operations](#) to start SMM debugging.

Access the SMRAM State Save Map

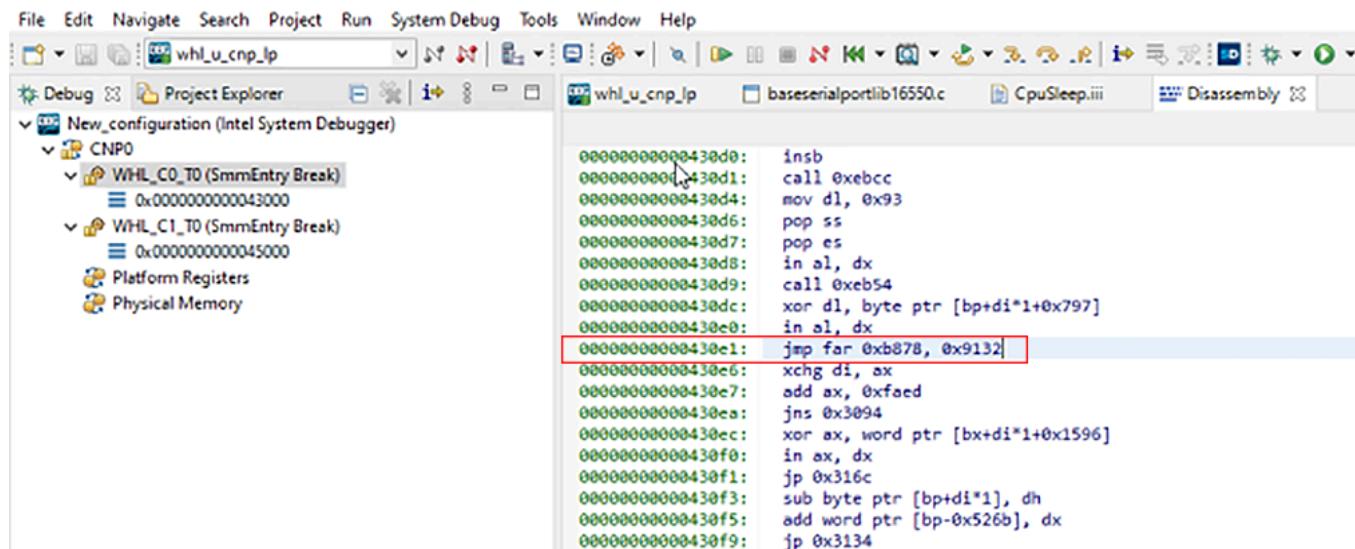
When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel® 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H]. User can access the register values of a particular thread using Intel® System Debugger. Refer to the [Accessing the SMRAM State Save Map](#) section in the product User Guide for instructions on how to display the State Save Map.

For more information of the SMRAM State Save Map, refer to the [Intel® 64 and IA-32 Architectures Software Developer Manuals](#).

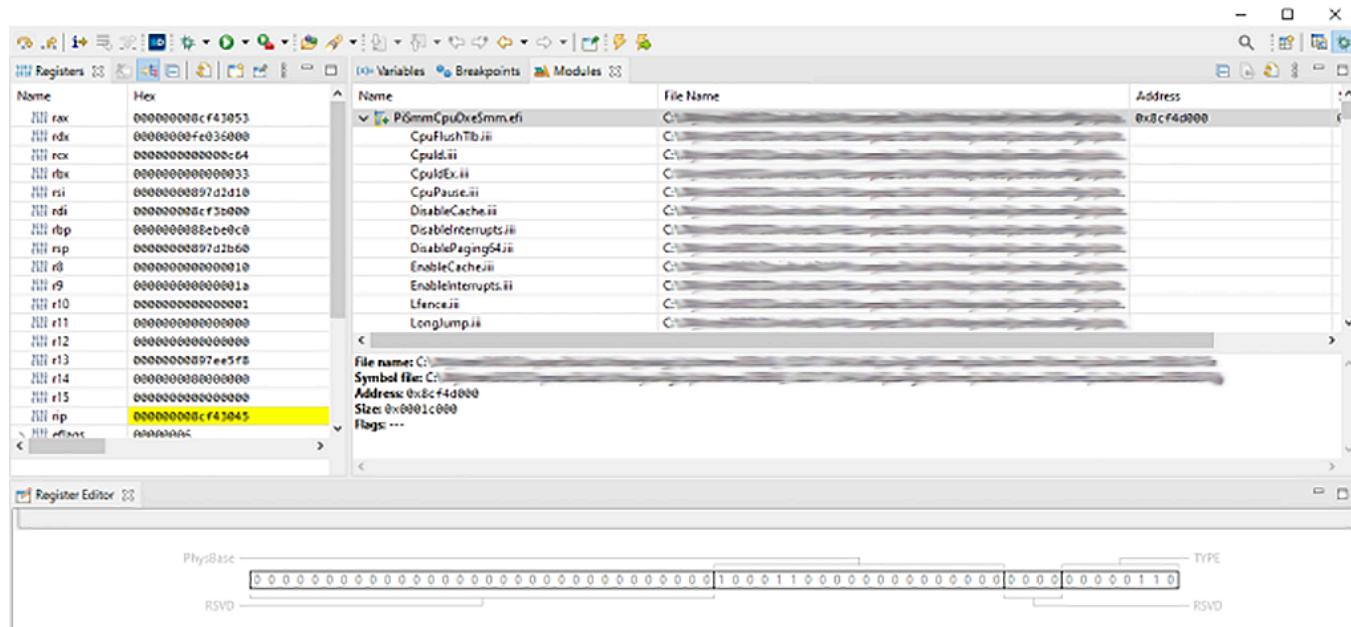
Source-Level Debugging within SMM

Various features available in Intel System Debugger for source-level debugging are also available for debugging source code running in SMM.

Debug information for SMM can be loaded only after switching to SMM protected mode. The CPU uses a `far jmp` instruction to jump to code located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an *intersegment jump*. After hitting the breakpoint for SMM entry, you have to manually step through a few instructions until the `far jmp` instruction is executed.



For loading the debug information, follow the default approach by [pressing the Load Available Debug Symbols \(formerly named Load This\) button](#). Additional settings like Path Mapping may be needed for loading debug symbols (see [Manual Loading](#)).

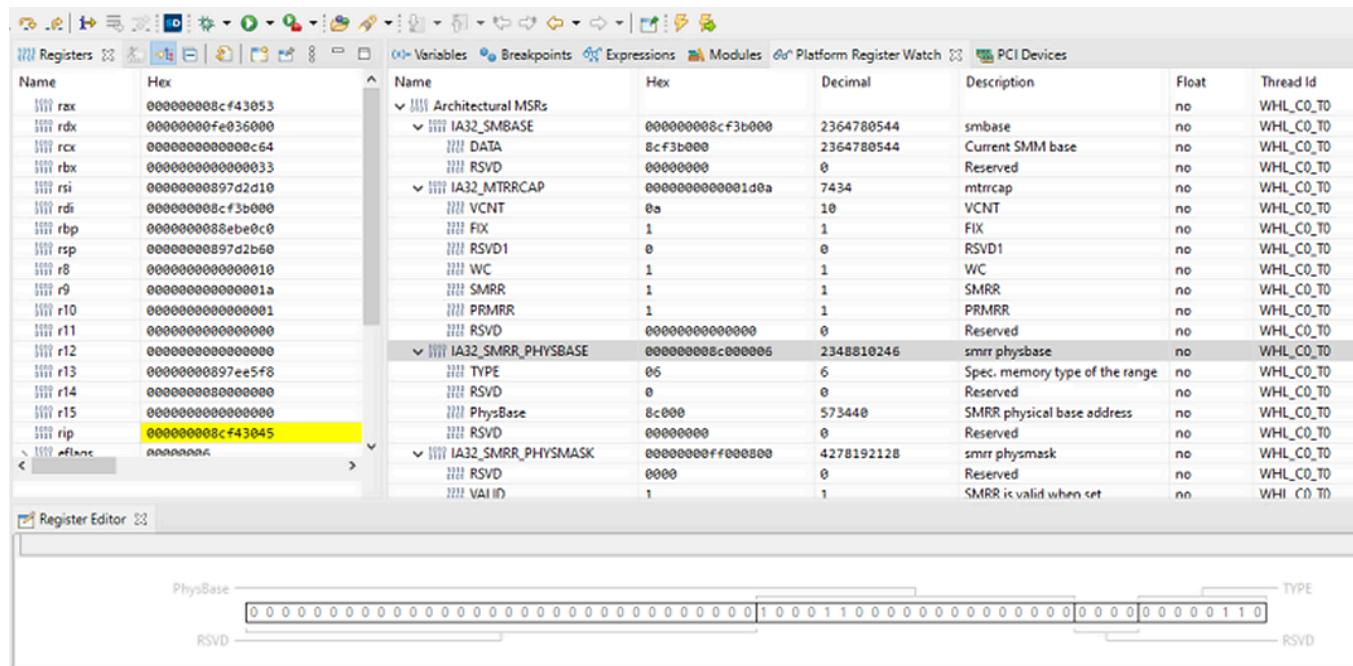


You can insert additional breakpoints within the SMM code following the [instructions](#). Any breakpoint inserted will be disabled at SMM-exit. However, these breakpoints can be restored again by using the SMM-entry breakpoint.

You may also benefit from the [Modules view](#), [Memory Browser view](#), [Variables view](#), and [others](#), while debugging SMM. The current [operation mode](#) of the processor is indicated at bottom left corner of the Eclipse tool.

View System-management Range Registers for SMM Debugging

SMM is configured through system-management range registers, which contain a pair of model-specific register (MSRs). For example, the [IA32_SMRR_PHYSBASE](#) MSR defines the base address for the SMRAM memory range and the memory type used to access it in SMM. The [IA32_SMRR_PHYSMASK](#) MSR contains a valid bit and a mask that determines the SMRAM address range protected by the SMRR interface. These MSRs can be viewed and/or modified using the [Platform Register Dictionary](#), [Platform Register Watch](#) and [Platform Register Editor](#) views in the Intel System Debugger.



Troubleshooting

Issue

Failed to connect to target

Solution

- **Hardware:** Design guidelines to enable CCA interface over USB interface are available in the Platform Design Guide (PDG). Failure to follow the recommended design guidelines may prevent the tool from communicating to the target.
- **Firmware:** Firmware running on the target must be configured to enable CCA interface.
- **BIOS:** If your target is executing BIOS, then the BIOS must be configured to enable CCA interface. For more information, refer to the BIOS user guide or [Intel System Debugger User Guide: Target Setup](#).

For additional help, refer to the platform documentation, primary [Troubleshooting](#) chapter, or contact your Intel representative. Priority support is available for Intel System Debugger NDA users through [Online Service Center](#) or [Intel® Premier Support](#).

Debug PCI Device

Intel® System Debugger – System Debug offers several features for debugging issues with a PCI device. This recipe demonstrates how to combine features like PCI device viewer, memory viewer, breakpoints, and others, to quickly identify issues with a PCI device or within its device driver. Possible issues may include a non-functional device due to a mismatch in device/vendor ID, invalid configuration of the PCI device, and more.

Ingredients

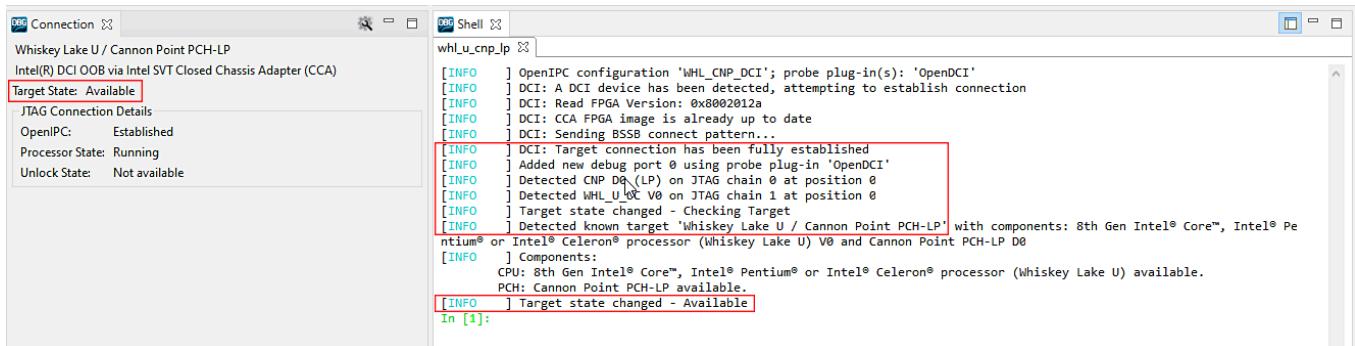
- Software Tools: [Intel® System Debugger NDA](#)
- Debug Probe: [Intel® Silicon View Technology \(Intel® SVT\) Closed Chassis Adapter \(CCA\)](#) or [Intel® SVT Debug Class \(DbC\) USB Debug Cable](#)
- Hardware: Whiskey Lake U platform running [TianoCore EDK II BIOS](#) with Intel® Direct Connect Interface (Intel® DCI) debug interfaces enabled.

Important

This recipe also demonstrates source-level debugging using the Intel® Serial IO UART Driver. For this step, it is necessary to have the BIOS build tree including the sources and symbols. The same instructions can also be applied for debugging issues in kernel drivers.

Connect Intel® System Debugger to the Target

Follow instructions to connect to the target. Ensure that a target connection has been successfully established.



Note

This recipe assumes that the platform is properly configured for Intel® DCI debugging. Refer to the troubleshooting section of this recipe for instructions on troubleshooting target connection issues.

Boot to EFI Shell

Once the target connection is fully established, do the following:

1. Create a debug configuration.
2. Select one of the CPU threads in the [Debug view](#) and halt execution using the CPU [run-control buttons](#) available in the Eclipse* toolbar.
3. To view the various PCI devices and their configuration, open the [PCI Devices view](#) (Windows > Show View > PCI Devices).

PCI Devices	Variables	Breakpoints	Expressions	Modules	Platform Register Watch
00:19.02					
00:1f.00 Cannon Point-LP LPC Controller					
00:13.00 Cannon Point-LP Integrated Sensor Hub					
00:08.00 Xeon E3-1200 v5/v6 / E3-1500 v5 / 6th/7th Gen Core Processor Gaussian Mixture Model					
00:15.00 Cannon Point-LP Serial IO I2C Controller #0					
00:1f.06 Ethernet Connection (6) I219-LM					
00:14.00 Cannon Point-LP USB 3.1 xHCI Controller					
00:15.01 Cannon Point-LP Serial IO I2C Controller #1					
00:1f.05 Cannon Point-LP SPI Controller					
00:1d.00 Cannon Point-LP PCI Express Root Port #15					
00:1f.04 Cannon Point-LP SMBus Controller					
00:14.02 Cannon Point-LP Shared SRAM					
00:00.00					
00:16.00 Cannon Point-LP MEI Controller #1					
00:04.00 Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor Thermal Subsystem					

4. Select the device to be analyzed from the drop-down list of PCI devices. Use the refresh button if needed.

For this recipe we are using the UART controller located on the Cannon Point PCH. Identify the UART controller based on its [Bus:Device.Function](#) (BDF) identifier. BDF of PCI devices are specified in the External Design Specification (EDS).

PCI Devices	Variables	Breakpoints	Modules	Platform Register Watch																																	
00:19.02																																					
<table border="1"><thead><tr><th>Name</th><th>Hex</th><th>Decimal</th></tr></thead><tbody><tr><td>▼ Type 0 Configuration Space Header</td><td></td><td></td></tr><tr><td>Header Type</td><td>80</td><td>128</td></tr><tr><td>BIST</td><td>00</td><td>0</td></tr><tr><td>BAR0</td><td>Fe036000</td><td>4261634052</td></tr><tr><td>BAR1</td><td>00000000</td><td>0</td></tr><tr><td>BAR2</td><td>00000000</td><td>0</td></tr><tr><td>BAR3</td><td>00000000</td><td>0</td></tr><tr><td>BAR4</td><td>00000000</td><td>0</td></tr><tr><td>BAR5</td><td>00000000</td><td>0</td></tr><tr><td>Cardbus CIS Pointer</td><td>00000000</td><td>0</td></tr></tbody></table>					Name	Hex	Decimal	▼ Type 0 Configuration Space Header			Header Type	80	128	BIST	00	0	BAR0	Fe036000	4261634052	BAR1	00000000	0	BAR2	00000000	0	BAR3	00000000	0	BAR4	00000000	0	BAR5	00000000	0	Cardbus CIS Pointer	00000000	0
Name	Hex	Decimal																																			
▼ Type 0 Configuration Space Header																																					
Header Type	80	128																																			
BIST	00	0																																			
BAR0	Fe036000	4261634052																																			
BAR1	00000000	0																																			
BAR2	00000000	0																																			
BAR3	00000000	0																																			
BAR4	00000000	0																																			
BAR5	00000000	0																																			
Cardbus CIS Pointer	00000000	0																																			

The PCI device viewer displays device configuration as shown in the image above. The PCI address domain consists of three distinct address spaces:

- Configuration

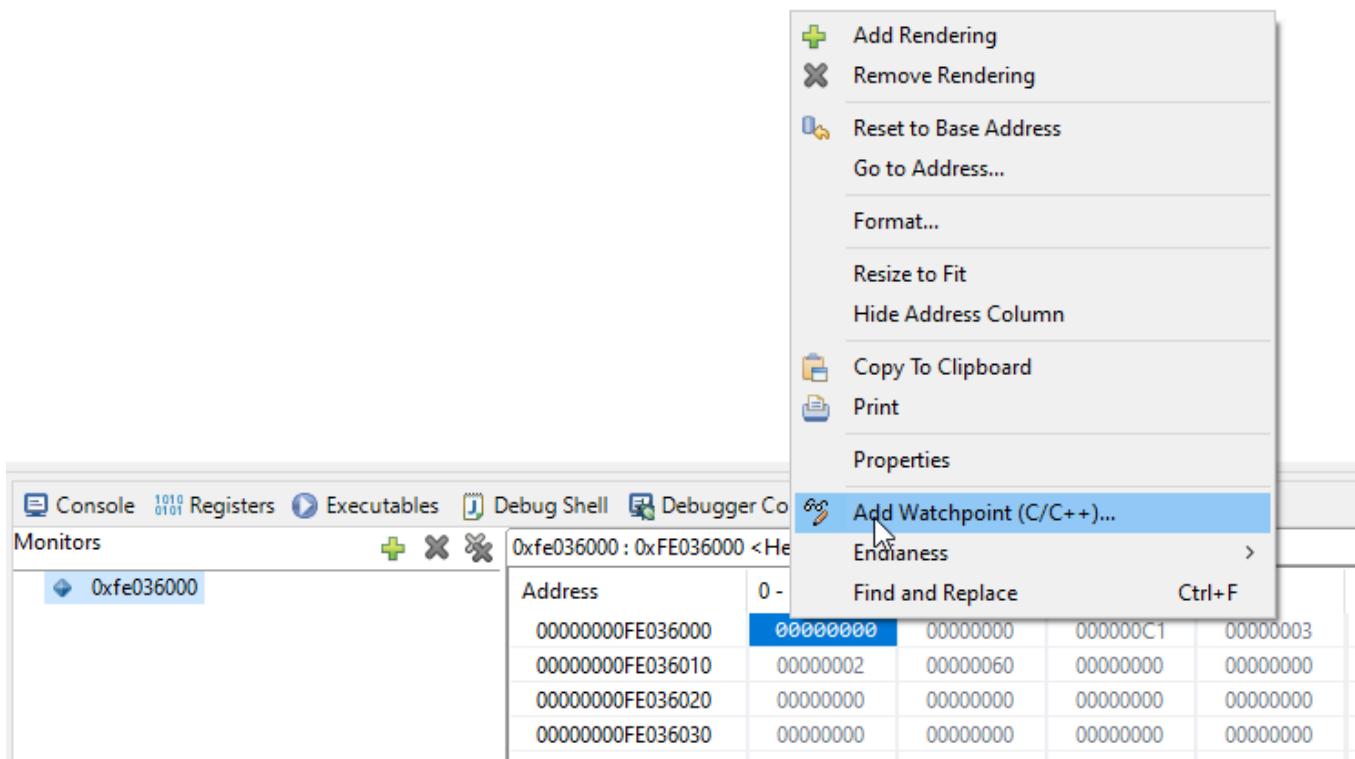
- Memory
- I/O space.

Each peripheral device contains a set of well-defined configuration registers in its PCI configuration space. These registers are used not only to identify devices but also to supply device configuration information. For example, Base Address Registers (BAR) in the device configuration space must be mapped before a device can respond to data access. System firmware assigns regions of memory space in the PCI address domain to PCI peripherals. The base address of a region is stored in the BAR. Device addresses in memory space are memory-mapped into the host address domain so that data access to any device can be performed by the processor's native load or store instructions.

For the UART controller in the figure above, the BAR0 holds the system memory base address of the controller assigned by the Intel Serial IO UART Driver during the PCI device initialization phase. You can find the details of this UART controller in the External Design Specification (EDS) Volume 2 for Canon Lake PCH, which is available at [Resource & Documentation Center](#). According to EDS, various device registers including the Receive Buffer Register (RBR), Transmit Holding Register (THR), and others are memory-mapped with the base address in the BAR0. You can view these device registers by mapping the contents of the BAR0 in the Memory view. Refer to the [instructions how to view the memory contents](#).

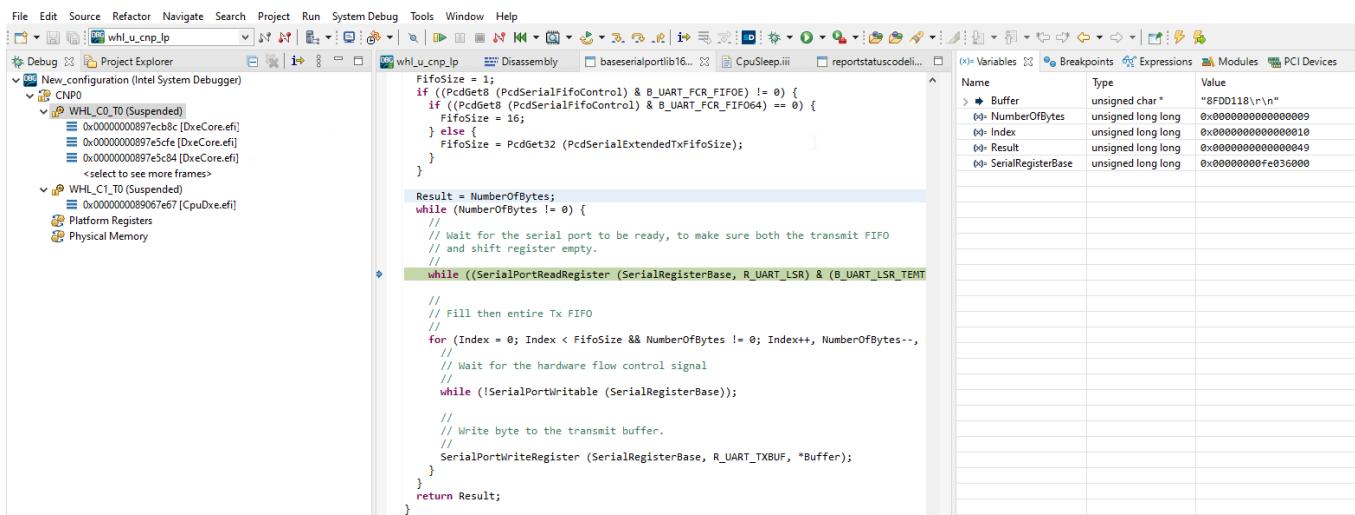
Address	0 - 3	4 - 7	8 - B	C - F
00000000FE036000	00000000	00000000	000000C1	00000003
00000000FE036010	00000002	00000060	00000000	00000000
00000000FE036020	00000000	00000000	00000000	00000000
00000000FE036030	00000000	00000000	00000000	00000000
00000000FE036040	00000000	00000000	00000000	00000000
00000000FE036050	00000000	00000000	00000000	00000000
00000000FE036060	00000000	00000000	00000000	00000000
00000000FE036070	00000000	00000000	00000000	00000006

[Watchpoints](#) feature available in Intel® System Debugger – System Debug can be used to halt the execution of the CPU when any of these device registers are modified. For example, setup a watchpoint at the base address (0xe036000) listed under BAR0 and resume the cores. The cores will be halted again when they access the UART device register mapped to address 0xe036000.



Once the cores are halted, use the [Load available debug symbols](#) function to load the source code. Path mapping may be necessary if the BIOS was not built on the same host where the debugger is running. Follow the instructions on [manual symbol loading](#).

In this example, the cores were halted when the Intel Serial IO UART Driver checked the status of the UART buffers. The variable `SerialRegisterBase` used to access the UART registers points to the address in the BAR0. You can confirm this by checking the value of this variable listed in the [Variables view](#).



Troubleshooting

Issue

Failed to connect to target

Solution

- Firmware: Firmware running on the target must be configured to enable CCA interface.
- BIOS: BIOS must be configured to enable CCA/DbC interface. For more information, refer to the BIOS user guide or [Intel System Debugger User Guide: Target Setup](#).

For additional help, refer to the platform documentation, primary [Troubleshooting](#) chapter, or contact your Intel representative. Priority support is available for Intel System Debugger NDA users through [Online Service Center](#) or [Intel® Premier Support](#).

Capture and Decode Early-boot Intel® Converged Security and Management Engine (Intel® CSME) Traces in a Cold Boot Scenario

Platform boot failures can be quickly analyzed with the help of Intel® CSME traces captured using Intel® System Debugger. This approach is especially useful when the platform hangs during the early boot phases and the CPU does not exit the reset mode.

To capture the Intel® CSME traces, you must configure the Intel® Trace Hub (Intel® TH), also known as North Peak (NPK), located on the PCH. This is often a challenge during a cold-boot scenario where the platform hangs soon after its powered on and you do not have time to configure the Intel TH from the Intel® System Debugger – System Trace. Any settings applied on Intel TH using the Intel System Debugger in advance are lost during the power-cycle.

This recipe provides you instructions on how to halt the platform at the platform-boot-stall so that the Intel TH can be configured using Intel® System Debugger - System Trace. This allows you to capture Intel® CSME traces right from the Intel® CSME boot phase.

Ingredients

- Software Tools: [Intel® System Debugger NDA](#)
- Debug Probe: [Intel® Silicon View Technology \(Intel® SVT\) Closed Chassis Adapter \(CCA\)](#)
- Hardware: Platform supported by System Trace (see [Supported Platforms](#)) and on which the issue can be reproduced.
- Collaterals: Intel® CSME trace decoder (optional).

! Note

The Intel® CSME trace decoder ([.traceext](#)) is part of the Intel® CSME kit available at the [Resource & Documentation Center](#). This decoder is required only if you want to decode the captured Intel® CSME traces. You can skip this step and send the captured traces to Intel for further analysis.

Connect Intel® System Debugger to the Target

1. Follow instructions to connect to the target.
2. Ensure that the PCH is detected by the tool. See example output in the [ISD Shell](#) as below:

```

Modules:
  isd      - Intel(R) System Debugger CLI (v3.0.4985.100) [Loaded]
  tca      - Target Connection Assistant CLI (v3.0.4985.100) [Loaded]
  ipcccli  - OpenIPC CLI (v1.2110.2178.100) [Loaded]
  sysdbg   - System Debug CLI (v1.21122+27ce0) [Loaded]
  trace    - System Trace CLI (v1.2112.1993.200) [Loaded]
  crashlog - Intel(R) Crash Log Framework (v3.43.21114.200) [Loaded]
  trace_agent - System Trace Target Agent CLI [Loaded]

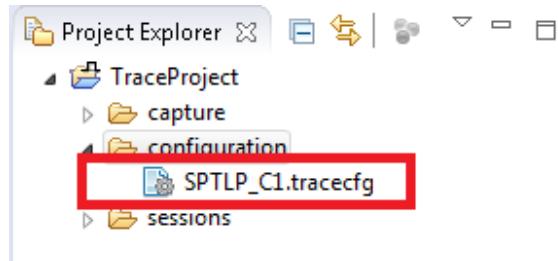
[INFO ] OpenIPC configuration 'WHL_CNP_DCI'; probe plug-in(s): 'OpenDCI'
[INFO ] DCI: A DCI device has been detected, attempting to establish connection
[INFO ] DCI: Read FPGA Version: 0x8002012a
[INFO ] DCI: CCA FPGA image is already up to date
[INFO ] DCI: Sending BSSB connect pattern...
[INFO ] DCI: Target connection has been fully established
[INFO ] Added new debug port 0 using probe plug-in 'OpenDCI'
[INFO ] Detected CNP D0 (LP) on JTAG chain 0 at position 0
[INFO ] Detected WHL_U_UC V0 on JTAG chain 1 at position 0
[INFO ] Survivability Preset was requested, but no supported devices were detected. Preset will be applied when a supported device is detected.
[INFO ] Target state changed - Checking Target
[INFO ] Detected known target 'Whiskey Lake U / Cannon Point PCH-LP' with components: 8th Gen Intel® Core™, Intel® Pentium® or Intel® Celeron® processor (Whiskey Lake U) V0 and Cannon Point PCH-LP D0
[INFO ] Components:
  CPU: 8th Gen Intel® Core™, Intel® Pentium® or Intel® Celeron® processor (Whiskey Lake U) available.
  PCH: Cannon Point PCH-LP available.
[INFO ] Target state changed - Available
[INFO ] IPC-CLI: 1.2110.2178.100, OpenIPC:Main (rev 669752) : 1.2110.5203.200
In [1]: █

```

Create and Configure a System Trace Project

Once the target connection is fully established, do the following:

1. Open the System Trace perspective in Eclipse* and create a System Trace project.
2. Select trace sources. If a corresponding view for the newly created `.tracecfg` file has not opened after the project setup, double-click the file under the **configuration** node in the **Project Explorer**.



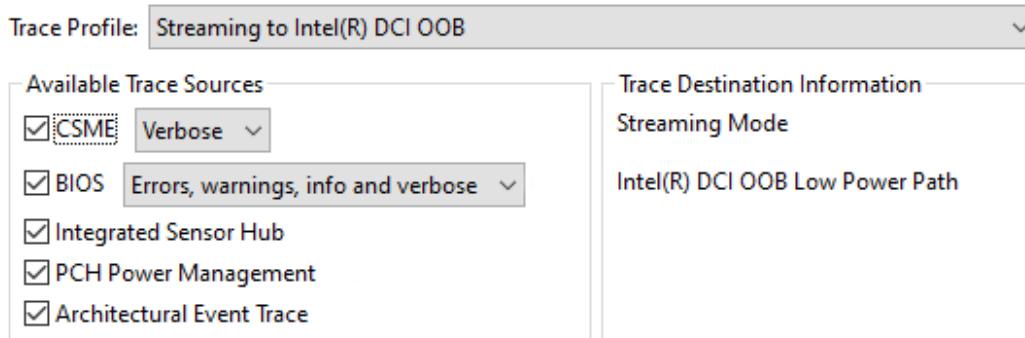
3. Select Intel® Direct Connect Interface (Intel® DCI) Out of Band (OOB) as a trace profile. In the **Trace Configuration tab (Profiles pane)**, set **Trace Profile** to *Intel® DCI OOB* from the drop-down list.

This setting allows the Intel TH to live stream trace information over the Intel SVT CCA debug probe.

! Note

You can also use the *Intel® Trace Hub Memory* as an alternative to streaming over Intel SVT CCA. *Intel® Trace Hub Memory* is an on-chip RAM, which can store few kilobytes of trace data and is available right from the platform boot phase. Other trace destinations (for example, *BIOS Reserved Trace Memory*) is not recommended as DRAM is not initialized and USB Debug Class (also known as USB DbC) may not be available at this stage. The exact boot phase at which a USB DbC connection can be established may vary across platforms. For more information, refer to the platform documentation or contact your Intel representative.

4. Under the Trace Sources (Trace configuration tab), ensure that Intel® Converged Security Engine (Intel® CSE) or Intel® Converged Security and Management Engine (Intel® CSME) is selected. Also select other trace sources as needed (PMC).



5. (Optional) If you want to decode the trace (not only capture it), install the Intel® CSME trace decoder extension (`.traceext`) because the Intel® CSME traces captured by the tool will not be automatically decoded. In this case, they may appear as hexadecimal codes in the summary column of the [Message View](#).

To decode the Intel® CSME trace, download a corresponding Intel® CSME Kit from the [Resource & Documentation Center](#) and import the Trace Extension file as follows:

1. In the **Project Explorer**, right-click any empty space and select **Import**
2. In the opened dialog box, expand the **System Trace** node, select **Trace Extension Package**, and click **Next**.
3. Click **Browse** to find and add the `.traceext` file from the downloaded Intel® CSME kit.
4. Click **Finish**.

Halt the Target at Platform Boot and Capture System Trace

1. Configure the platform to halt at the platform boot phase. You can do it using the OpenIPC command line interface, in two different ways:

- Using the [ISD Shell](#) available inside Eclipse:

- Once TCA successfully connected to the target execute the following command:

```
itp.stalls.platformboot=1
```

- Using the standalone shell:

- Launch the shell using the [isd_shell.bat](#) under the installation path (by default, it is `C:\IntelSWTools\system_debugger\<version>-nda`).
 - Start an IPC CLI session using the command

```
ipccli
```

- Once a successful connection is established, execute the following command to enable platform-boot-stall:

```
itp.stalls.platformboot=1
```

At this stage, the target platform is configured to halt its execution at the platform boot phase.

- Power-cycle the target by disconnecting power to the target. Wait a few seconds before re-enabling power to the target.

Once powered, the target will halt itself at an early stage of the platform boot phase. OpenIPC will try to connect to the target automatically once power is re-established. If not, use the TCA Connect button to reconnect the target.

- [Start trace capture](#) to apply the configuration created earlier to the Intel TH.

- Release the target from the platform-boot-stall using the below command (to be executed in the ISD Shell):

```
itp.stalls.platformboot=0
```

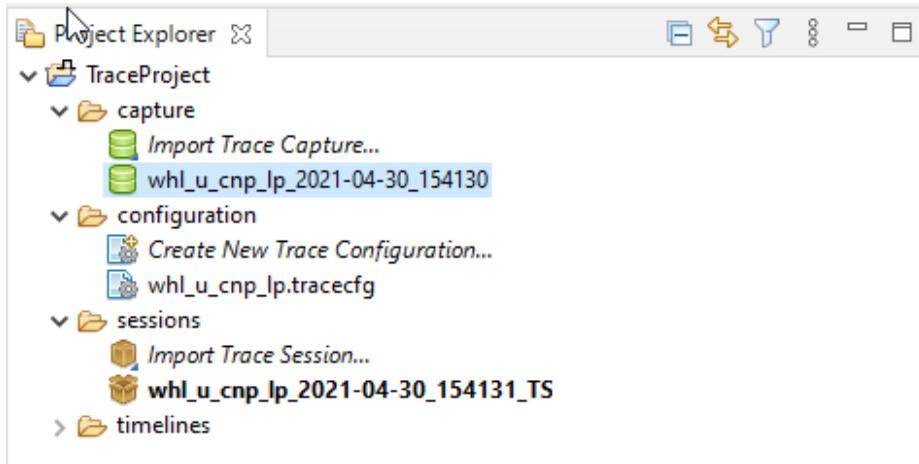
Wait a few seconds for the platform to boot-up or wait until the platform enter the hung-state.

- Traces will appear in the [Message View](#) if live tracing over Intel® Direct Connect Interface (Intel® DCI) Out of Band (OOB) was selected under the [Trace Profile](#).

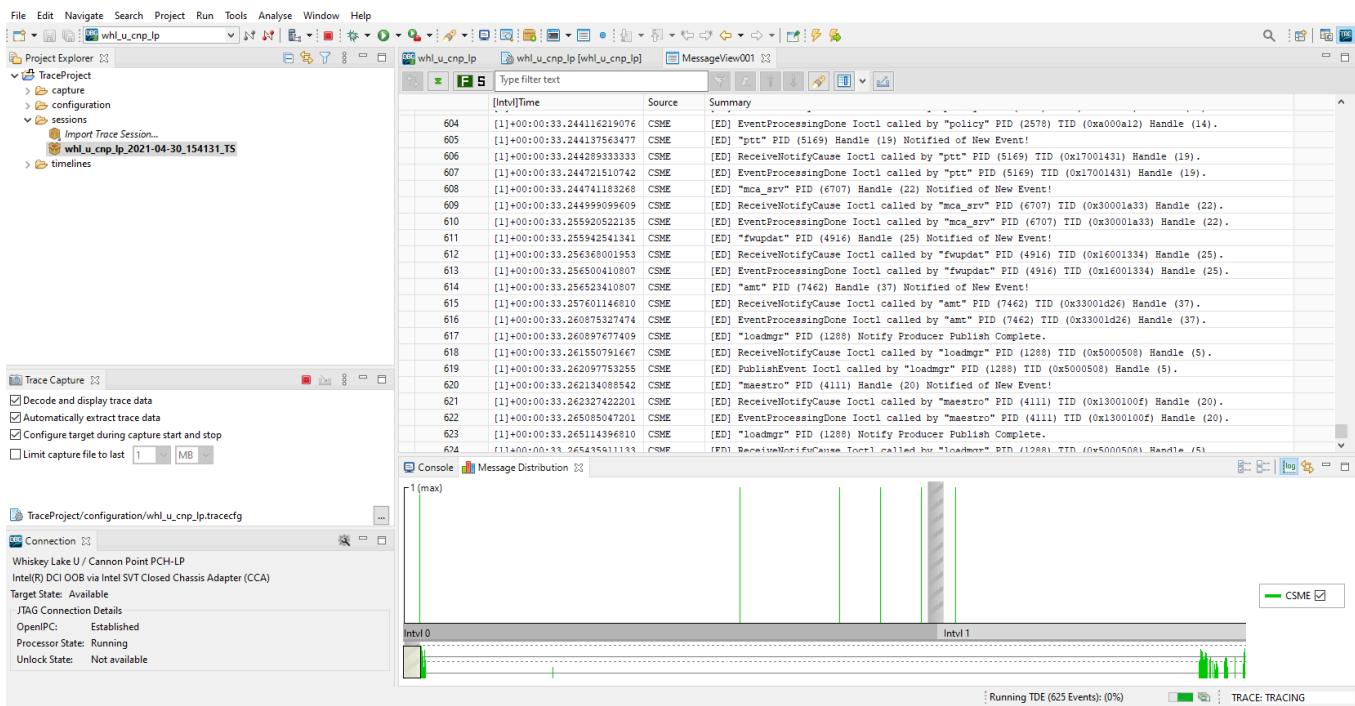
If you selected tracing to Intel® Trace Hub Memory, [Stop trace capture](#) using the buttons to extract the traces.

Decode and Analyze System Trace

The trace data captured by the tool is stored in a capture file and is listed under the capture folder within the Trace Project (see **Project Explorer** in Eclipse).



The decoded traces should be visible in the **Message View**.



Refer to the primary [System Trace User Guide](#) to learn more about the advanced features for analyzing System Trace within Eclipse.

Troubleshooting

Issue

Failed to connect to target

Solution

- **Hardware:** Design guidelines to enable CCA interface over USB interface are available in the Platform Design Guide. Failure to follow the recommended design guidelines may prevent the tool from communicating to the target.
- **Firmware:** Firmware running on the target must be configured to enable CCA interface.
- **BIOS:** If your target is executing BIOS, then the BIOS must be configured to enable CCA interface. For more information, refer to the BIOS user guide or [Intel System Debugger User Guide: Target Setup](#).

This step is not needed for capturing early-boot Intel® CSME traces. However, the interface may be disabled or Intel TH may be reconfigured by the BIOS when it starts execution, breaking the trace capture.

Issue

Unable to locate the Intel® CSME trace extension file.

Solution

Intel® CSME trace extension file (also called trace decoder) is shipped inside the Intel® CSME Kit which also contains the Intel® CSME firmware running on your target. Contact your Intel representative for additional help.

Issue

Platform does not halt at platform-boot-stall.

Solutions

Make sure that the power stays disconnected for an extended period of time (> 30 sec) and try again.

Issue

Intel® CSME messages are not decoded correctly (showing hexadecimal values).

Solution

No correct Intel® CSME trace decoder extension is installed. See the section *Create and Configure a System Trace Project* above.

Issue

Target does not halt on platform boot.

Solution

Try Intel CSE boot stall instead:

```
itp.stalls.cseboot=1
```

This captures traces after the Intel CSE ROM phase (slightly after platform boot).

For additional help, refer to the platform documentation, primary [Troubleshooting](#) chapter, or contact your Intel representative. Priority support is available for Intel System Debugger NDA users through [Online Service Center](#) or [Intel® Premier Support](#).

Target Indicator

Intel(R) System Debugger Target Indicator is a standalone cross-platform tool that indicates the status of a debug connection to a target platform. It indicates the connection status of USB-hosted Intel(R) Direct Connect Interface (Intel(R) DCI) connections ("DbC" connections) and provides availability details of Intel(R) Silicon View Technology (Intel(R) SVT) Closed Chassis Adapter, Intel(R) In-Target Probe (Intel(R) ITP) - XDP3, and Lauterbach* Adapter.

Launching the Target Indicator

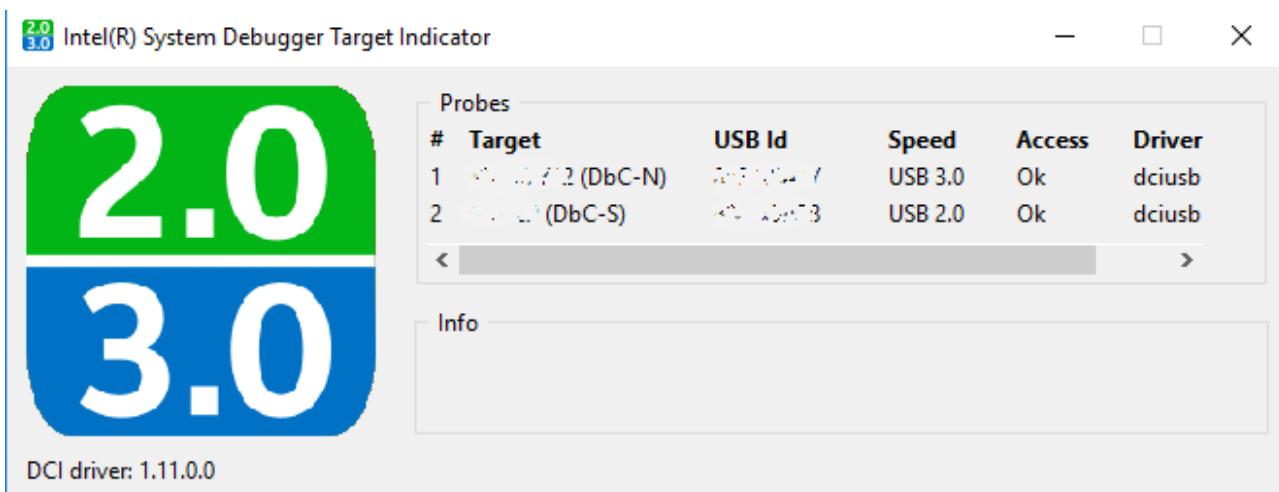
To start the tool, click the desktop shortcut "Intel(R) System Debugger - Target Indicator".

Alternatively, execute the binary located at

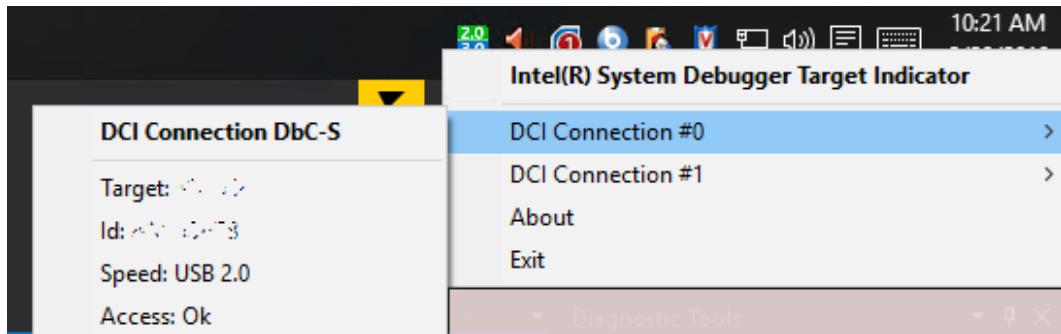
```
<install_dir>\system_debugger_<version>\target_indicator\bin\TargetIndicator.exe .
```

A tray icon appears in the task bar indicating the state of the debug target connection. The icon changes depending on the detected debug target connections to easily indicate the presence of a DTS and the debug resources available. For more information on the tray icon images, see [Target Indicator States](#).

To display additional information for detected target connections, double-click on the tray icon. The following dialog box is displayed:



Alternatively, right-click the tray icon and hover over the desired connection in the pop-up menu to display the information for each connection separately. A separate sub-menu is created for each connection:



For each target connection, the following basic information is provided:

Target

The target platform name and the location of the debug resource (for DbC connections).

USB ID

The USB connection vendor ID (VID) and the product ID (PID) that the target exposes. The target is detected based on this ID pair.

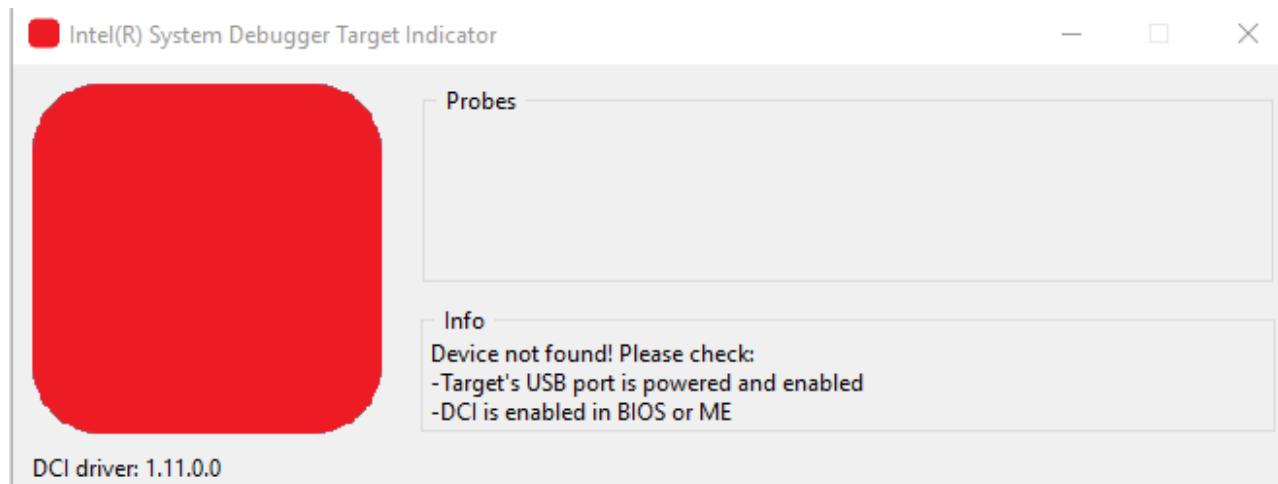
Speed

The connection speed of the hosting USB interface.

Access

Shows the availability of debug resources. For UNIX-based OS, it verifies if you have permissions for USB endpoints of the debug connection. For Windows* OS, it additionally verifies if the correct driver is selected for the USB connection (the Intel(R) Direct Connect Interface (Intel(R) DCI) Driver).

If no target is currently connected, the pop-up menu is empty and the dialog box looks as following:



Target Indicator States

This section lists all tray icon images that reflect particular target connection states.

Single Probe States

 **1.1** USB 1.1 DbC Connection

 **2.0** USB 2.0 DbC Connection

 **3.0** USB 3.0 DbC Connection

 **3.1** USB 3.1 DbC Connection

 **CCA** Intel(R) Silicon View Technology (Intel(R) SVT) Closed Chassis Adapter is connected

 **XDP** Intel(R) In-Target Probe (Intel(R) ITP) - XDP3 is connected.

 **LTB** Lauterbach* probe is connected.

 Unknown state. See the following table for possible root causes and suggested solutions to fix the state.

Cause
For Windows* OS only: the Intel(R) Direct Connect Interface (Intel(R) DCI) driver is not installed or the wrong version is installed.
For UNIX-based OS only: you do not have access to the USB endpoints of the connected device.
Unknown combination of devices is attached. The usecase is unclear.
Intel DCI debugging is not enabled in BIOS settings.

Multiple Probe States

For multiple connections, the tray icon displays an accumulated state. See examples below:

3.0 USB 3.0 and USB 1.1 DbC connections exist simultaneously.
1.1

X **2.0** One connection is in an error state.

ΨΨ Too many targets are connected and the use case is unclear.

Lauterbach TRACE32* Software Support

This document describes how to install the Lauterbach* TRACE32 software package and integrate it to the Intel® System Debugger. Follow these instructions only if you use Lauterbach probes, the Lauterbach CombiProbe MIPI60-Cv2 (for Intel® platforms) or the Lauterbach TRACE32 QuadProbe on pre-launch NDA products.

To install the Lauterbach TRACE32* software, complete the following steps:

Prerequisites

To obtain Lauterbach TRACE32, you must complete the following pre-work steps:

1. Sign Multi-Party, Non-disclosure Agreement

Your company needs to sign a Multi-Party, Non-disclosure Agreement with Intel and Lauterbach GmbH. This permits Lauterbach to share the Trace32 NDA software with your company.

To sign this agreement, complete the form below and email it to debugtoolssupport@intel.com:

Your Company's Legal Name:

The name of the requestor:

The email of the requestor:

The name of signer:

The title of signer:

The email address of signer:

If available, the Intel Confidential Non-Disclosure Agreement number:

List the names **and** email addresses of those who should receive a completed copy.

An Intel agent will receive it, assemble the necessary document(s), and send them to the person you designate for e-signature. Copies of the completed document(s) will be delivered to a person at Lauterbach GmbH and the people you designate in the form. To expedite, please discuss this with the person signing before sending this form.

2. Create an account at <https://www.lauterbach.com>.

3. Lauterbach GmbH will ensure that each account request has a proper Multi-Party NDA agreement in place between Intel Corporation, Lauterbach GmbH, and the account member's parent corporation prior to authorizing NDA release access.

This extra step may take a day or two. If after a few days access is not automatically generated, you can send a support request to either Intel or Lauterbach for assistance.

4. After authorization is granted, you will receive a confirmation email and can login to the portal for access to the NDA update at <https://login.lauterbach.com/>.

Determine the Required Version of the Lauterbach TRACE32* Software

Though lower versions of the Lauterbach* TRACE32 debugger are likely to operate correctly with the latest Intel® System Debugger, Intel does not guarantee that all features and bug fixes are incorporated if you continue using an older Lauterbach TRACE32 release while updating your Intel System Debugger installation.

You can find the recommended Lauterbach TRACE32 release to be used with the Intel® System Debugger in the following locations:

- In the description field for the latest Intel System Debugger release
- Bill of Materials included into the Intel System Debugger package, which contains all component versions including the expected version of the Lauterbach TRACE32 debugger on your host machine.

Within the Bill of Materials, the following file numbering is used:

`Trace32.LTB.2.<intel_system_debugger_build>.<package_number>.<lauterbach_build>.msi`

where `<intel_system_debugger_build>` is a four-digit number indicating the year and the workweek when the release package for the Intel® System Debugger was built (for example, `1929`) and `<lauterbach_build>` refers to the Lauterbach TRACE32 software build number. The build number will be common between the Intel System Debugger release notes and the download found on the Lauterbach download portal.

The Lauterbach TRACE32 releases have file naming such as this example:

`trace32_n_<year>_<month>_<build_number>_intel_complete_<timestamp>.zip`, where `<build_number>` is equal to `<lauterbach_build>` above.

Download the Lauterbach TRACE32 Debugger

If you are a Lauterbach user, you need to create an account on the [Lauterbach TRACE32 customer portal](#). Lauterbach GmbH will confirm if your account has a proper Multi-Party NDA agreement between Intel Corporation, Lauterbach GmbH, and your parent corporation.

After registration, you will receive a confirmation email. Then you can login to the [portal](#) and download the highest version of the Lauterbach* TRACE32 debugger.

Verify the Download Version

Confirm the version available for download on the portal matches the build version as specified in the Intel® System Debugger release notes. Contact Lauterbach support in case of issues in identifying the download file. You can find the contact information on the Lauterbach website.

Install the Lauterbach TRACE32 Debugger

! Important

The Lauterbach* TRACE32 package from the Lauterbach portal is not a complete copy of the Lauterbach TRACE32 debugger available on the latest DVD release. The full Lauterbach TRACE32 software from the original DVD package must already be installed on the host system. After downloading the Lauterbach TRACE32 update package, extract the contents of the .zip file into the existing installation folder.

You should install the Lauterbach TRACE32 software prior to installing the Intel® System Debugger. This allows the Intel® System Debugger to install script add-ons into the existing Lauterbach TRACE32 installation folders. If no previous version of the Lauterbach TRACE32 debugger is detected, the Intel® System Debugger automatically installs these items into the default path (`C:\T32`).

The necessary run-time pointers between the Intel System Debugger and the Lauterbach TRACE32 debugger use a system environment variable and therefore do not require any specific installation order between the Intel® Platform Validation Toolkit (Intel® PVT) and the Lauterbach TRACE32 debugger (see below for multiple Lauterbach TRACE32 installations).

Depending on what installations are present on your system, follow the corresponding procedure:

- If the Lauterbach TRACE32 debugger is already installed but the Intel® System Debugger is not installed:

Install the Intel® System Debugger and the Lauterbach TRACE32 update package in any order.

- If the Lauterbach TRACE32 debugger is not installed but the Intel® System Debugger is already installed:

Manually copy the script extensions into the eventual Lauterbach TRACE32 folder. The Intel System Debugger will place the add-ons into the `C:\T32` folder. Either install the Lauterbach TRACE32 debugger into the `C:\T32` folder or after Lauterbach TRACE32 has been installed to an alternate location, copy and paste all of the contents of the `C:\T32` folder into the Lauterbach TRACE32 top level folder.

- If multiple Lauterbach TRACE32 installations exist:

Currently, no support is offered for the resident installation of multiple versions of the Lauterbach* TRACE32 debugger. If you need to keep different versions of Lauterbach TRACE32 and do not plan to use it with the Intel® System Debugger, you can copy the installation folder for the alternative installation to a folder with a different name. The Lauterbach TRACE32 software is not sensitive to the folder name it resides in. Make sure the system environment variable that the Intel System Debugger relies upon is set to the last Lauterbach TRACE32 installation folder location, therefore this version should be installed last.

Verify Lauterbach TRACE32 Integration

User Smart Start to configure and launch the Python* CLI shell. The Lauterbach TRACE32 debugger should start automatically. From the Lauterbach TRACE32 GUI, select **Help > About** to confirm the executing version of Lauterbach TRACE32 has the correct build number.

Debugging Zephyr* Applications with Intel® System Debugger

This document describes steps to setup your Eclipse* IDE and debug Zephyr* applications with Intel® System Debugger under Non-Disclosure Agreement (NDA).

Prerequisites

To build and debug a Zephyr* project, make sure you fulfill the following requirements:

- Linux* OS (Ubuntu* 20.04 LTS).

This is the only OS distribution fully supported now.

- Zephyr* project tree: Intel® Programmable Services Engine (Intel® PSE) SDK and build dependencies
- Intel® System Debugger NDA 2XXX. See [installation instructions](#).
- Elkhart Lake target: Integrated Firmware Image (IFWI) with orange token enabled and without Intel® PSE firmware stitched.

Check the required BIOS configurations below.

- Eclipse* Plug-in for Zephyr* Project. See [installation instructions](#).

Required BIOS Configurations

To enable debugging, go to **Intel Advanced Menu > Debug Settings** and enable **Platform Debug Consent** as follows:

- If you plan to use Intel® Silicon View Technology (Intel® SVT) Debug Class (DbC) USB cable, select **Enabled (USB2 DbC)** or **Enabled (USB3 DbC)**.

- If you plan to use Intel® Direct Connect Interface (Intel® DCI) Out of Band (OOB) via Intel® SVT Closed Chassis Adapter (CCA), select **Enabled (Intel DCI OOB)**.

To change Intel® Programmable Services Engine (Intel® PSE) settings, go to **Intel Advanced Menu > PCH-IO Configuration > PSE Configuration** and do the following:

- Set **PSE Debug (JTAG/SWD) Enable** to **Selected**.
- Set **PSE JTAG/SWD PIN MUX** to **Disabled**.

Configure Eclipse* IDE

To debug Zephyr* applications, you need to install the Intel® System Debugger NDA, install the Eclipse* plug-in and build a Zephyr project. See the instructions below.

Install Intel® System Debugger

1. Download the latest Intel® System Debugger NDA version from [Intel® Products](#).
2. Run the installer as follows:

```
sudo ./l_sys_dbg_ndp_p_<year>.<version>.0.<build>_offline.sh
```

where `<version>` is a digital number of Intel System Debugger NDA version (for example, 2038), `<build>` is the build number and `<year>` is the period when this version was released. You can tell this by the first two digits of the version number (for example, version 2038 was released in 2020 and version 2106 in 2021).

3. Launch Intel® System Debugger with Zephyr* environment by executing `start_zephyr_eclipse.sh` located at `/opt/intel/oneapi/system_debugger/<version>-nda/system_debug/gdbserverproxy/`.

If you have a different Intel® Programmable Services Engine (Intel® PSE) installation path than the default one (`~/pse_sdk/code/pse-dev-code-base/`), introduce the path to `pse-dev-code-base` as an argument:

```
./start_zephyr_eclipse.sh ~/<path>/pse_sdk/code/pse-dev-code-base/
```

First-time launch can take some time, so do not close the terminal where installation logs are displayed.

See also

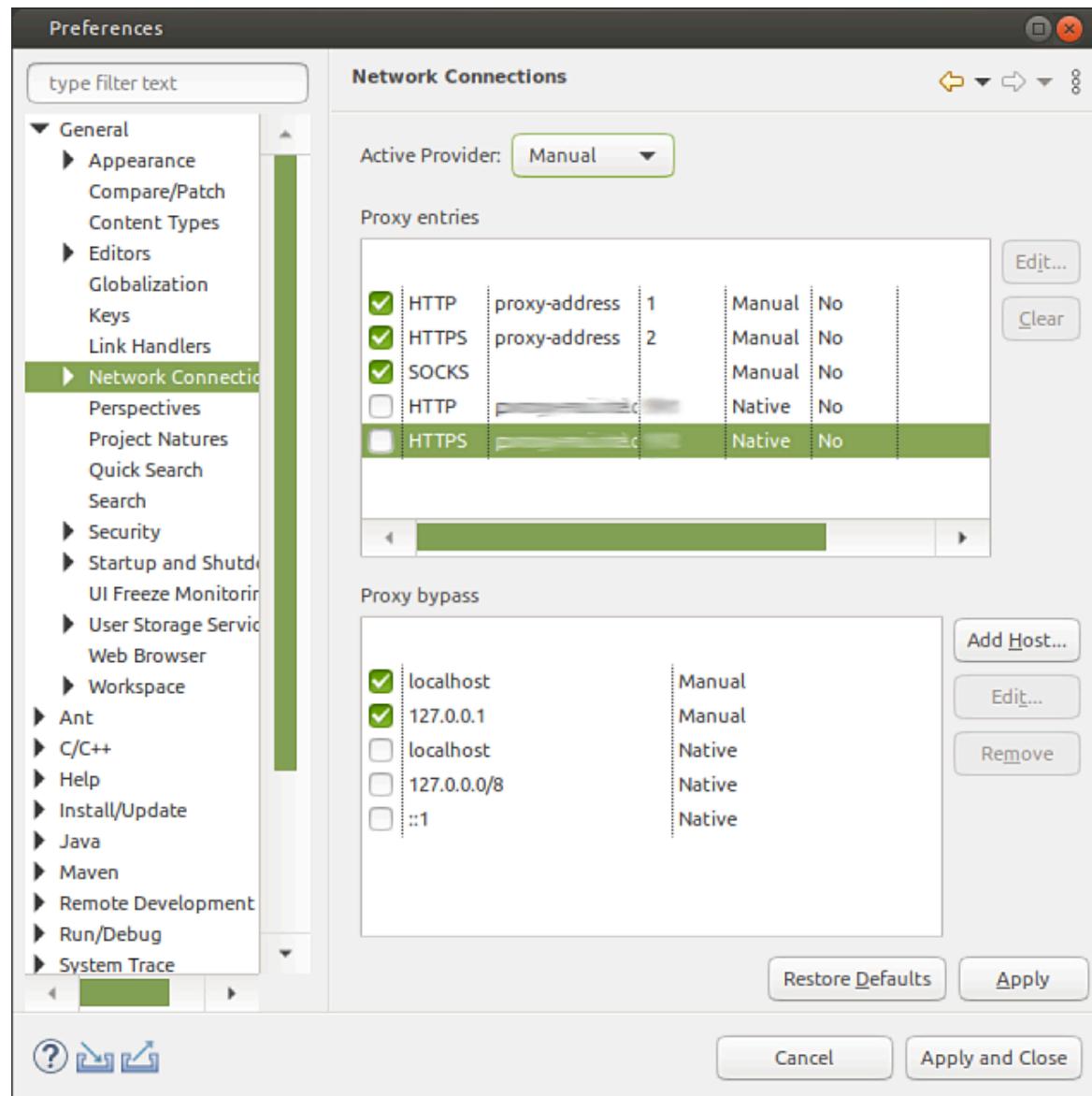
[West \(Zephyr's meta-tool\)](#)

Install Eclipse* Plug-in for Zephyr* Project

Follow the steps below to install the plugin:

1. Check your proxy settings:

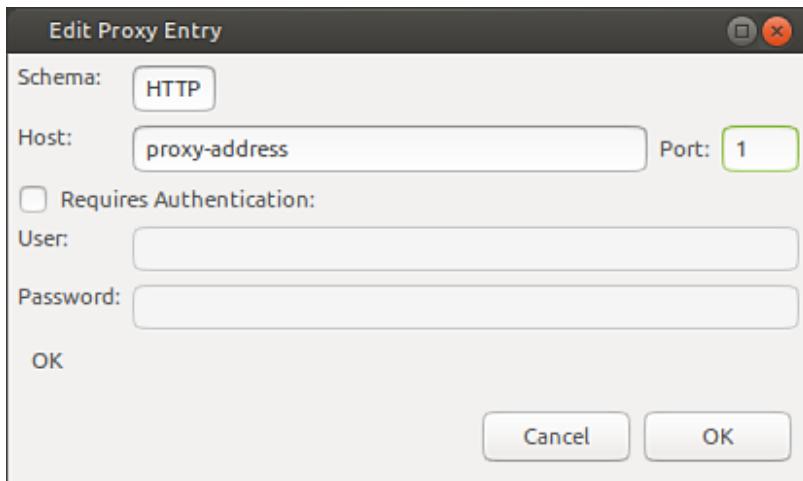
1. In Eclipse* IDE, open Window > Preferences.
2. Under General node, select Network Connection.
3. In Active Provider field, select Manual (or Native if you have proxy connection configured on your system).



4. Add proxy data as follows:

- Schema: `HTTP`
- Host: `<your-proxy-address>`
- Port: `<your-port>`

If authentication is required, provide your user credentials.



5. Click **OK** to save the proxy settings.

2. In Eclipse IDE, open **Help > Install New Software**.

3. Click **Add** to add a new update site:

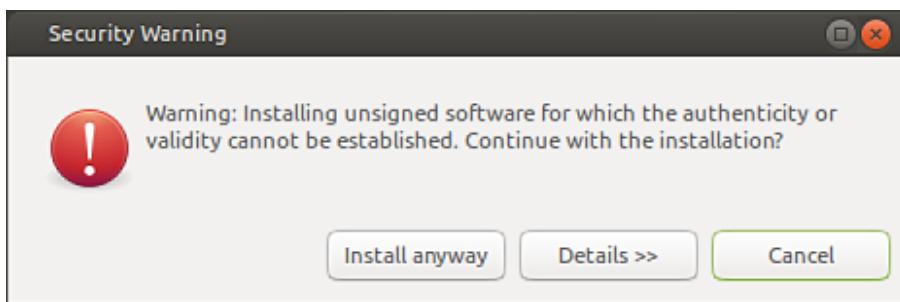
- Name: `zephyr-eclipse`
- URL: `https://builds.zephyrproject.org/eclipse-plugin/`

4. Select the newly added update site.

5. Select **Zephyr Project** and **Zephyr Project Development Support**.

6. Follow the instructions on the wizards to install the plugin.

7. If the following security warning appears, select **Install anyway**.



When installation is completed, restart Intel® System Debugger NDA.

! See also

[Eclipse* Plug-in for Zephyr* Project: repository at GitHub* software development platform](#)

Set up Intel® System Debugger Workspace

To build the projects, you must configure the workspace by adding some environment variables.

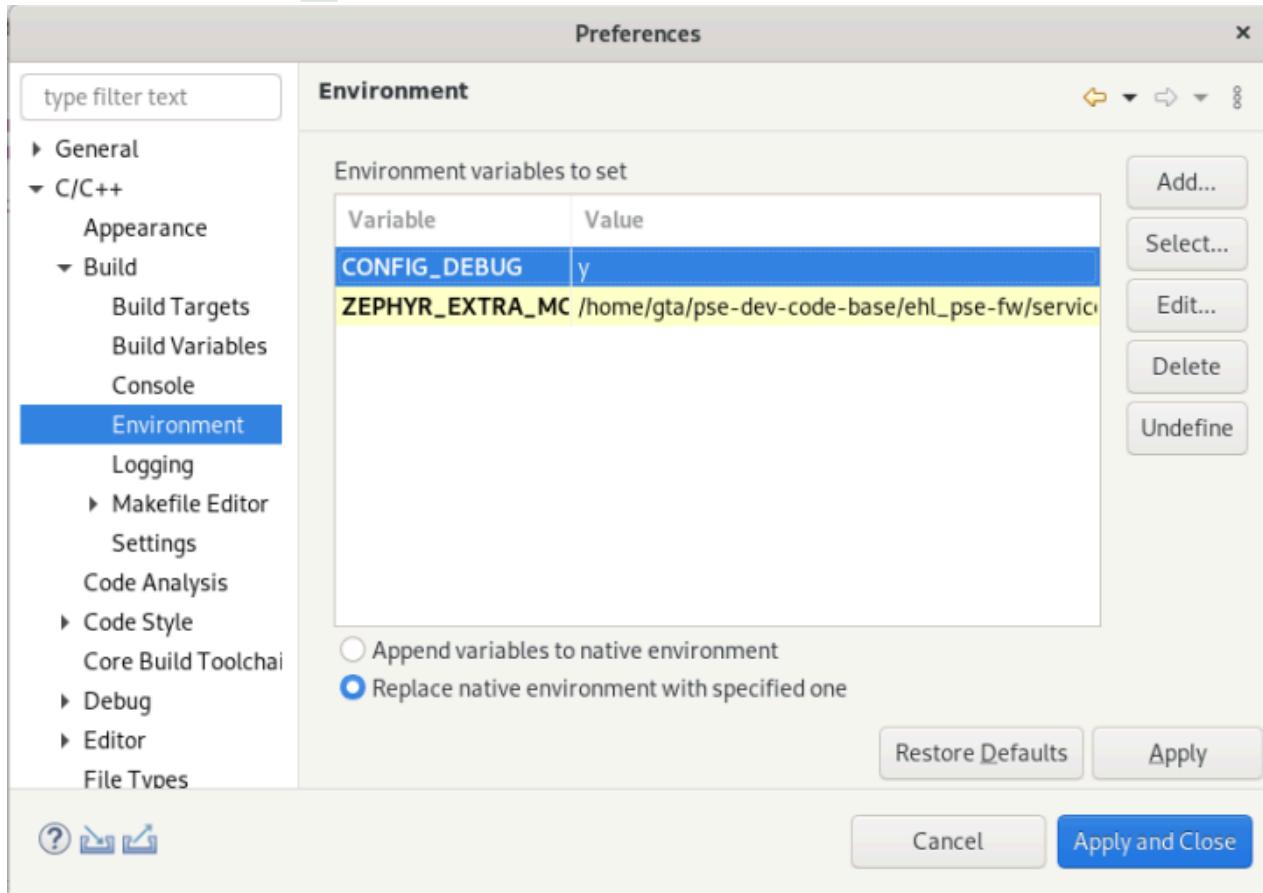
1. In Eclipse* IDE, open **Window > Preferences**.

2. Under C/C++ node, unfold Build and select Environment.

3. Select Replace native environment with specified one.

4. Add the following variables:

- ZEPHYR_EXTRA_MODULES <path_to_pse>/ehl_pse-fw/services
- CONFIG_DEBUG y



5. Click Apply and close.

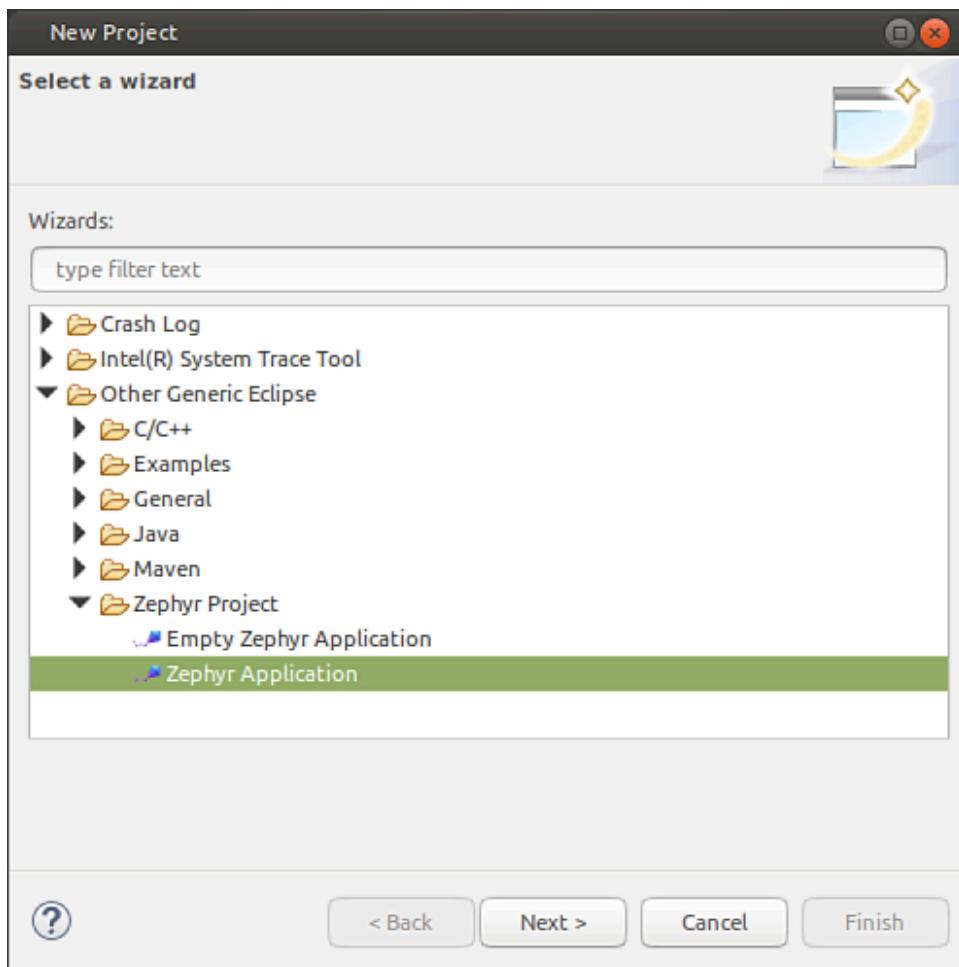
Verify that the path introduced in `ZEPHYR_EXTRA_MODULES` is a valid directory. If not, the build will fail.

Create a Project

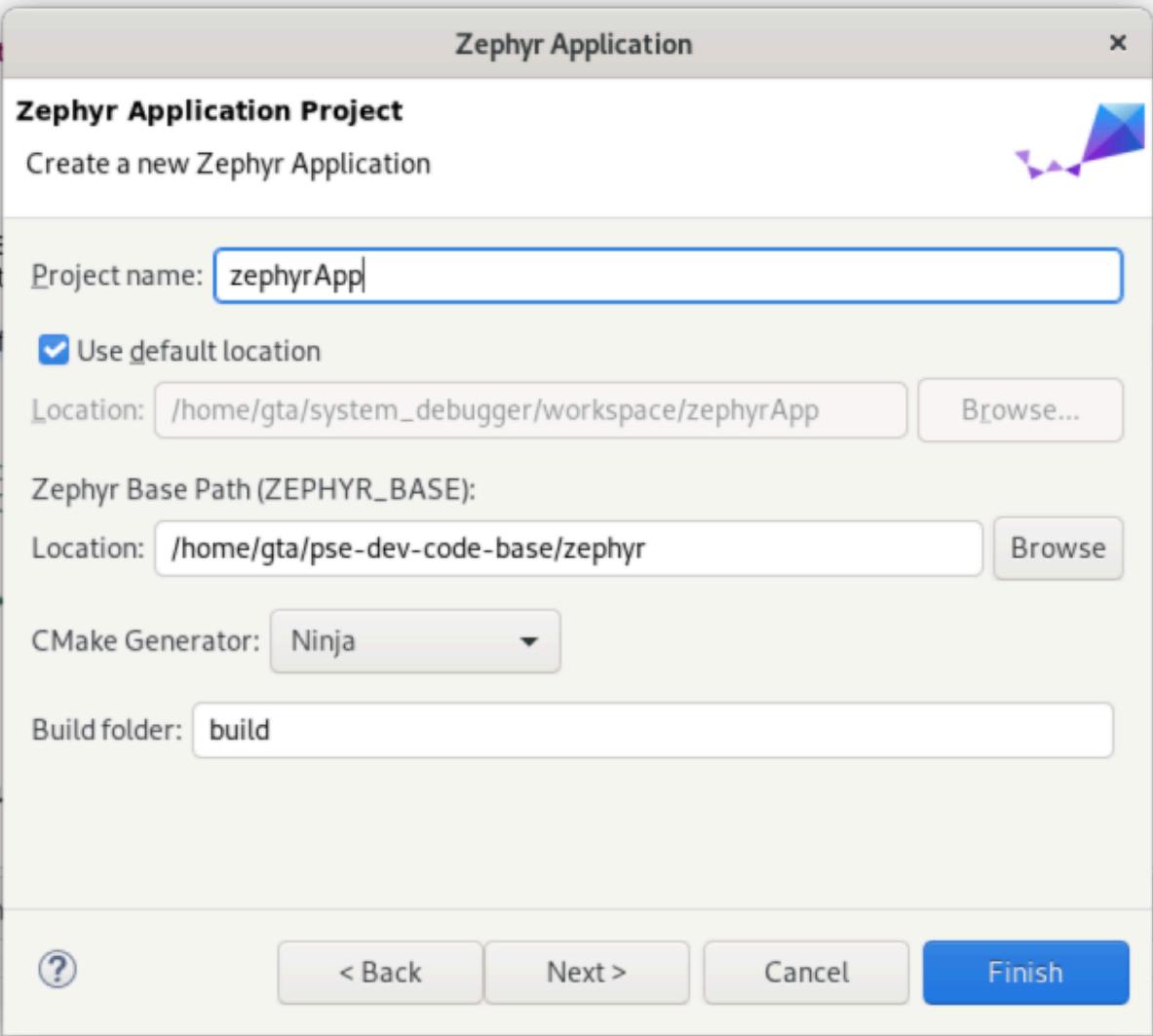
1. In Project Explorer view, click Create a project.

Alternatively, go to File > New > Project.

2. In the Select wizard dialog box, navigate to Other Generic Eclipse > Zephyr Project and select Zephyr Application and click Next.



3. Type in your project name.
4. Add the Zephyr* Base Path (ZEPHYR_BASE): <path-to-PSE>/pse-dev-code-base, where <path-to-PSE> is the installation location of Intel® Programmable Services Engine SDK (Intel® PSE SDK).



Click **Next**.

5. Specify Zephyr SDK Install Path (ZEPHYR_SDK_INSTALL_DIR).

Zephyr Application



Zephyr Application Project - Toolchain Selection

Specify the Toolchain to Build this Application



Toolchain Variant: Zephyr SDK



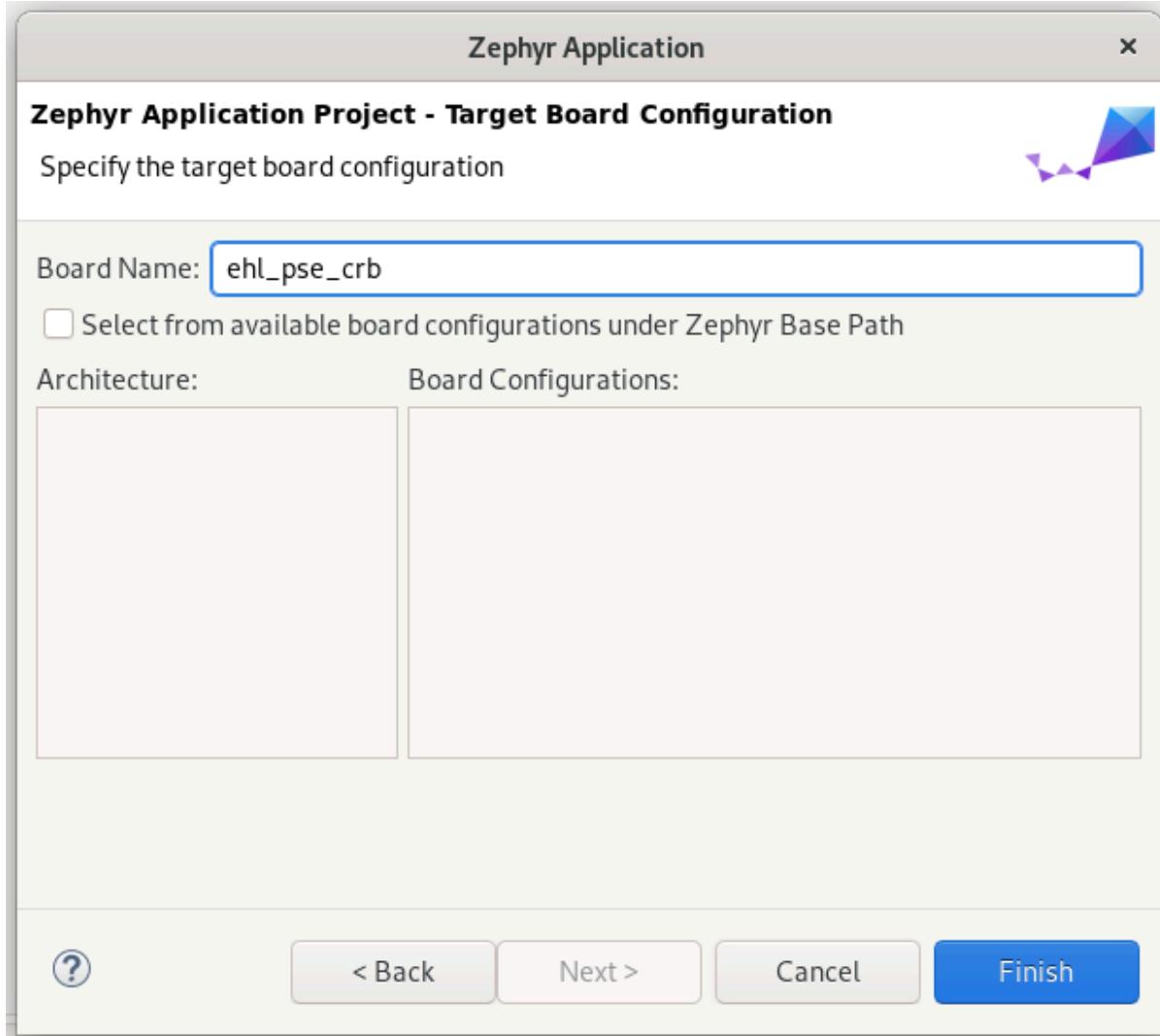
ZEPHYR_TOOLCHAIN_VARIANT =

Zephyr SDK Install Path (ZEPHYR_SDK_INSTALL_DIR):



Click **Next**.

6. Type the desired board: **eh1_pse_crb**.



7. Click **Finish**.

Now the project is created.

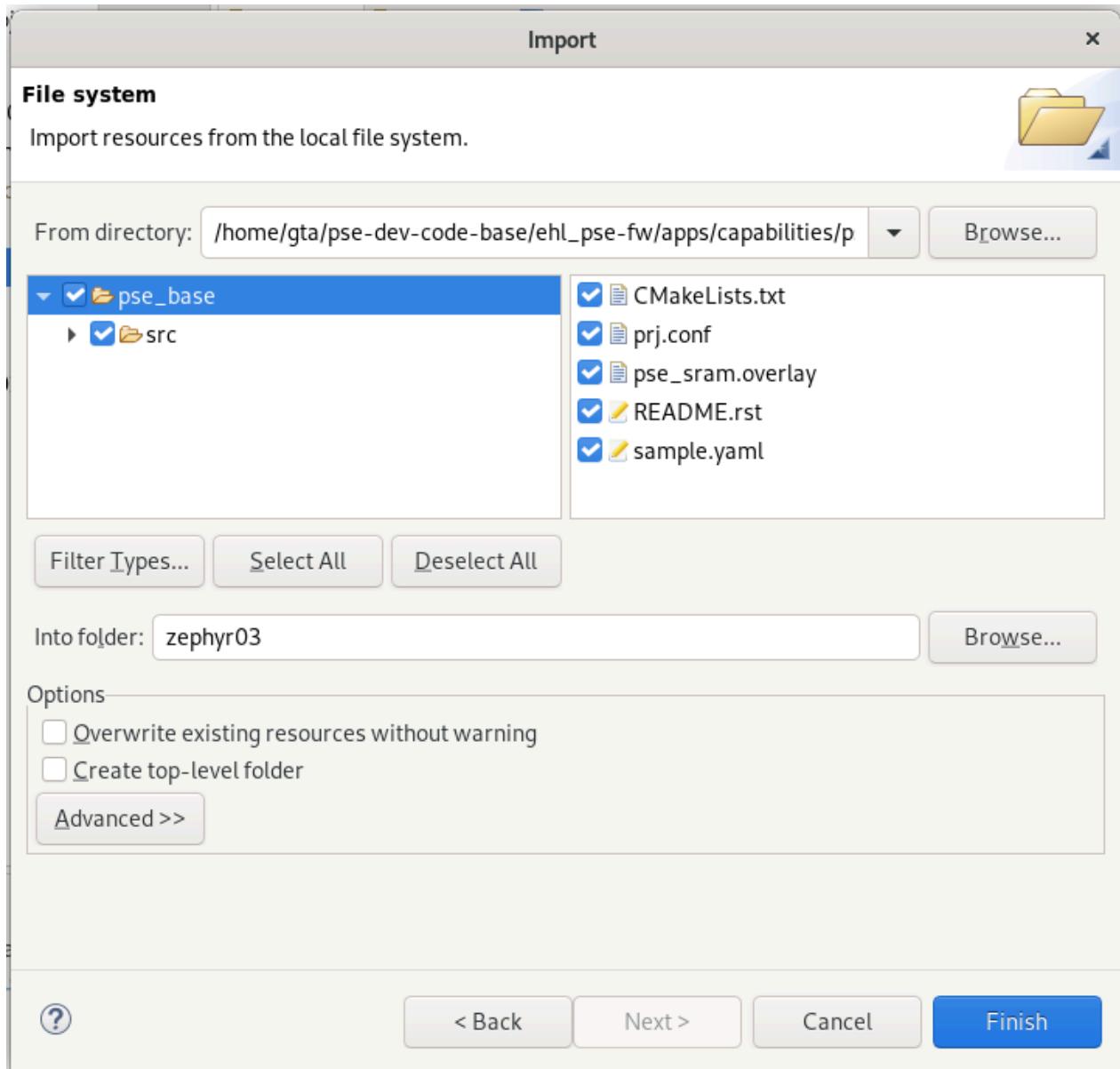
Importing a Project

You cannot directly import a Zephyr project or sample created outside Eclipse IDE. Alternatively, you can import source files into an empty (new or existing clean) Zephyr application.

To import source files into an empty project:

1. Create a project by following instructions [here](#) but in step 2 select **Empty Zephyr Application**.
2. Right-click the project in Eclipse IDE and select **Import**.
3. In the opened dialog, open **General** node, select **File System**, and click **Next**.
4. In the wizard, click **Browse** and choose the files to import.

Select the project from <path-to-PSE>/pse-dev-code-base/ehl_pse-fw/apps/ that was built on Intel® Programmable Services Engine (Intel®) PSE.



5. Click **Finish**.

Now source files are displayed in the Project Explorer when you select the project pointing to Zephyr SDK and Zephyr base during project creation.

Build the Project

Build in Eclipse* IDE

Locate the project in the **Project Explorer** view, right-click it, and select **Build project**.

Build in Intel® Programmable Services Engine (Intel® PSE)

1. Introduce the absolute path to the application in the build properties so that it points to the proper Zephyr* application.

2. To build the project, locate it in the **Project Explorer** view, right-click it, and select **Build project**.
3. In the **Console** view, check that the project is built with Python* interpreter and West tool from the virtual environment created at `~/Intel/gdbserverproxy/2020.2030/200`.

```
CDT Build Console [test1]
.... Generating CMake files for board intel_ehl_crb in build
/usr/bin/cmake -DBOARD=intel_ehl_crb -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
Zephyr version: 1.14.2
-- Selected BOARD intel_ehl_crb
-- Found west: ./Intel/gdbserverproxy/2020.2030/200/venv/bin/west (found suitable version "0.7.2", minimum required is "0.5.6")
-- Loading /zephyr/pse-dev-code-base/modules/rtos/pse_zephyr/boards/arm/intel_ehl_crb/intel_ehl_crb.dts as base
-- Overlaying /zephyr/pse-dev-code-base/modules/rtos/pse_zephyr/dts/common/common.dts
extract_dts_includes.py: WARNING: id field set in 'QEP Base Structure', should be removed.
Parsing Kconfig tree in /zephyr/pse-dev-code-base/modules/rtos/pse_zephyr/Kconfig
Loading /system_studio/workspace/test1/build/zephyr/.config as base
Configuration written to '/system_studio/workspace/test1/build/zephyr/.config'
-- Cache files will be written to: /users/.../.cache/zephyr
-- OOB being built as Library
-- Configuring done
-- Generating done
-- Build files have been written to: /system_studio/workspace/test1/build
.... Done generating CMake files for board intel_ehl_crb in build
.... Building for board intel_ehl_crb in build
/usr/bin/ninja
[1/114] cd /zephyr/pse-dev-code-base/services/fw_version && /usr/bin/cmake -D PROJECT_BINARY_DIR=/system_st
[2/114] Generating misc/generated/syscalls.json
```

Debug Zephyr* Application

Before you begin, make sure you have met the following requirements:

- Prerequisites
- Eclipse plug-in for Zephyr* project installed.
- Zephyr* Application project

When the project is successfully built, you can proceed with target setup and debugger launch.

Prepare the Target

If you want to debug the application that is already stitched in the Intel® Programmable Services Engine (Intel® PSE) firmware or a prebuilt binary, you need to point to that binary instead of the Zephyr* Application. Commonly, you can find this binary at `<path-to-PSE>/pse-dev-code-base/ehl_pse-fw/build/zephyr/zephyr.elf`.

Note

If you want to debug another application, clean the `pse` project and build it again pointing to another application.

Launch the Debugger

1. Launch the GDB* debug server located at `/opt/intel/oneapi/system_debugger/<version>-nda/system_debug/gdbserverproxy/` in a new terminal.

```
./start_gdbserverproxy.sh
```

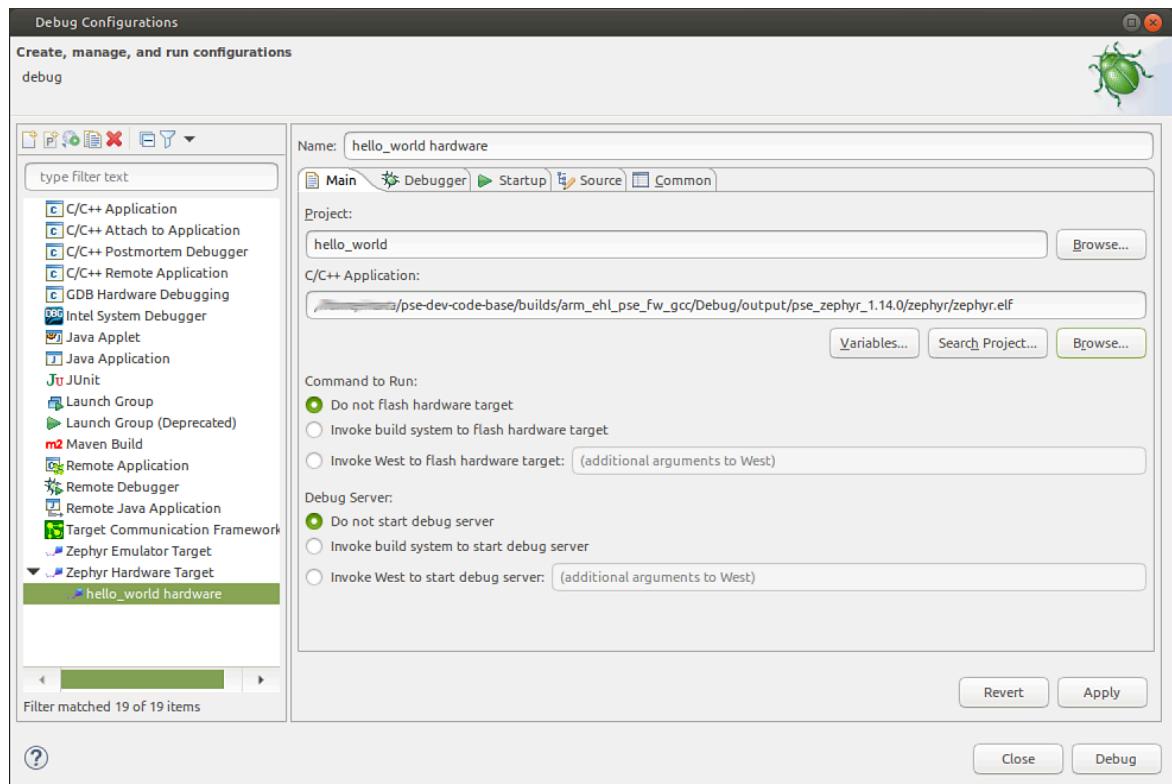
2. Ensure that the `gdbserver` successfully detects your target.
3. Launch Intel® System Debugger NDA.
4. In the **Project Explorer** view, right-click on your application project and select  **Debug As > Debug Configurations**.
5. In the opened dialog box, create a new Zephyr* debug configuration by double-clicking **Zephyr Hardware Target**.
6. Open the **Main** tab and apply the following configurations:

1. Specify the name of the project to debug.
2. In the C/C++ Application field, select the binary to debug. By default, the Zephyr* application binary is selected.

If you want to change the binary, click **Browse** and locate the new one (for example:

`<path-to-PSE>/pse-dev-code-base/ehl_pse-fw/build/zephyr/zephyr.elf`).

3. Select *Do not flash hardware target* and *Do not start debug server*.
4. Click **Apply**.



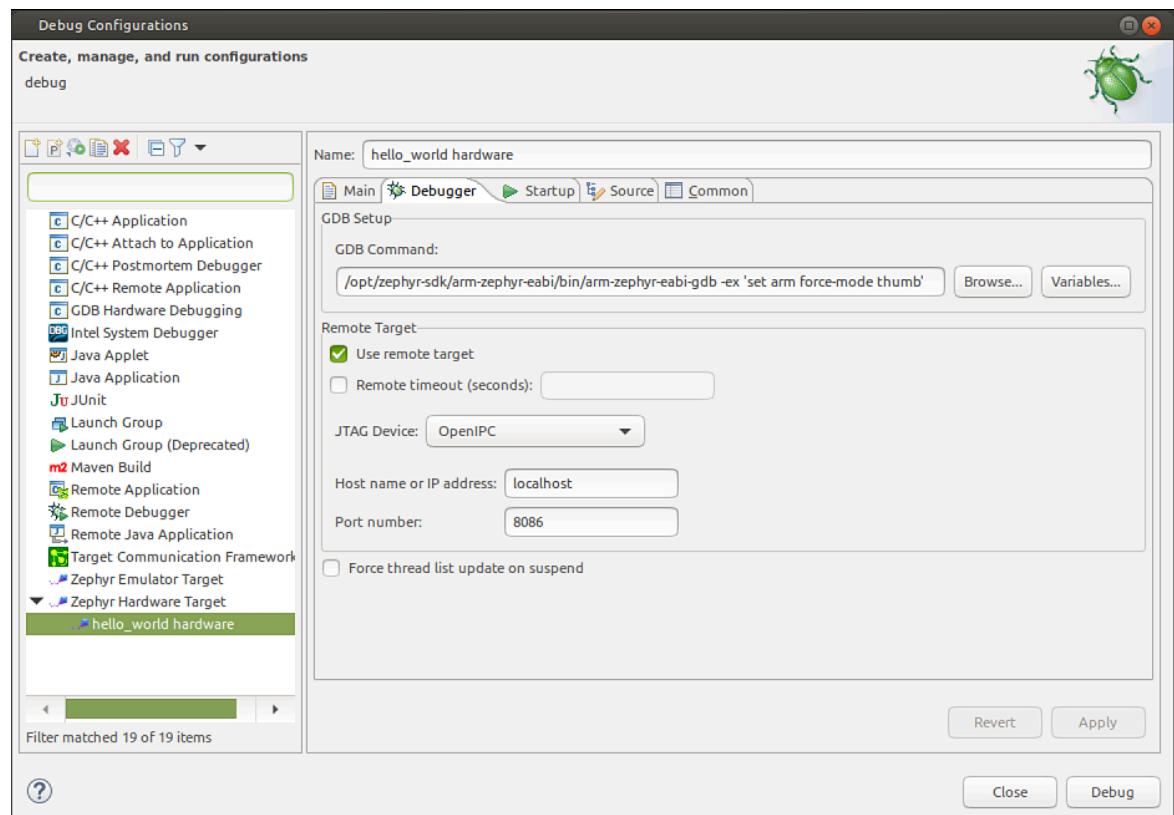
5. Switch to the **Debugger** tab.
6. Ensure that the path in **GDB Command** points to the proper `gdbserver` from Zephyr* SDK. Example:

`<path-to-zephyr-sdk>/arm-zephyr-eabi/bin/arm-zephyr-eabi-gdb`

- Otherwise, click **Browse** and navigate to the proper location.
7. Add the following arguments to the command line: `-ex 'set arm force-mode thumb'`.
 8. In **JTAG Device** drop-down list, select **OpenIPC**. Host name `localhost` and port `8086` must appear automatically.

If OpenIPC is not in the drop-down list, upgrade the plugin.

9. Click **Apply** to save tab changes.



7. Switch to the **Startup** tab:

1. Check **Load image** box and select **Use project binary** option.

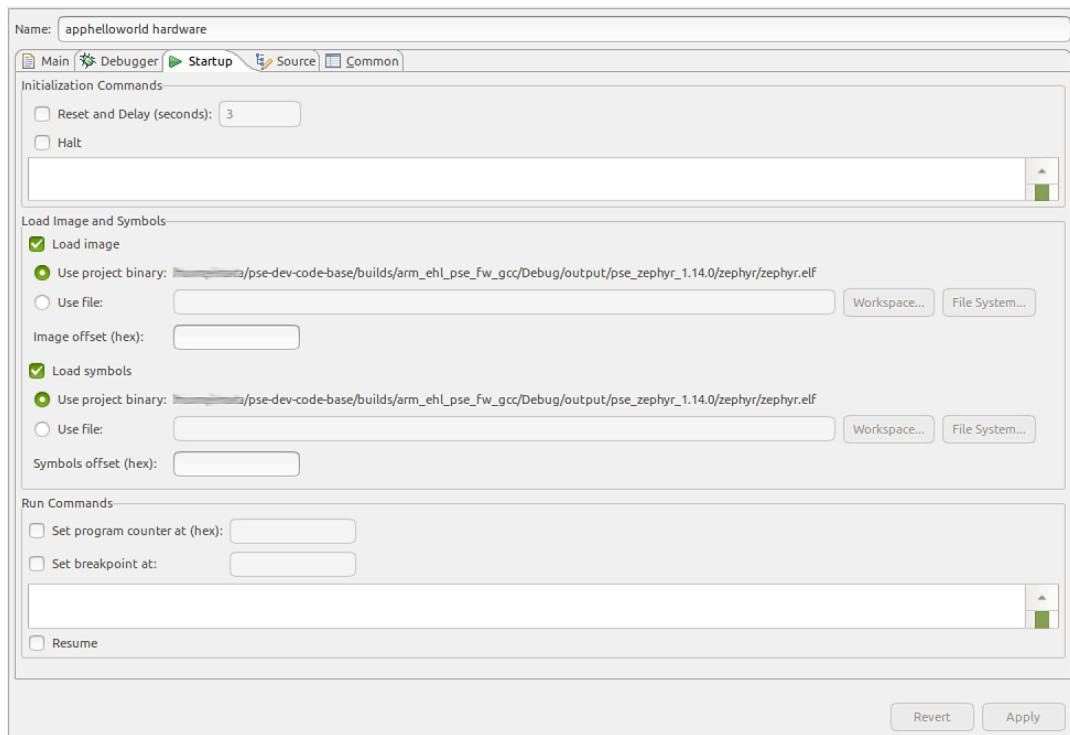
Note

If you want to debug an application that is already stitched to Elkhart Lake, uncheck the **Load Image** box.

2. Depending on the binary configuration, check other boxes as follows:

- If the binary was built with symbols and source code (debug configuration), check the **Load symbols** box.

Select the **Use project binary** option and ensure that it points to the same binary as the one in **Main** tab. If you want to debug the binary from Zephyr* application, it will point to `zephyr/zephyr.elf`. Otherwise, it will be similar to the one under the **Load image** box.



- If the binary was built without symbols and source code (release configuration), leave the **Load symbols** checkbox empty.

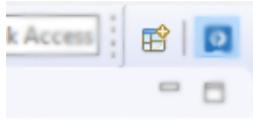
3. Click **Apply** to save tab changes.

8. Click **Debug**.

Now, the target connection is established. In the `gdbserver` terminal, you can see a message exchange that indicates successful connection.

Perform Basic Debugging Tasks

If you are not prompted to the **Debug** perspective automatically, click the **Open Perspective** button on the top right corner of the Eclipse* toolbar:



Select **Debug** and click **Open**.

Load Application

To load an application, you must be in the ROM space address. The target needs to be reset for that. There are some preconditions for that:

https://psefdk.app.intel.com/_zephyr_code_base.html#autotoc_md5.

Execute the **monitor** command as follows:

```
monitor load  
si
```

If the debugger breaks into 0x700... address, you can load the program as follows:

```
load <path-to-binary>/zephyr.elf
```

If the address is still 0x600..., execute the instructions again.

Note: This issue is also present in OpenOCD. For more information, read:

- https://psefdk.app.intel.com/_debug_with_open_o_c_d_know_how.html
- <https://openocd.org/doc/html/GDB-and-OpenOCD.html>

Run Control

To run control from GUI, use **Suspend** and **Resume** buttons in the toolbar.

Alternatively, you can run or resume the application by executing the **c** command from the console. To suspend the execution, use GUI **Suspend** button.

Check Registers

After starting a new debug session, do a step and display registers to ensure that you are on a start function.

In **Debugger console**, execute the following commands:

```
si  
info registers
```

Registers are now displayed in the  Registers view. To open it, go to **Window > Show view > Registers**.

Program counter register must be pointing to the start address close to  `0x6000xxx __start() + 2`. Now you are ready to start debugging.

Step Through Code

To step through the disassembled code (displayed in the  Disassembly view), click the  **Instruction Stepping Mode** button in the Debug view. Now you can use stepping functions:

-  **Step Into** - executes the current line, including any routines, and proceeds to the next statement.
-  **Step Over** - executes the current line, following execution inside a routine.
-  **Step Return** - continues execution to the end of the current routine, then follows execution to the caller of the routine.

Alternatively, execute the  `si` command from the console to step.

Set Breakpoints

From GUI

1. In the Disassembly view, right-click the required address in the source code. Select **Add breakpoint**.
2. Set the type **Regular** or **Hardware**.
3. Set additional parameters:
 - **Enabled** box (checked by default) - uncheck if you want to keep the breakpoint disabled now.
 - **Condition** - a custom condition for hitting the breakpoint.
 - **Ignore count** - the number of times a breakpoint hit must be ignored.
4. Click **Apply and Close**.

From the console

To set a software breakpoint, execute the command:

```
break <filename.extension:line>
```

To set a hardware breakpoint, execute the command:

```
hbreak <filename.extension:line>
```

New breakpoint is displayed in the  Breakpoints view.

See also

[Setting Breakpoints with GDB*](#)

Enable Interrupts

By default, interrupts in Intel® Programmable Services Engine (Intel® PSE) are disabled. You can enable and disable them and also see the status by using these commands in the debugger console:

```
monitor enable_interrupts  
monitor disable_interrupts  
monitor status_interrupts
```

Finish Debug Session

To end debug session, follow the steps below (follow the order as presented):

1. Stop debugging from Intel® System Debugger.

Click the  **Terminate** button in the toolbar.

2. Stop `gdbserverproxy`:

1. Go to the terminal where you launched it from and press **Ctrl+C**.
2. Type `exit()` and press Enter.

Now `gdbserverproxy` will finish successfully.

3. Switch the target off.

Appendix: Set up Intel® Programmable Services Engine in Intel® System Debugger Workspace

It is possible to import Intel® Programmable Services Engine into workspace and build from Eclipse.

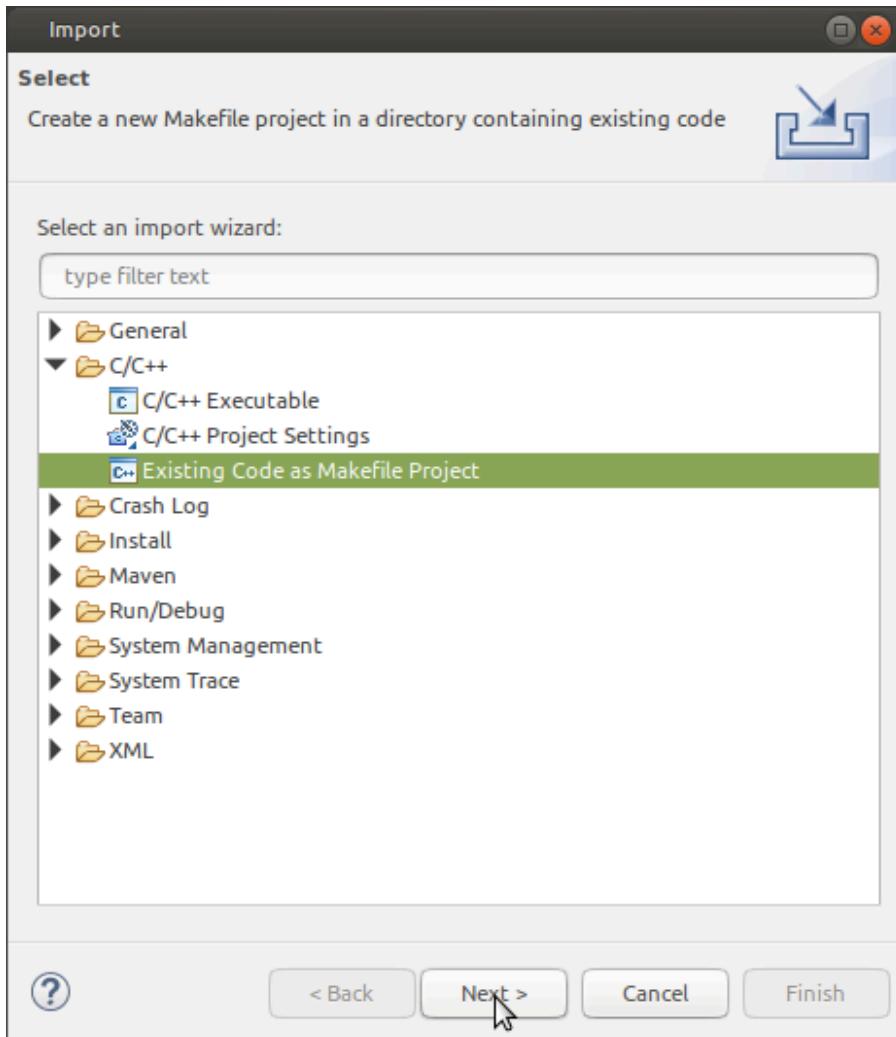
Import pse-dev-code-base Project into Intel® System Debugger Workspace

You can import Intel® Programmable Services Engine (Intel® PSE) into an existing workspace and build it from Intel® System Debugger. You can also build the Zephyr* application in Intel® PSE by pointing to it in `project.json`.

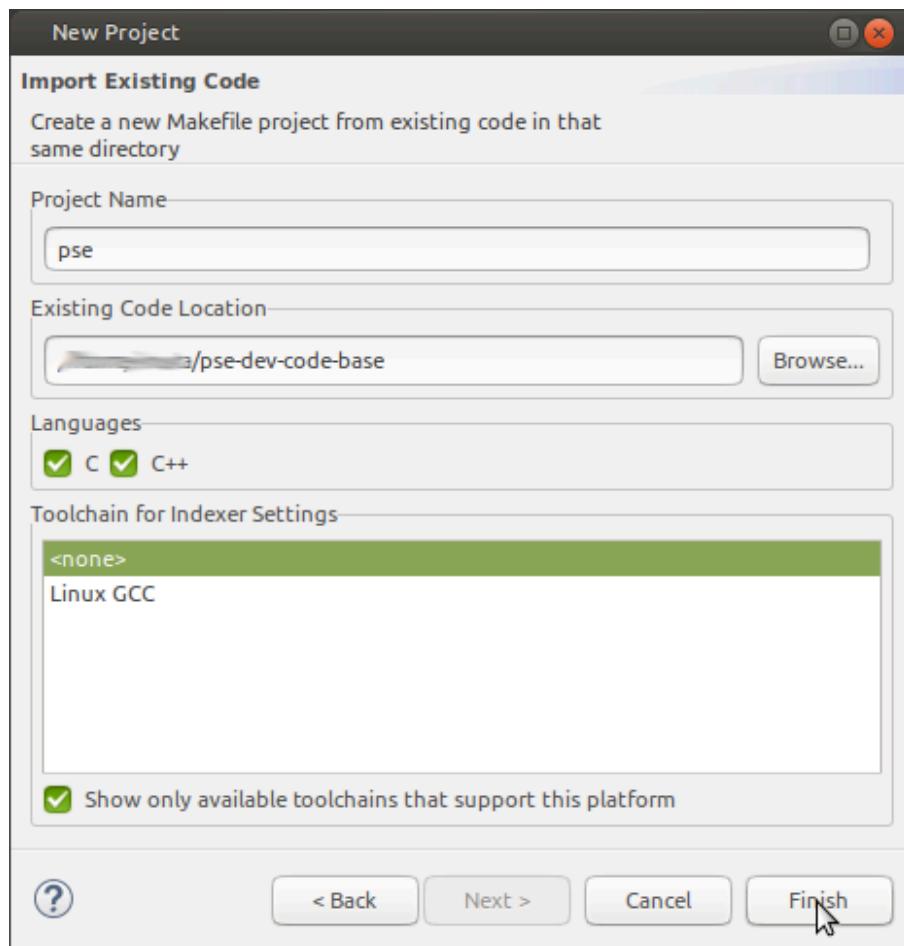
1. In the Project Explorer, select **Import projects**.

A wizard will pop up.

2. In the opened wizard, unfold the C/C++ node and select *Existing Code as Makefile Project*.

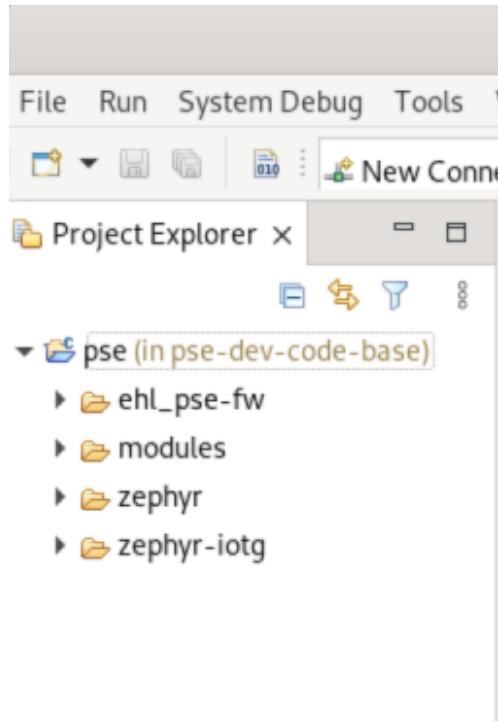


3. Specify the Project Name as `pse` and select the location of `pse-dev-code-base` from your system (browse or type it).



Click **Finish**.

Now the project should appear in the workspace.



Note

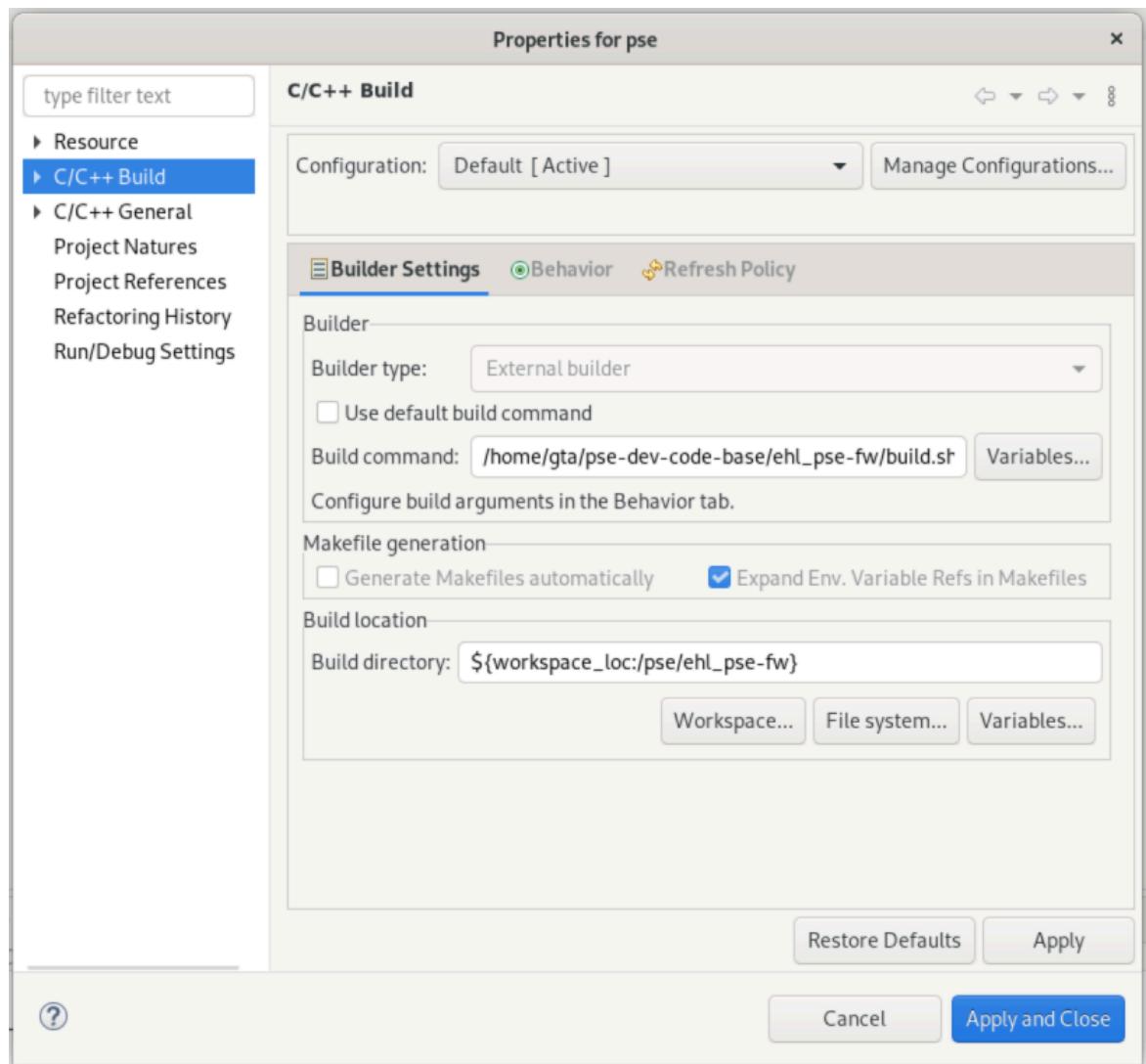
All files and folders from `pse-dev-code-base` location are imported in the workspace as linked resources. This means that they are still located on the primary location, not copied, so any change that you make to the project inside Eclipse*, will also be applied to `pse-dev-code-base` from the local one.

Set up Debug Builder

To build the project using `build.sh` from pse-dev-code-base, you must configure the build for the project.

1. In the Project Explorer, right-click on the `pse` project and select **Open properties**.
2. Go to the **C/C++ Build** property tab and modify the data in the **Builder Settings** tab:

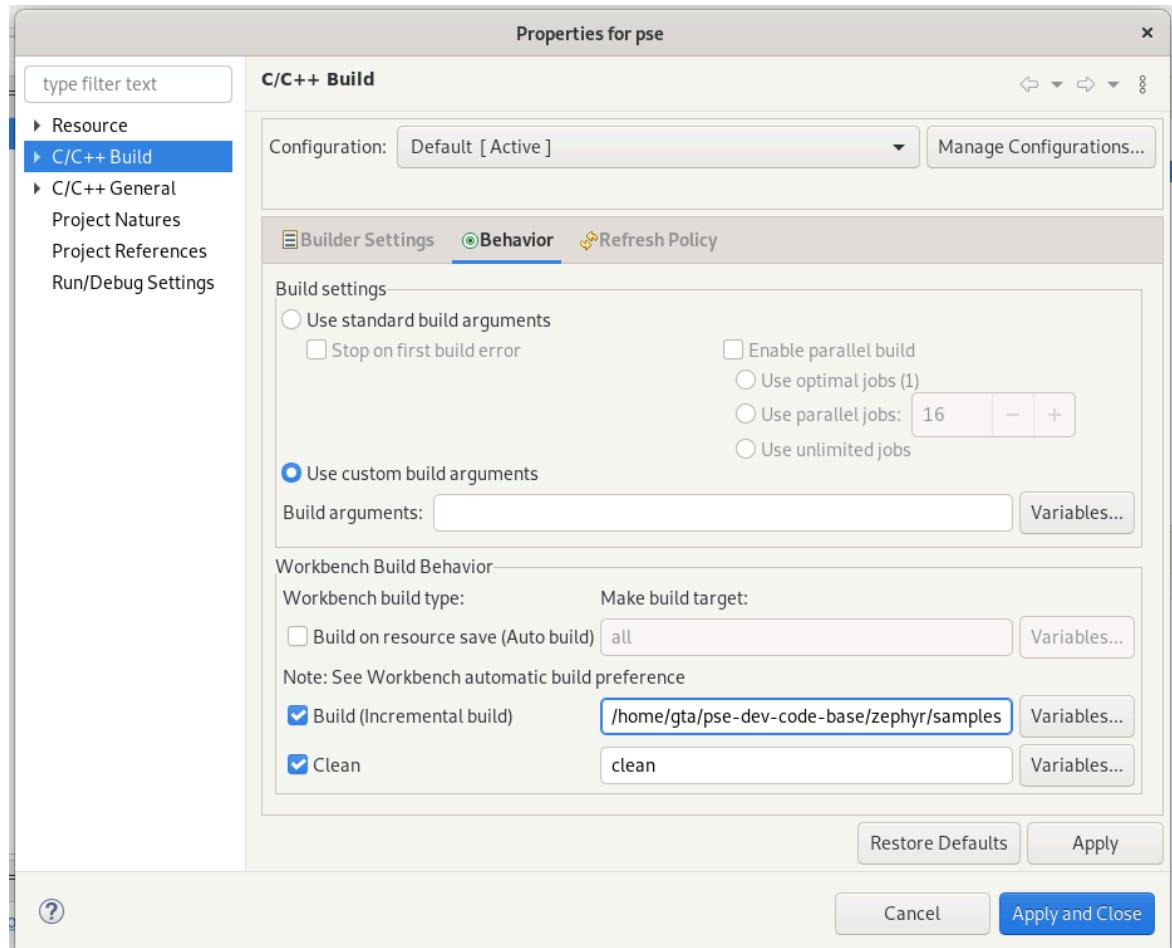
- Unselect `Use default build command`.
- For the Build command field, type absolute path to build.sh file.
- For the Build directory field, click **Workspace** and select `ehl_pse-fw`.



3. Click **Apply** and switch to the **Behavior** tab.

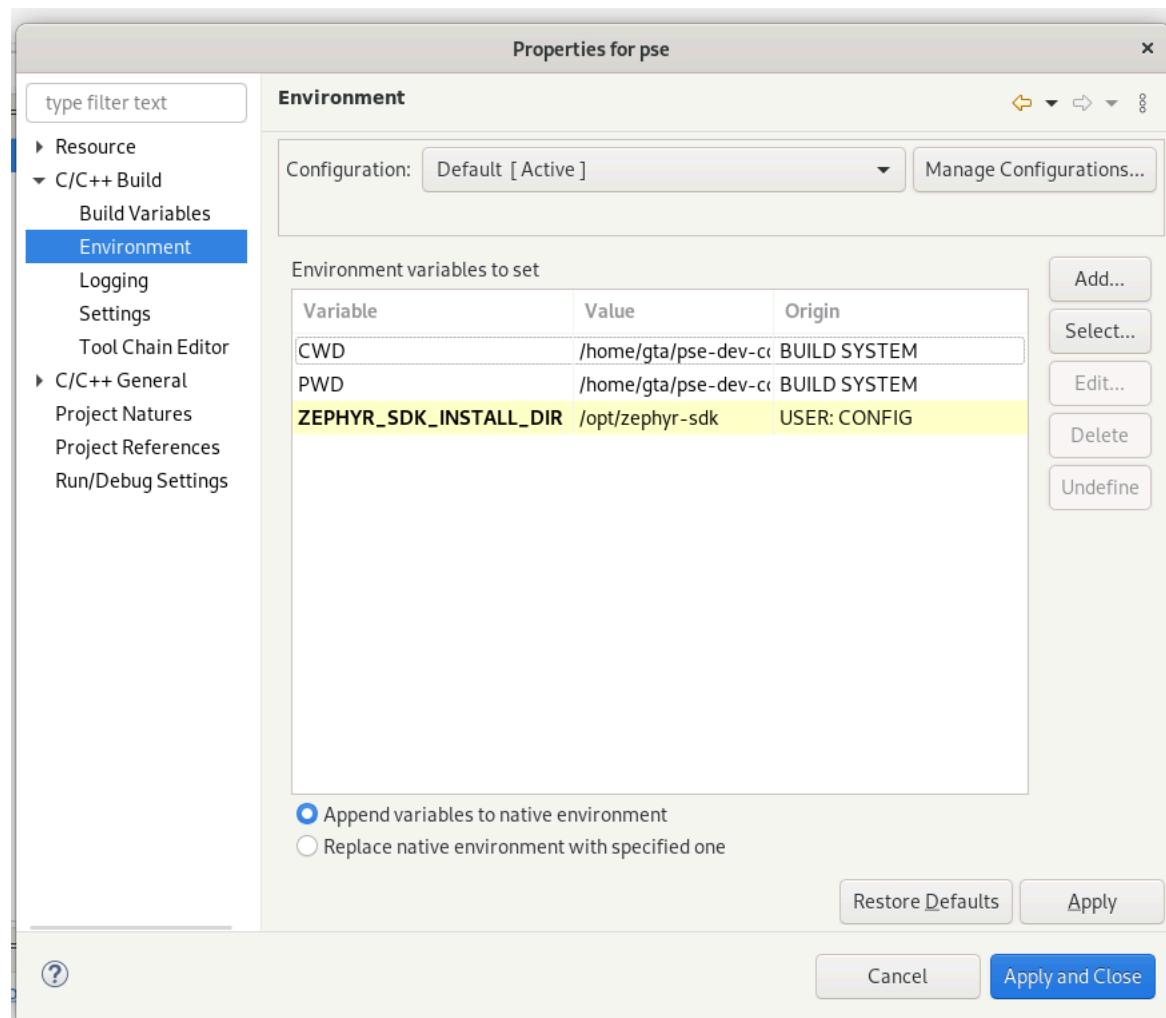
- Select `Use custom build argument` and erase data from Build arguments field.
- Unselect `Build on resource save`.

- Select **Build (incremental build)** and type the absolute path to the application that you want to build in the field.
- Select **Clean** and add **clean** in the field.



4. Click **Apply** to save the changes. Unfold **C/C++ Build** property and select **Environment** tab.
5. Click **Add** to set the following environment variables necessary to build the project:

- ZEPHYR_SDK_INSTALL_DIR = /opt/zephyr-sdk
- ZEPHYR_TOOLCHAIN_VARIANT = zephyr



6. Click **Apply** to save the changes

7. Click **Apply and Close**.

The project will now be built as if you are executing it from the command line as follows:

```
./build.sh <application_path>
```

It will build the Zephyr* application selected. To build a Zephyr application from Intel® System Debugger workspace, you must point **pse-dev-code-base** to that workspace.

Now you can [create a new Zephyr application](#).

Troubleshooting

Issue

Console (Debug console or another one) does not display commands that you type.

```
(gdb) info break  
No breakpoints or watchpoints.  
(gdb) info registers  
r0      0x20          32  
r1      0xffffffff    4294967295  
r2      0xe000ela0    3758154144  
r3      0xe000ela0    3758154144  
r4      0x7000cd38    1879100728  
r5      0x3           3  
r6      0x4           4  
r7      0x61034        397364  
r8      0x40f00140    1089470784  
r9      0x79000000    2030043136  
r10     0x40f00148    1089470792  
r11     0x4700dcca    1191238858  
r12     0x0           0  
sp      0x7000cc90    0x7000cc90  
lr      0x71000f97    1895829399  
pc      0x60002dc6    0x60002dc6 <__start+2>  
cpsr   0x0           0
```

Workaround

Resize the console window.

Issue

Errors occur during application builds (for example, Python* errors on the Zephyr base)

Workaround

Create a new workspace. If the error persists, create a new tree folder:

`~/system_debugger_new/workspace`.

Issue

IFWI reboots when `gdbserverproxy` connects to it

Workaround

Follow the steps below:

1. Go to ROM memory address to load the program. In the debugger console, execute the following instructions:

```
monitor load  
si
```

2. If the debugger breaks into 0x700... address, you can load the program (go to the next step). If the address is still 0x600..., execute the instructions again.
3. In the debugger console, execute the following instructions:

```
load <path-to-binary>/zephyr.elf
```

This will load the program and set the program counter to `__start`.

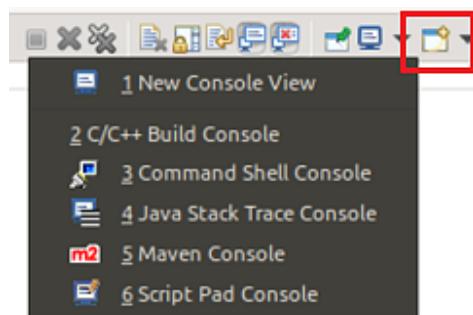
Issue

Python* package is missing

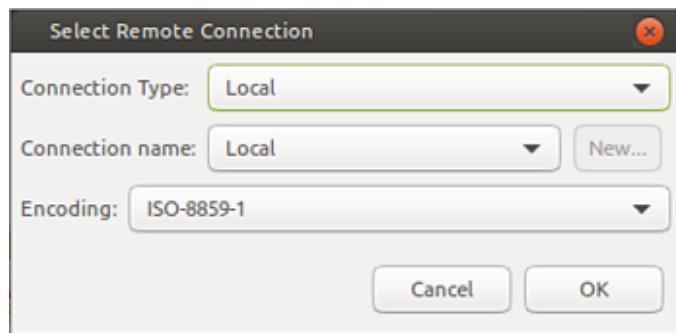
Workaround

Install a package in the virtual environment via Eclipse* IDE:

1. Click **Open Console** button and select **Command Shell Console**.



2. In the opened dialog box, set **Connection Type** and **Connection name** to *Local*.



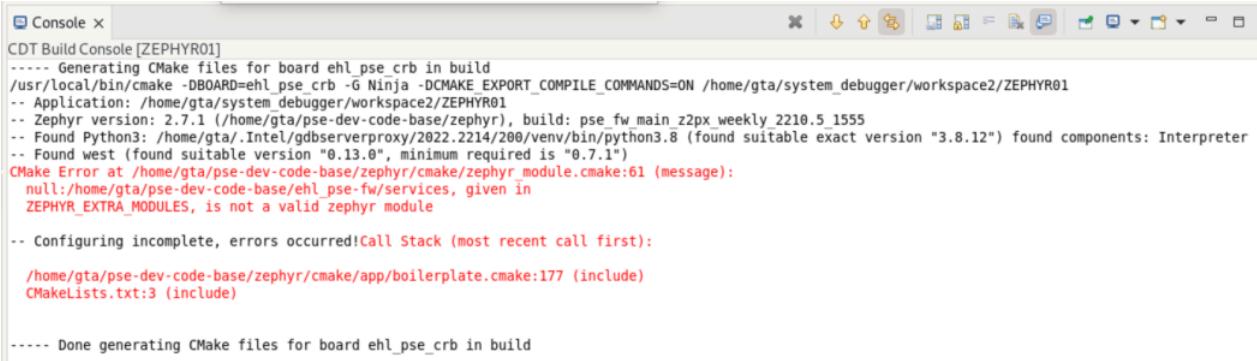
Click OK.

3. Local shell opens in the **Console** view. Execute the following command to install a Python* package:

```
pip install <package-name>
```

Issue

Error with environment variable ZEPHYR_EXTRA_MODULES prepending `null:`.



```
CDT Build Console [ZEPHYR01]
----- Generating CMake files for board ehl_pse_crb in build
/usr/local/bin/cmake -DBOARD=ehl_pse_crb -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON /home/gta/system_debugger/workspace2/ZEPHYR01
-- Application: /home/gta/system_debugger/workspace2/ZEPHYR01
-- Zephyr version: 2.7.1 (/home/gta/pse-dev-code-base/zephyr), build: pse fw main z2px_weekly_2210.5_1555
-- Found Python3: /home/gta/.Intel/gdbserverproxy/2022.2214/200/venv/bin/python3.8 (found suitable exact version "3.8.12") found components: Interpreter
-- Found west (found suitable version "0.13.0", minimum required is "0.7.1")
CMake Error at /home/gta/pse-dev-code-base/zephyr/cmake/zephyr module.cmake:61 (message):
  null:/home/gta/pse-dev-code-base/ehl_pse-fw/services, given in
  ZEPHYR_EXTRA_MODULES, is not a valid zephyr module

-- Configuring incomplete, errors occurred!Call Stack (most recent call first):
  /home/gta/pse-dev-code-base/zephyr/cmake/app/boilerplate.cmake:177 (include)
  CMakeLists.txt:3 (include)

----- Done generating CMake files for board ehl_pse_crb in build
```

Workaround

Modify environment variables to apply “Replace native environment with specified one”.

1. In Eclipse* IDE, open **Window > Preferences**.
2. Under **C/C++** node, unfold **Build** and select **Environment**.
3. Select **Replace native environment with specified one**.
4. Modify the following variables to a different value. For example:

- ZEPHYR_EXTRA_MODULES <path_to_pse>/ehl_pse-fw/
- CONFIG_DEBUG n

5. Click **Apply and close**, and set the correct value again.
6. Under **C/C++** node, unfold **Build** and select **Environment**.
7. Select **Replace native environment with specified one**.
8. Modify the following variables to the correct value:

- ZEPHYR_EXTRA_MODULES <path_to_pse>/ehl_pse-fw/services
- CONFIG_DEBUG y

9. Click **Apply and close**.
10. Clean the project and build again.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.