AMI Software Usage Document

AMI Debug for UEFI v3.0 for AptioV: BIOS Integration Guide and User Manual

Document Revision 1.09

Aug. 23, 2017

American Megatrends, Inc.
5555 Oakbrook Parkway
Suite 200
Norcross, GA 30093 (USA)

Legal

Disclaimer

This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced, transcribed, stored in a retrieval system, translated into any language or computer language, or transmitted in any form whatsoever without the prior written consent of the publisher, American Megatrends, Inc. American Megatrends, Inc. retains the right to update, change, modify this publication at any time, without notice.

For Additional Information

Call American Megatrends, Inc. at 1-800-828-9264 for additional information.

Limitations of Liability

In no event shall American Megatrends be held liable for any loss, expenses, or damages of any kind whatsoever, whether direct, indirect, incidental, or consequential, arising from the design or use of this product or the support materials provided with the product.

Limited Warranty

No warranties are made, either expressed or implied, with regard to the contents of this work, its merchantability, or fitness for a particular use. American Megatrends assumes no responsibility for errors and omissions or for the uses made of the material contained herein or reader decisions based on such use.

Trademark and Copyright Acknowledgments

American Megatrends, Inc.
5555 Oakbrook Parkway
Suite 200
Norcross, GA 30093 (USA)

All product names used in this publication are for identification purposes only and are trademarks of their respective companies.

Table of Contents

Document Information

Purpose

User guide for AptioV Debugger

Change History

Date	Revision	Description
2012-07-24	0.10	Start and first draft of the document
2012-07-27	0.11	Added halt module view and debug menu.
2013-02-08	0.90	Document reviewed and formatted.
2013-02-18	0.91	Document reviewed and few corrections made.
2014-02-18	0.92	Added Description for 3.01 AptioVDebugger features and added Windbg Debugger section.
2014-03-14	0.93	Added description for Performance Measurements window Sub-options
2014-07-25	0.94	Added description for Halt Checkpoint View and Troubleshooting section for AptioVDebugger Host and updated screenshots for detailed info
2014-08-06	0.95	Updated the AMIDebugRx User manual Link
2015-02-17	0.96	Updated the manual with info on new host features and dynamic loader plugin
2015-03-28	0.97	Updated first time install procedure for loader plugin
2015-04-01	0.98	Updated Toolbar options, corrected auto load-auto update settings, Added Debug port detection info, Corrected SMM Entry exit guidelines
2015-05-12	0.99	Updated description
2015-06-11	1.00	IO.ini rules and timestamp feature for debug log updated
2015-11-06	1.01	Added usage info of New features as conditional breakpoint ,inspection point, list of handle, current TPL, NVRAM variable view and Cscript support.
2016-01-12	1.02	Updated with AMI Debug for UEFI 23 features, tips and troubleshooting techniques
2016-04-08	1.03	Updated the visual studio redistributable dependency from 2005 to 2010 and 2013,debugger Usage, tips and troubleshooting techniques
2016-10-11	1.04	Updated the resolution related setting required for AptioVDebugger's optimal performance
2017-01-24	1.05	Updated Source Level Debugging information
2017-02-2	1.06	Updated the troubleshooting info.
2017-02-24	1.07	Updated the prerequisite check error codes in troubleshooting tips
2017-06-14	1.08	Updated for consolidated debugger modules.
2017-08-23	1.09	Updated for DCI Debug support information

Audience

Aptio Core Engineers Generic Chipset Porting Engineers, OEM Porting Engineers, and Aptio Customers.

Functional Description

Introduction

AMI Debug for UEFI is a set of features, supporting source-level symbolic (C and Assembler) debugging, during the development of UEFI based AMI firmware projects based on Aptio or UEFI Shell. It provides a software solution to firmware debugging, which goes along with the actual firmware build.

AMI Debug for UEFI uses a host/target configuration. The target system, which is debugged, communicates to a host system via an USB 2.0 port with an AMI Debug Rx device. AMI Debug for UEFI provides the BIOS and UEFI developer with functionality to control and debug Firmware/Software.

Definitions

- HOST - The System running AMI Debug for UEFI (AptioVDebugger host – VeB Eclipse interface)
- TARGET - The platform debugged by AMI Debug for UEFI using the above host interface.
- VeB – VisualeBIOS – Eclipse based IDE used with Aptio Projects

System Requirements

Software

Host software requirements

- PC Host with one of the following version of Windows OS 7\8\8.1
- VisualeBIOS (VeB). recommended to use VeB 7.19.516 or higher for better performance
- Java Runtime Environment (JRE) x86 - [Link]
- Microsoft Visual C++ 2010 Redistributable x86 - [Link]
- Microsoft Visual C++ 2013 Redistributable x86 - [Link]
- For source-level firmware/application debugging, the AptioV Project source & build environment (Project Build folder with necessary PDB information) for the Target Platform must be present.
- Microsoft Windows platform drivers for AMI Debug RX Device (Supports Win 7/8/8.1) (available with Debugger eModule)
- For USB3.0 Debugging - Microsoft Windows platform drivers for USB 3.0 Debug Cable (Supports Win 7/8/8.1) (available with Debugger eModule)
- Refer Recommended resolution and display setting

Target software requirements

Below are the reference target platform validated with AptioVDebugger

1. SugarBay
2. MSI Skylake
3. Purley

Hardware

Host hardware requirements

USB 2.0

An unlocked AMI Debug Rx Device to connect the Target platform and the Host System

USB 2.0 EHCI Debug Port and the target system must have USB 2.0 Controller with the Debug Port Capability feature. The cable only works with Specific EHCI USB port (Normally port 0) that has debug capability feature.

SERIAL

A Serial Null modem cable to connect the Target platform and the Host System.

USB 3.0

A USB 3.0 Debug Cable to connect the Target platform and the Host System.

*Note: Debugger _3.01.0011 onwards depends on the AMIDebugRx device for Authentication Checks. This means an **AMIDebugRx device with at least FW ver 3.3.6** needs to be connected on the Host side for the Application to pass the authentication check. AMIDebugRx 3.3.6 Firmware can be found with the Debugger eModule. Information on How to update the AMIDebugRx Firmware can be found in the AMIDebugRx User Manual.*

Target hardware requirements

In case of USB2 or Usb3.0 Debugging, target platform should have respective USB2, USB3.0 debug capable ports. Refer USB debug Port detection methods for more details.

BIOS Setup

AMI Debug for UEFI is a collection of eModules and Host Applications whose combination can be used to achieve different levels of debugging options on Aptio V Projects. Below table describes the module required for different debugging option.

Options	Module(s) required
Basic Redirection Only - Debug Messages - Status code Checkpoints	• AMIDebugRx eModule • AMIDebuggerPkg Interface
USB 2.0 Redirection Only - Debug Messages - Status code Checkpoints - Console Redirection	• AMIDebugRx eModule • AMIDebuggerPkg Interface • USBRedirection eModule
Source Level Debugging over USB 2.0\3.0 (USB Debugger) - Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc) - Debug Messages - Status code Checkpoints	• AMIDebuggerPkg Interface • Debugger eModule
USB 3.0 Redirection Only - Debug Messages only	• USB3Statuscode eModule Only
Source Level Debugging over Serial (Serial Debugger) - Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc) - Debug Messages - Status code Checkpoints	• AMIDebuggerPkg Interface • Debugger eModule

AMI Debug for UEFI consists of the below components.

- **AMIDebugRx eModule**

[Path: AptioV: $\AptioV\Binary\Modules\AMIDebugRx]

- **AMIDebuggerPkg Interface**

[Path: AptioV: $\AptioV\Binary\Interfaces\AMIDebuggerPkg]

- **Debugger eModule**

[Path: AptioV: $\AptioV\Binary\Modules\Debugger]

- **USBRedirection eModule**

[Path: AptioV: $\AptioV\Binary\Modules\USBRedirection]

- **USB3Statuscode eModule**

[Path: AptioV: $\AptioV\Binary\Modules\USB3Statuscode]

- **AptioVDebugger Host Application**

[Path: AptioV: $\AptioV\Binary\Modules\Debugger \AptioVDebugger,

Debug Mode Flags

The AptioV Project needs to be built\compiled with the Debug Flags enabled, so that the Debug information etc, required for the source to be debugged, will be made generated and available during the build process.

In Aptio V, the **Debug Mode** flags can be configured in the following way.

DEBUG_MODE - Making the ***DEBUG_MODE*** SDL token to ***OFF*** builds the firmware in Release mode. Once the ***DEBUG_MODE*** SDL token is ***ON*** it will build the project in Debug Mode and enables the following sub tokens that will affect the build as mentioned below.

OPTIMIZATION - This switch is used to control compiler and linker optimization. If ***DEBUG_MODE*** is set to ***OFF***, this switch is ignored. For source level debugging, it is recommended to disable this token

DEBUG_CODE - This switch is used to enable or disable debugging code such as debugging messages, assert statements, and any code enclosed into ***#ifdef EFI_DEBUG ... #endif***. If

DEBUG_MODE is set to off, this needs to be enabled or ***ON***. This switch is ignored when ***DEBUG_MODE*** is disabled.

Refer the following table for the effects of these tokens

DEBUG_MODE	OPTIMIZATION	DEBUG_CODE	Description	Build Mode
OFF	X	X	Debug_Mode is OFF, hence sub-tokens are disabled.	Release mode. Used for asm debugging of optimized code.
ON	OFF	OFF	Without any Optimization and Debug messages.	Used for asm debugging of un-optimized code without Debug messages.
ON	ON	OFF	Optimized and without any Debug messages	Used for asm debugging of optimized code.
ON	OFF	ON	Without any Optimization and with Debug messages enabled.	**Recommended for Source level Debugging**
ON	ON	ON	Optimized with Debug messages.	Source level debugging of optimized code and with Debug messages. (take less space, but debugging may not be very effective). This configuration is used for message redirection only case also.

☛ ***NOTE**: Build the target bios with DEBUG_MODE on (Optimization OFF and DEBUG_CODE ON for best results)*

☛ ***NOTE**: Once setup is made, first start the Debug session on the Host side (Ref: Setting Up Debugger) and then Power ON the Target.*

Master SDL Tokens

AMIDEBUGGERPKG_SUPPORT – Master SDL token to support AMIDebuggerPkg functionality in the project.

AMI_DEBUGGER_SUPPORT – Master SDL token to support Debugger functionality in the project. (Requires AMIDEBUG_RX_SUPPORT to be ON)

AMIDEBUG_RX_SUPPORT – Master SDL token to support AMIDebugRx functionality in the project. (Requires AMI_DEBUGGER_SUPPORT to be OFF)

USB_REDIRECTION_SUPPORT – Master SDL Token to support USB Redirection functionality in the project. (Requires AMIDEBUG_RX_SUPPORT to be ON)

SERIAL_DEBUGGER_SUPPORT – Master SDL Token to support SerialDebugger functionality in the project.

Master SDL Token	Redirection Only (AMIDebugRx only)	Redirection Only (USBRedirection - Debug Messages and Console Redirection)	USB Debugging (Source-Level-Debugging thru USB 2.0/3.0)	Serial Debugging (Source-Level-Debugging thru Serial)
AMIDEBUGGERPKG_SUPPORT	ON	ON	ON	ON
AMI_DEBUGGER_SUPPORT	OFF	OFF	ON	ON
AMIDEBUG_RX_SUPPORT	ON	ON	OFF	OFF
USB_REDIRECTION_SUPPORT	OFF	ON	OFF	OFF
SERIAL_DEBUGGER_SUPPORT	OFF	OFF	OFF	ON

Other SDL Tokens and their Usage

SEND_DEBUG_MESSAGE_SUPPORT– Enables the redirection of the debug Messages through the debugger. This can be switched off if the user does not want the Debug Messages redirection through debugger.

SEND_CHECKPOINT_SUPPORT – This token redirects the Progress code checkpoints as debug Messages through the Debugger. If this is not desired the user can turn OFF this token.

GENERIC_USB_CABLE_SUPPORT – This token enables the setting for Generic USB Debugger. To enable Debugger after the USB Stack is available in late DXE.

USB_3_DEBUG_SUPPORT – This token enables the settings to use the USB 3.0 Debugger.

AMIDebugRx Only Setup

To setup AMIDebugRx Only (for PEI + DXE Redirection only) using USB 2.0 EHCI Debug Port.

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface AMIDebugRx eModule	AMIDebugRx Device (Switch in Debug)	Checkpoint Mode- No Host application Debug Mode- VisualeBios with AptioVDebugger

Supports

- Debug Messages
- Statuscode Checkpoints

Steps

1. Download the AMIDebuggerPkg Interface and the AMIDebugRx eModule from AptioV database through VisualeBIOS.
2. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT** to ON.
3. Set the master SDL tokens **AMIDEBUG_RX_SUPPORT** to ON.
4. Set the required Debug Mode Flags in accordance
5. Switch AMI Debug Rx Device (Red box – Switch in Checkpoint) to get progress code checkpoints Switch AMI Debug Rx Device (Red box – Switch in Debug) to get traces in debugger console
6. Connect Host and Target using AMI Debug Rx Device

7. Power on the target system and the progress code checkpoints will be visible in the AMI Debug Rx Device screen if connected in checkpoint mode, if connected in debug mode user can observe traces in debugger console of VisualeBIOS application.

***NOTE**: For information on how to Use the AMIDebugRx Device with switch in Checkpoint, Refer to AMIDebugRx User Manual.*

USBRedirection Setup

To setup USBRedirection (for PEI + DXE Message Redirection and Console Redirection) using USB 2.0 EHCI Debug Port

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface AMIDebugRx eModule USBRedirection eModule	AMIDebugRx Device (Switch in Debug)	USBRedirection Host Or VisualeBios with AptioVDebugger

Supports

- Debug Messages
- Statuscode Checkpoints
- Console Redirection

Steps

1. Download the AMIDebuggerPkg Interface and the AMIDebugRx eModule from AptioV database through VisualeBIOS.
2. Download the USBRedirection eModule from AptioV database through VisualeBIOS.
3. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT** to ON.
4. Set the master SDL tokens **AMIDEBUG_RX_SUPPORT** to ON.
5. Set the master SDL tokens **USB_REDIRECTION_SUPPORT** to ON.
6. Set the required Debug Mode Flags in accordance
7. Connect the AMI Debug Rx Device (Red box – Switch in Debug) to the Debug Port of the target system and the Host system.
8. Run the USBRedirection Host App or start a Debugging Session on AptioVDebugger.

9. Power on the target system and the Debug Messages will be displayed in Host APP or debugger target console of AptioVDebugger.

Debugger Setup (USB 2.0 Debugger)

To setup the Debugger (PEI + DXE + SMM Source-level-Debugger) using USB 2.0 EHCI Debug Port

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface Debugger eModule	AMIDebugRx Device (Switch in Debug)	VisualeBios with AptioVDebugger

Supports

- Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc)
- Debug Messages
- Statuscode Checkpoints
- Console Redirection

Steps

1. Download the AMIDebuggerPkg Interface and the Debugger eModule and from AptioV database through VisualeBIOS.
2. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT** and **AMI_DEBUGGER_SUPPORT** to ON.
3. Set the required **DEBUG_MODE** flags to enable source level debugging or required settings (described above).
4. Follow the steps to Find the correct USB debug port as described below.(refer USB Debug Port Detection methods)
5. Connect the Host development system and the Target platform with an Unlocked **AMI Debug Rx Device (Red box – Switch in Debug)**.

Note: USB 2.0 Debugger can be used only with an Unlocked AMI Debug Rx Device. (For more information on how to unlock AMIDebugRx Device Refer to AMIDebugRx User Manual).

6. Select connectivity options from the debugger toolbar “Debug→Options” and select USB 2.

7. Start the **AptioVDebugger Host** application through menu option, select Debug→Start Debug. (Ref: Debugger Setup)
8. Power ON the Target

Debugger Setup (USB 3.0 Debugger)

To setup the Debugger (PEI after Memory + DXE + SMM Source-level-Debugger) using USB 3.0 XHCI Debug Capability Port

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface Debugger eModule	USB 3.0 Debug Cable + AMIDebugRx Device (Switch in Debug) - Used for Authentication	VisualeBios with AptioVDebugger

Supports

- Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc)
- Debug Messages
- Statuscode Checkpoints
- Console Redirection

Steps

1. Download the AMIDebuggerPkg Interface and the Debugger eModule and from AptioV database through VisualeBIOS.
2. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT**, and **AMI_DEBUGGER_SUPPORT** to ON.
3. Set the SDL token **USB_3_DEBUG_SUPPORT** to **ON**.
4. Set the required **DEBUG_MODE** flags to enable source level debugging or required settings (described above).
5. Follow the steps to Find the correct USB 3.0 debug capability port as described below.(refer XHCI Debug Port Detector Shell Application)
6. Connect the Host development system and the Target platform with a **USB 3.0 Debug Cable**.

Note: USB 3.0 Debugger still requires an Unlocked AMI Debug Rx Device with FW v3.3.6 to be connected for Authentication. (For more information on how to unlock AMIDebugRx Device Refer to AMIDebugRx User Manual).

7. Select connectivity options from the debugger toolbar "Debug→Options" and select USB 3 option.
8. Start the **AptioVDebugger Host** application through menu option, select Debug→Start Debug. (Ref: Debugger Setup)
9. Power ON the Target

Generic USB Debugger Setup (using USB 2.0\3.0 in DXE after USB)

To setup the Generic USB Debugger using USB 2.0\3.0 Port. This setup is only supported after the USB stack is available in late DXE phase.

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface Debugger eModule	AMIDebugRx Device (Switch in Debug) Or Retail USB Debug Cable	VisualeBios with AptioVDebugger

Supports

- Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc)
- Debug Messages
- Statuscode Checkpoints
- Console Redirection

Steps

1. Download the AMIDebuggerPkg Interface and the Debugger eModule and from AptioV database through VisualeBIOS.
2. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT**, and **AMI_DEBUGGER_SUPPORT** to ON.
3. Set the **GENERIC_USB_CABLE_SUPPORT** token to ON
4. Set the required **DEBUG_MODE** flags to enable source level debugging or required settings (described above).

5. Connect the Host development system and the Target platform with an Unlocked **AMI Debug Rx Device (Red box – Switch in Debug) or a Retail USB Debug Cable**.
6. Select USB 2 option from the debugger toolbar and start a Debug session on the Host application. (Ref: Debugger Setup)
7. Power ON the Target

Serial Debugger setup

To setup Serial Debugger (for PEI + DXE + SMM debugging) using the Serial Port.

Requirements

eModules	Connectivity	Host Application
AMIDebuggerPkg Interface Debugger eModule	Serial Null-Modem Cable + AMIDebugRx Device (Switch in Debug) - Used for Authentication	VisualeBios with AptioVDebugger

Supports

- Source Level Debugging (Step in\over\out, Memory, IO, PCI…etc)
- Debug Messages
- Statuscode Checkpoints
- Console Redirection

Steps

1. Download the AMIDebuggerPkg Interface and the Debugger eModule and from AptioV database through VisualeBIOS.
2. Set the master SDL tokens **AMIDEBUGGERPKG_SUPPORT**, and **AMI_DEBUGGER_SUPPORT** to ON.
3. Set the SDL token **SERIAL_DEBUGGER_SUPPORT** to ON.
4. Set the required **DEBUG_MODE** flags to enable source level debugging or required settings (described above).
5. Connect the Target and Host with Serial Null-Modem Cable.
6. Select connectivity options from "Debug→Options" and enter the COM Port Number

7. Enable/Disable Hardware Flow control ON/OFF check box based on project token set in the Target project [PcdSerialUseHardwareFlowControl value TRUE/FALSE in Debugger.sdl]
8. Start the **AptioVDebugger Host** application thru menu option, select Debug→Start Debug. (Ref: Debugger Setup)
9. Power ON the Target

NOTE: Serial Debugger still requires an Unlocked AMI Debug Rx Device with FW v3.3.6 to be connected for Authentication. (For more information on how to unlock AMIDebugRx Device Refer to AMIDebugRx User Manual).

USB3 Redirection Setup

To setup USB3Redirection (for PEI + DXE Message Redirection and Console Redirection) using USB 3.0 XHCI Debug Port

Requirements

eModules	Connectivity	Host Application
Usb3Statuscode eModule	USB3 Debug Cable	USB3StatusCode Host

Supports

- Debug Messages

Steps

1. Download the Usb3Statuscode eModule from AptioV database through Visual eBIOS.
2. Set the master SDL token **Usb3Statuscode_Support** ON.
3. Set the master SDL token **USB_3_DEBUG_SUPPORT** to ON.
4. Set the required **DEBUG_CODE** for enabling debig messages
5. Connect the target and host using USB.0 Debug cable
6. Run the USB3StatusCode Host App.
7. Power on the target system and the Debug Messages will be displayed on host app.

USB Debug Port Detection methods

Finding USB 2.0 EHCI Debug Port on Target

Using Debug Port Detector Shell Application

AMI now has an UEFI Shell application to help the users recognize the USB Debug Port. This application, “DebugPortDetector” will be available with the Debugger eModule under DebugPortDetector.zip.

Below are the steps involved to use the application →

- Flash a Bios Project which is bootable to shell .
- Boot to Shell with no AMIDebugRx connected.

Run the DebugPortDetectorX64 or DebugPortDetectorX86 EFI application from Shell. Based on the BIOS architecture.

- Connect the AMIDebugRx Device to any USB port on the board and follow the on-screen instructions.
- If an USB Device is detected in the Debug port then "A USB Device is CONNECTED In DebugPort" message will be displayed. Also the device type connected will be displayed (Eg: AMI DEBUG Rx or USB Optical Mouse etc.)

```
The USB Debug Port of EHCI Controller 1 with BDF(0,1A,0) is Located
A USB Device is DETECTED in this Port

***==> The USB Device Connected is  AMI DEBUG RX <==***
```

- If No USB device is detected in the debug port then it will show the message as "No USB Device is CONNECTED In the DebugPort".

```
The USB Debug Port of EHCI Controller 1 with BDF(0,1A,0) is Located
USB Device is NOT DETECTED in this Port

The USB Debug Port of EHCI Controller 2 with BDF(0,1D,0) is Located
USB Device is NOT DETECTED in this Port
```

- Now connect the device to some other USB port and press "ENTER" key to check if the debug port is detected.
- Press "ESC" key to exit the application.

Using Special BIOS (Polling method)

With This Method user can build the BIOS project with below mentioned SDL setup, so that on connecting the AMI Debug Rx Device to any USB Port and power ON the target, if the USB port connected is the Debug port the user will be notified via AMI Debug Rx Device.

Build and Flash the BIOS project with AMIDebugRx module enabled with the following SDL token Setup.

DEBUG_PORT_DETECTION_MODE - ON (will be OFF by default, ON it to build it in Port detection mode)

DEBUG_PORT_DETECTION_TIMEOUT - 0xXXXX

This SDL Token determines the Timeout for the Module to Poll Checkpoint 0xDB to the USB Debug Port, default Value is 0x1770 to poll for 1 Min, the rules to set a custom timeout is described in the SDL token Help.

[How to Set Timeout value - 0xDB Checkpoint happens in interval of 10mSec, so for 1 sec timeout you will need (1000/10=100)0x3e8/0xa=0x64. Default set for 0x1770 = 1 min]

- Connect the AMIDebugRx device to any USB port in target and power ON the target.
- If the connected USB port is the debug port then the AMI Debug Rx device screen will show checkpoint "DB" for some time based on the timeout.
- Try the same process for all USB ports available in the platform to find out the Debug port.

***NOTE**: If the user has tried in all the USB ports and is yet to identify the Debug port, this could mean that the USB Debug Port is probably available as an On board PIN,*
Please refer to the Board Specification to confirm the USB Debug port location and if it is an On-board PIN then verify the correct pin configuration with the Board Specification.
Connecting AMI Debug Rx or any USB Device with the incorrect PIN setup could damage the USB Device.
Also, in some platforms one or more EHCI controller may be disabled, in such cases please verify this with the Platform owner.

Finding USB 3.0 XHCI Debug Capability

Using XHCI Debug Port Detector Shell Application

AMI Debug for UEFI provides an UEFI Shell application to help the users recognize the XHCI USB Debug Capability Port. This application, "XHCIDebugPortDetector" will be available with the Debugger eModule under XHCIDebugPortDetector.zip.

Below are the steps involved to use the application:

1. Flash a Bios Project which is bootable to shell
2. Boot to Shell with no Cables connected on Target Side.
3. Run the XHCIDebugPortDetectorX64 or XHCIDebugPortDetectorX86 UEFI application from Shell. Based on the BIOS architecture.

   The Application will list if the platform supports XHCI Debug Capability and will list the BDF of the Debug Port.

Debugger Setup

Auto Update /First Time Installation

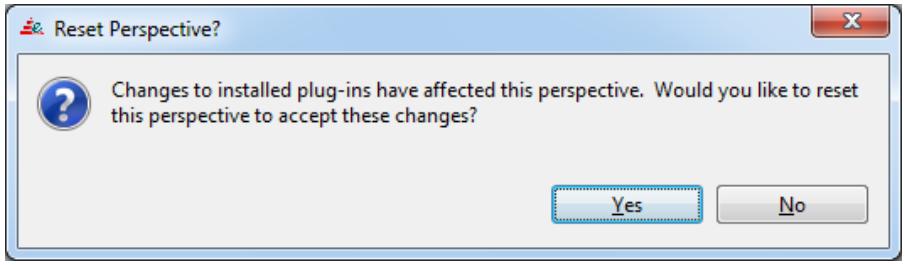
This approach is applicable starting from AptioVDebugger_3_02_0021 or above with Loader plugin support. Refer alternative Command Line Installation Approach if user wishes to install debugger version lesser than v3.02.0021.

When the VisualeBIOS already has a version of AptioVDebugger installed (> 3_02_xxxx) and the user wishes to upgrade to a newer version of AptioVDebugger or user wishes to install AptioVDebugger first time in VeB follow the below mentioned steps.

1 Download the required higher version of AptioVDebugger module to any AptioV project

2 Run VeB (which has AptioVDebugger installed using 'First Time Installation' steps) in Administrator mode

3 In VeB, Open the AptioV project containing the higher version of AptioVDebugger. When a window, like below, pops up. Choose YES For Perspective Switch Confirmation



4 The Debugger plugins in VeB would automatically be updated to the higher version available in the opened AptioV Project

5 The update is permanently saved so further launches of VeB will have the new version of AptioVDebugger loaded

Command Line Installation Approach

When the AptioVDebugger is being installed on a VisualeBIOS that did not previously have AptioVDebugger

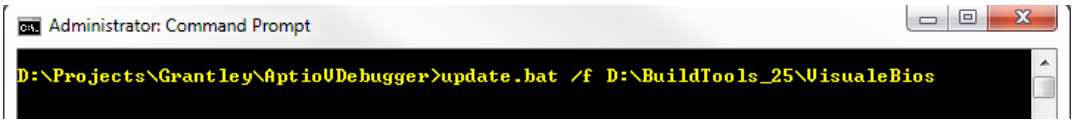
1 Open an AptioV project in VeB

2 Download the Latest labelled Debugger module from AptioV and Close VeB. User can observe AptioVDebugger folder in the Project path location.

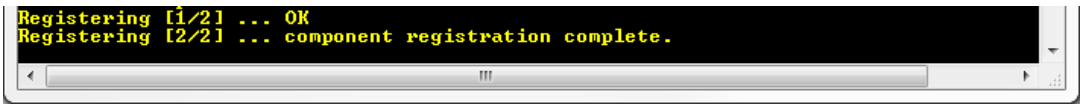
3 Open Command Prompt (As Administrator), navigate to the AptioVDebugger Directory (downloaded in Step 3) and run update.bat using following command.

update.bat /f <VeB_Path>

Example: update.bat /f D:\BuildTools_25\VisualeBios

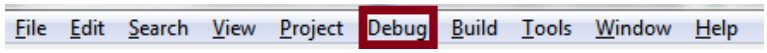


4 This will install the AptioVDebugger Plugins to VisualeBios (VeB). After completion of installation, user can see the message in command prompt as displayed in below screenshot.



5 Run VisualeBios (VeB) Application (As Administrator, First time only, not required everytime)

VisualeBios (VeB) should now be displaying the Debug menu and the Debug toolbar



In VeB, open the AptioV Project you wish to Debug

NOTE: If VEB was already running while executing the command update.bat in step 4, please close and reopen VeB to see the updated changes.

NOTE: If usb3 driver installation is required use /usb3 switch in step3. Command-Update.bat /f /usb3 <VeB_Path>

Hardware Setup

The physical connection between the host and target can be using

- AMI Debug RX Device (USB 2.0 Connection)
- Serial Null Modem Cable (COM Connection)
- Generic USB 2.0 Data Transfer Cable
- USB 3.0 Debug Cable (USB 3.0 Connection)

AMI Debug RX Device

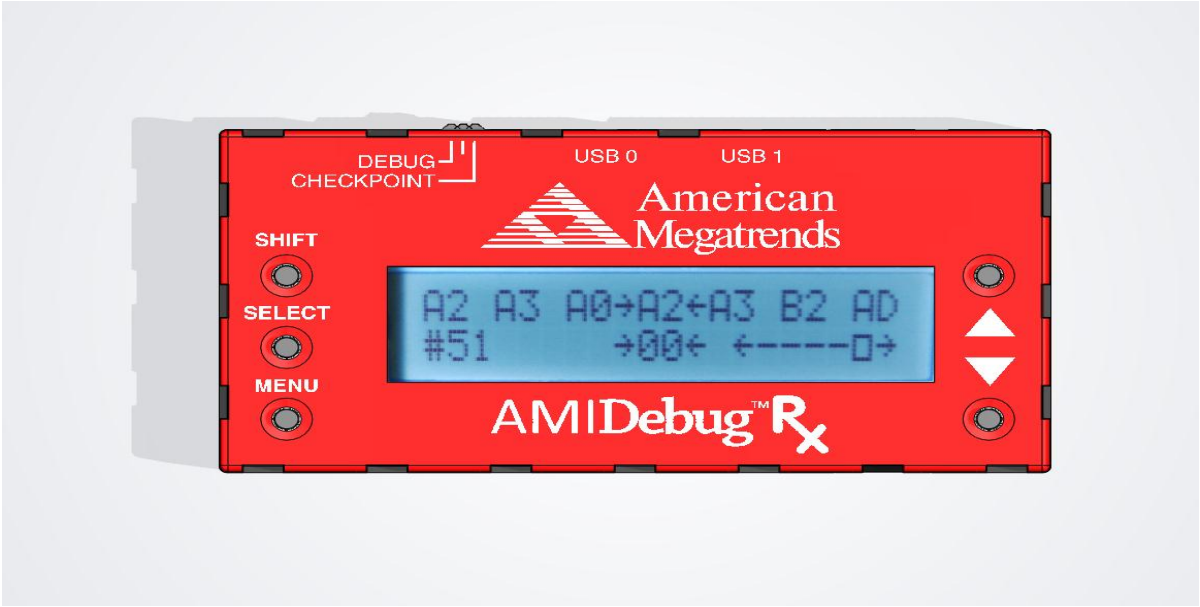


Figure: AMI Debug RX Device

The Host and the Aptio Target system can be physically connected with AMI Debug Rx device. Make sure to turn the switch to select the debug mode. AMI Debug RX device may need to be unlocked to support Debugger. Please check with your point of contact in AMI.

Stepping Into

*Note: Debugger_3.01.0011 onwards depends on the AMIDebugRx device for Authentication Checks. This means an **AMIDebugRx device with FW ver 3.3.6** needs to be connected on the Host side for the Application to pass the authentication check. AMIDebugRx 3.3.6 Firmware can be found with the Debugger eModule. Information on How to update the AMIDebugRx Firmware can be found in the AMIDebugRx User Manual.*

Serial Cables

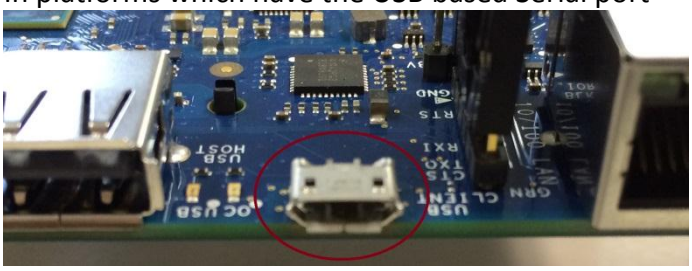
The serial cable connection between host & target requires a null-modem cable.

The target system requires RS-232 compatible serial port. The host system may use a RS-232 serial port or a USB-to-RS232 converter.

***NOTE**: This Serial port is capable of supporting the Hardware Flow control (depends on the Target platform also*

- **USB based Serial Port**

  In platforms which have the USB based Serial port -



  This Serial Port requires a Micro USB-to-USB cable

***NOTE**: This Serial Port cannot support Hardware Flow Control*

Generic USB 2.0 Data Transfer Cable

The Host and the Aptio Target system can be physically connected by Generic USB 2.0 Data Transfer Cable instead of an AMI Debug RX Device or a standard Serial null modem cable. These generic USB cables have the **limitation** that they can be used **only in DXE debugging**.

The recommended Generic USB 2.0 Data Transfer Cable is the BAFO Multi-LinQ USB 2.0 Generic File Transfer Cable. It is supported by the AMI Debug for UEFI. Any Generic USB 2.0 Data Transfer Cable might work, but they may require customization in Host driver INI file and the target's SDL tokens. For further details, please refer to the **Adding Support for a Generic File Transfer Cable** section in the **Generic File Transfer Cable** chapter.

Adding Support for a Generic File Transfer Cable

Vendor ID and Product ID

In order to add support for a different Generic USB 2.0 Data Transfer Cable, we need to first locate the Vendor ID and Product ID of the cable. To do this one may use the USBview tool from Microsoft (Usbview.exe is a Windows graphical user interface (GUI) application that allows you to browse all USB

controllers and connected USB devices on your computer.) or similar tool to find out which USB devices are connected to your HOST system. You can find information concerning Microsoft's USBview from following link

http://code.msdn.microsoft.com/windowshardware/USBView-sample-application-e3241039

Note: USBVIEW is available as sample code in Microsoft's Windows Driver Kit (WDK). The sample compiles and links in the standard Windows Driver Kit (WDK) build environment with Microsoft Visual C++ 6.0 on both x86-based and Alpha-based computers, producing a single executable binary Usbview.exe. One should build free or checked version as needed.

Select the particular port where your new cable is connected. You may see the Vendor ID (idVendor) and Product ID (idProduct) in Device Descriptor.

You can also find the same information with the driver INF file that can be found in the CD comes with the Generic USB 2.0 Data Transfer Cable. Nevertheless, there are some chances that the same CD may contains different drivers and for different cable. In that case, we need to find out which driver belongs to the Generic USB 2.0 Data Transfer Cable.

Target

Using different SDL tokens, support for new Generic USB 2.0 Data Transfer Cable can be added. The UsbDbgPort BIOS Driver detects the Generic USB 2.0 Data Transfer Cable and works with debugger drivers. The detection of USB cable is based on following SDL tokens.

USB_CABLE_VENDORID - Vendor ID of the cable.

USB_CABLE_PRODUCTID - Product ID of the cable.

USB_KNOWN_CLASSES - List of known class to ignore checking the UsbDbgPort support. UsbDbgPort will not be support for the Device classes listed. Please enter a list of known USB classes from 1-0xff.

The USB data transfer cables do not have a standard class code. So the detection of the cable is based on the USB_CABLE_VENDORID, USB_CABLE_PRODUCTID, and USB_KNOWN_CLASSES SDL tokens. If the USB device's Vendor ID and Product ID matches with these SDL tokens then it works with that device (Of-course, the USB device should have the BULK IN/OUT end points).

If the specified Vendor ID and Product ID not in the system, it checks each USB device's Class code. If the device's class code found in the USB_KNOWN_CLASSES list, it ignores that device and checks the next device. If the device's class code is not listed in the above list and it supports the bulk in/out endpoints, then UsbDbgPort supports that device.

Host

The same driver can be used without any modification for the new cables. The INF file has to be modified to add support for the new cable.

INF file used in windows for driver installation for support devices. For new cables, the device information needs to be added in the INF file. The two (2) sections of the INF file have to be modified based on the cable's Vendor ID and Product ID. The Vendor ID and Product ID can be found in the original driver's INF file that is coming with the Generic USB 2.0 Data Transfer Cable.

```
[AMI]
; Section contains list of devices that is support by this driver
%USB\VID_0525&PID_127A.DeviceDesc%=AMIUDBG.Dev, USB\VID_0525&PID_127A
%USB\VID_05e3&PID_0501.DeviceDesc%=AMIUDBG.Dev, USB\VID_05e3&PID_0501
%USB\VID_05e3&PID_0502.DeviceDesc%=AMIUDBG.Dev, USB\VID_05e3&PID_0502
; VID=0402&PID=5632
%USB\VID_0402&PID_5632.DeviceDesc%=AMIUDBG.Dev, USB\VID_0402&PID_5632
;//Add New Cables here///
```

```
[Strings]
AMI="American Megatrends, Inc"
MfgName="AMI"
USB\VID_0525&PID_127A.DeviceDesc="PLXTech Debug Cable"
USB\VID_05e3&PID_0501.DeviceDesc="GeneSys Debug Cable"
USB\VID_05e3&PID_0502.DeviceDesc="GeneSys Debug Cable"
; VID=0402&PID=5632
USB\VID_0402&PID_5632.DeviceDesc="Ali M5632 Debug Cable"
AMIUDBG.SvcDesc="AMI Debug Cable Driver"
;//Add New Cables Strings here///
```

USB 3.0 Debug Cable

The USB 3.0 Debugger requires a USB 3.0 Debug Cable to be connected between the Host and the Target. USB 3.0 debug Cable is a USB 3.0 Debug Crossover cable with USB 3.0 Male Pins on both ends.

Note: USB 3.0 Debug Cable needs to be connected to a USB 3.0 Port on Host side and a XHCI Debug Capability port on the Target side .

AptioVDebugger Feature Summary

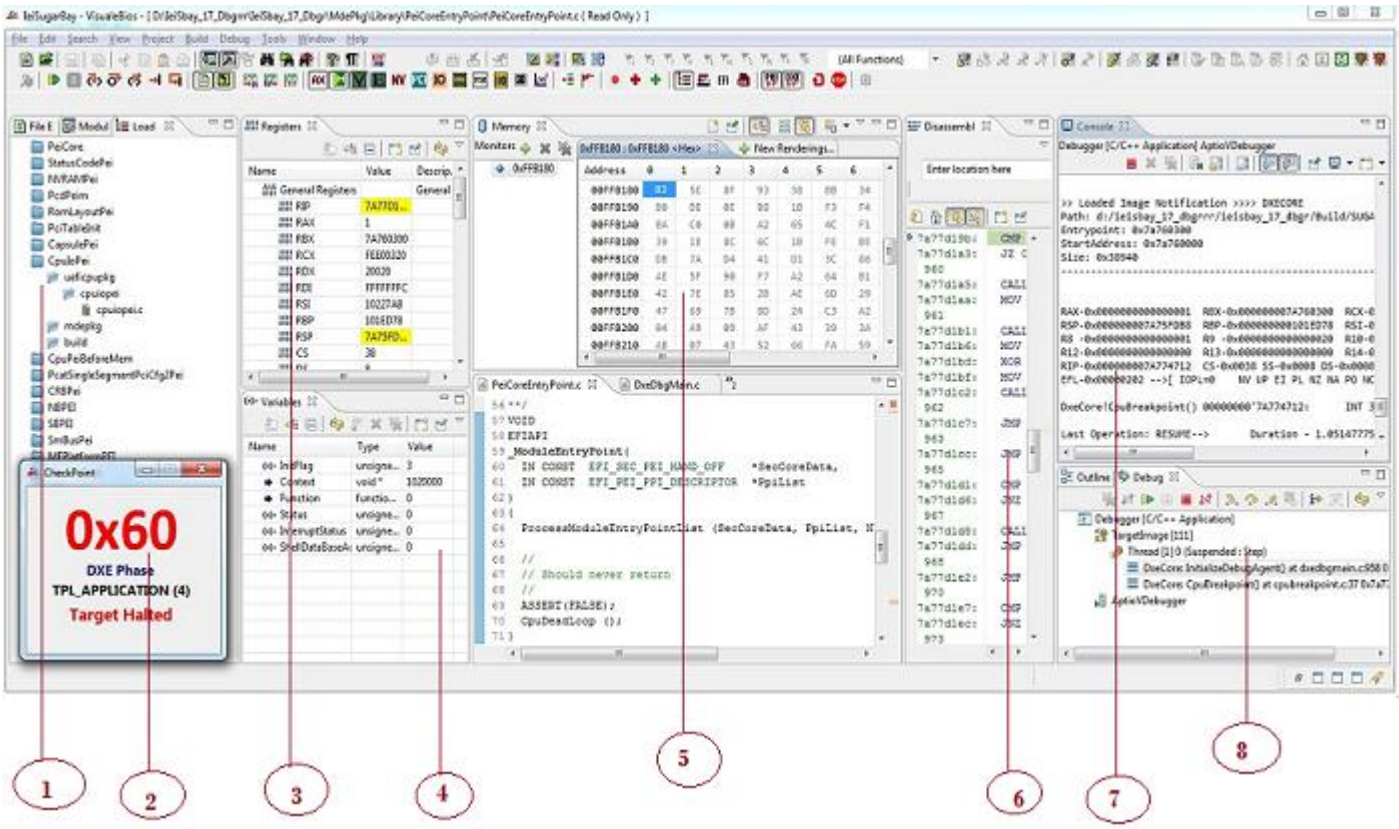


Figure: Screen shot of AptioVDebugger host

1- Loaded Driver View

2- Checkpoint View

3- Registers view

4- Variables view

5- Memory View

6- Disassembly View

7- Debugger Console Window

8- Call stack

AptioVDebugger Usage

Recommended resolution and display setting:

AptioVDebugger has been validated for below resolution and display setting. It is recommended to use below mentioned resolution and display setting for optimal performance.

1. Resolution Setting: It is advised to use AptioVDebugger having host system's resolution as per recommended setting at 1366x768 as per below image.

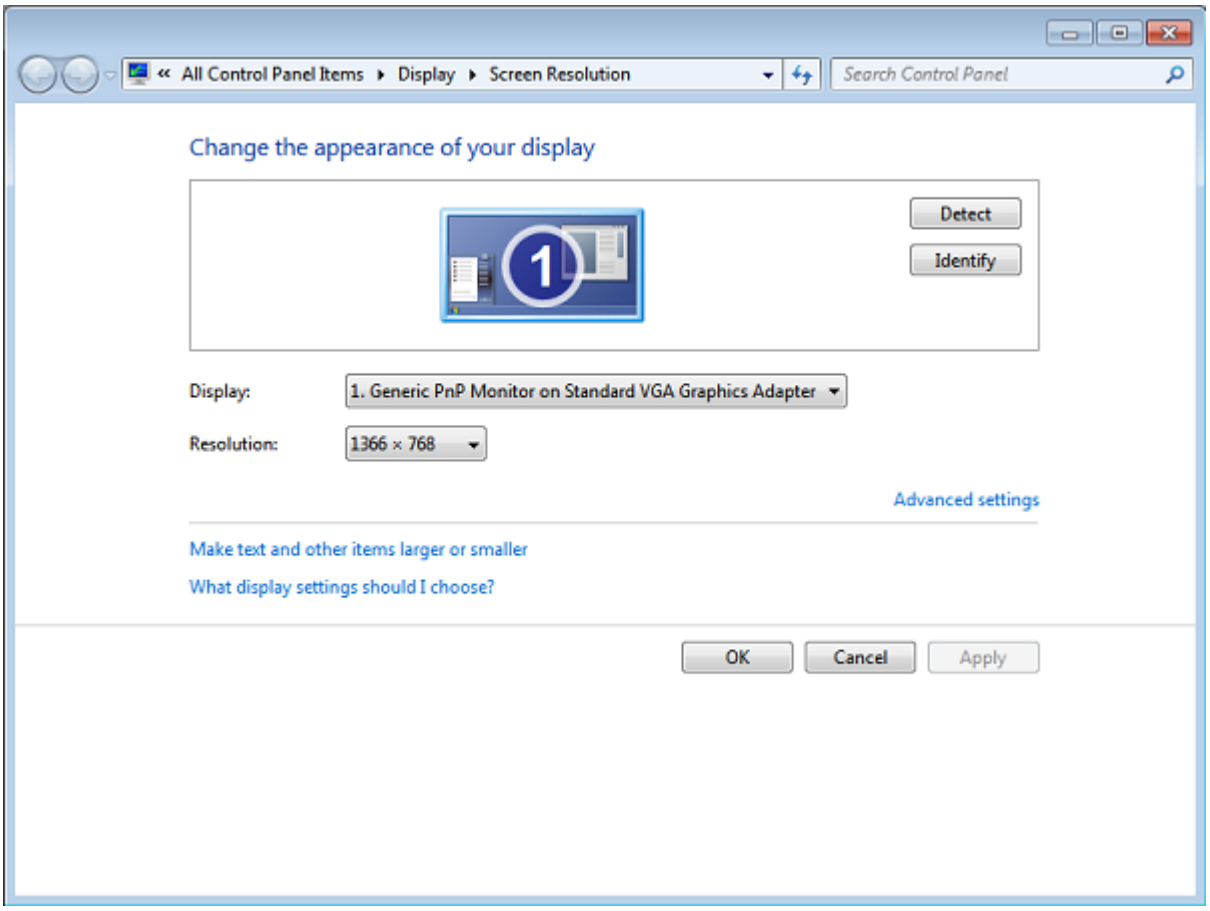


Figure: Recommended resolution setting

2. Display setting: Keep the preferred display setting to default setting at 100% for optimal usage of debugger as per below image.

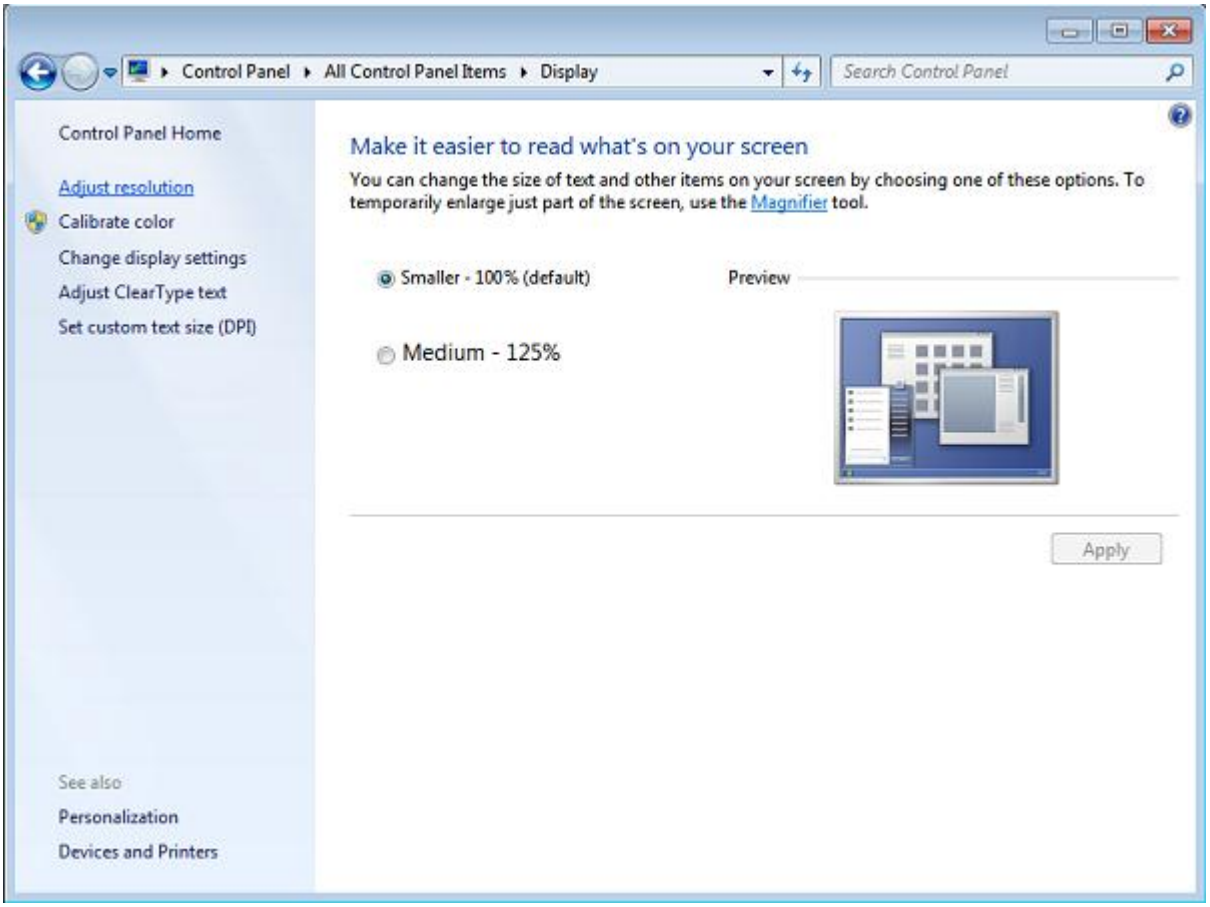


Figure: Recommended display setting

Starting a Debug Session

Follow the steps outlined below to start a new Debug Session,

1. Install AptioVDebugger on the host (Refer AptioVDebugger Installation).
2. Setup the Cables Connection:
    a. USB 2.0 Debugger: Connect using an unlocked AMI Debug Rx Device, on USB 2.0 (EHCI) Debug Port on Target and any USB port on Host. Target platform has to be powered off. Make sure the required driver is installed on Host system.
    b. USB 3.0 Debugger: Connect using a USB 3.0 Debug Cable, on USB 3.0 (XHCI) Debug Capability Port on Target and USB 3.0 port on Host. Target platform has to be powered off. Make sure the required driver(s) is installed on Host system.

c. Serial Debugger: Connect the Target platform and the Host System with a Serial Null Modem Cable with the Target platform powered off. Make sure the required driver is installed on Host system.

***NOTE**: Debugger v3.01 and above depend on the AMIDebugRx device for Authentication Checks. This means an **AMIDebugRx device with FW ver 3.3.6** needs to be connected on the Host side for the Application to pass the authentication check.*
AMIDebugRx 3.3.6 Firmware can be found with the Debugger eModule. Information on How to update the AMIDebugRx Firmware can be found in the AMIDebugRx User Manual.

3. Launch VisualeBIOS (with AptioVDebugger installed)
4. Load the AptioV BIOS project in VeB.DC
5. Target must be built with the required modules and SDL tokens set (Refer Bios Integration Guide).
6. Build the BIOS project in Debug Mode and flash the BIOS Image on the Target platform.
7. Debug options are found under the "Debug" menu
8. To Select between USB or COM, select "Debug→Options" from VeB or select the Options icon [ ] from Debug toolbar.

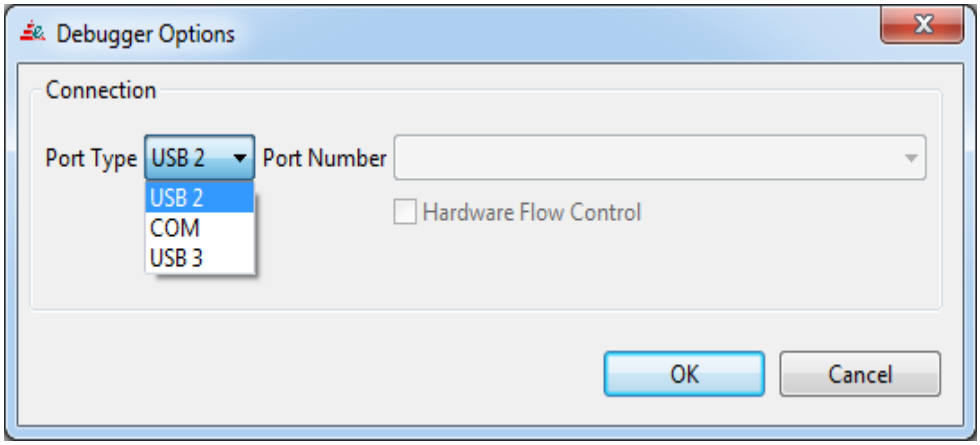
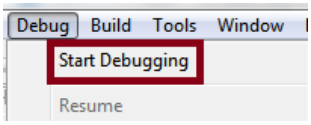


Figure: Debugger Options Window

***NOTE**: For COM option the Port number indicates the COM port number on the host side, USB Option does not require a Port Number.*

9. Now start a Debugging Session by selecting "Debug→Start Debugging" or select the Start Debug icon [ ] from Debug Toolbar



10. Wait until the launching Debugger process is finished. The launching progress can be found on the Right bottom corner of VisualeBIOS.

11. Now turn on the Target System, on Initial Breakpoint the Source level debugging will be available.

***NOTE**: Only One Debug session is supported at a time on a Host system, Multiple Debug sessions are not allowed.*

Stopping a Debug Session

To stop the target's Debugging Session,

- Click "Debug→Stop Debugging" on the Debug menu

- Or use the Hotkey: ***CTRL+F12***
- Or select the Debug Toolbar Icon [ ]

This action enables you to end a previous debug session and start a New Debug Session if required.

Displaying CPU Information

Select: Debug ->Windows->CPUID or select the Debug Toolbar Icon [ ]

Only one static CPUID Window is possible at a time. This window displays the details of the Target CPU.

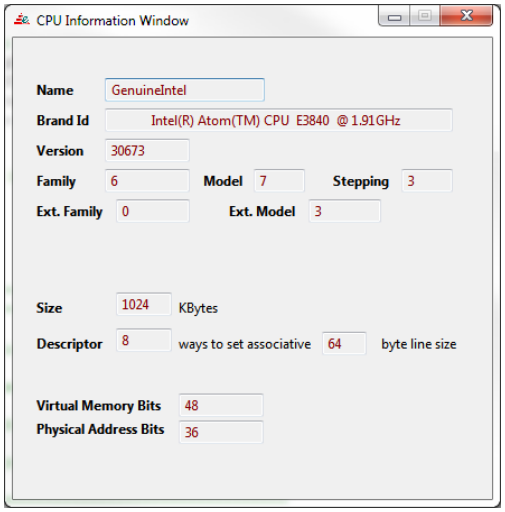


Figure: CPUID View.

Displaying MSR Information

Select: Debug →Windows→MSR or select the Debug Toolbar Icon [ ]

Only one static MSR Window is possible at a time.

On opening, the MSR window, the details of the MSR already present in the MSR.ini file will be displayed.

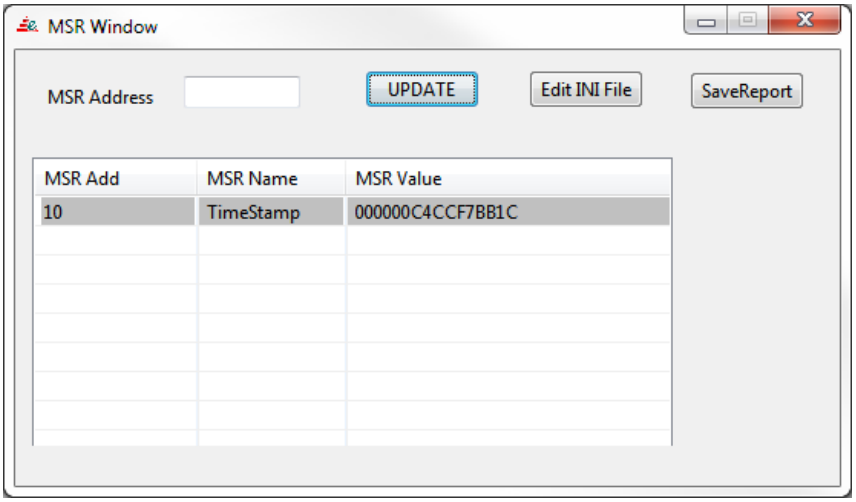


Figure: MSR View.

Users can add new entry of MSR in MSR window by entering address in MSR address in text filed and click update button. If the entered MSR is valid then new entry will be updated in MSR.ini file.

'Edit NI file' button helps users to Edit the MSR.ini in the VeB editor, INI file has to be modified according to the set of rules mentioned in the INI file.

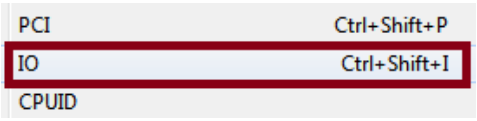
User can edit the values of a Writable MSR in MSR window.

Working with IO & II0

IO

Users can access the IO Dialog by -

- Select: *Debug → Windows → IO*



- Or using Hotkey: ***CTRL+SHIFT+I***
- Or using the Debug Toolbar Icon [IO]

Only one static IO Window is possible at a time.

Enter the IO Port address of the IO Space and length want to observe in the corresponding Textboxes.

Options

Edit - User can select the desired offset from the grid and edit it, select something outside the grid or press Enter to write the edited value to Target.

Display - The IO Space Display format can be changed by selecting Radio buttons Byte, Word, Dword. Displays in Big Endian Format.

Display Bits - displays the values in Binary format.

NOTE: Only one IO Window is possible at a time, it is not Auto updatable while stepping, user need to Refresh button.

Save Report - Dumps the data to a Text file.

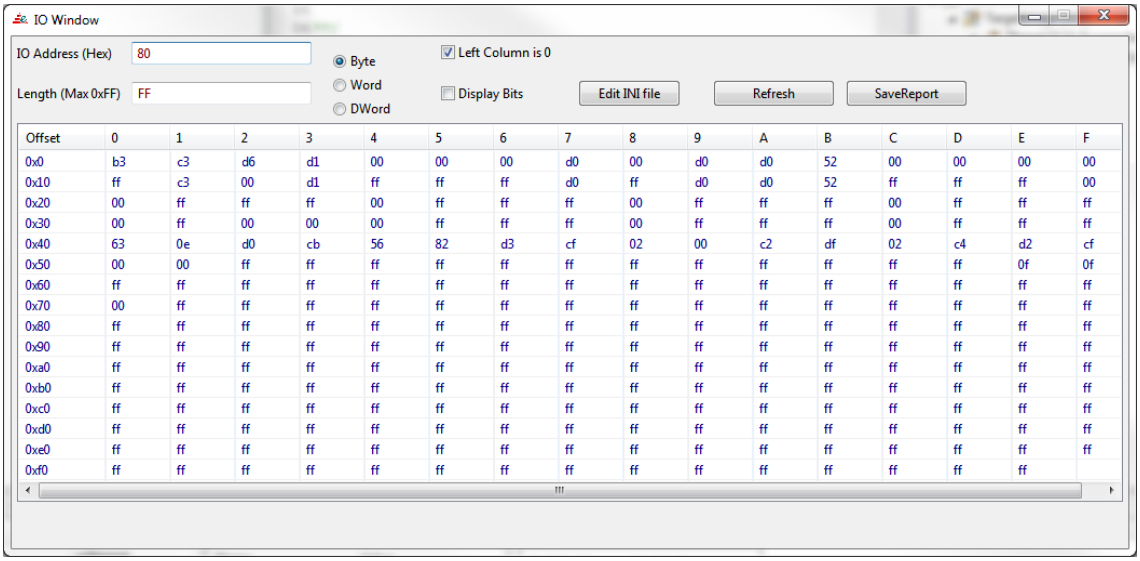


Figure: IO View.

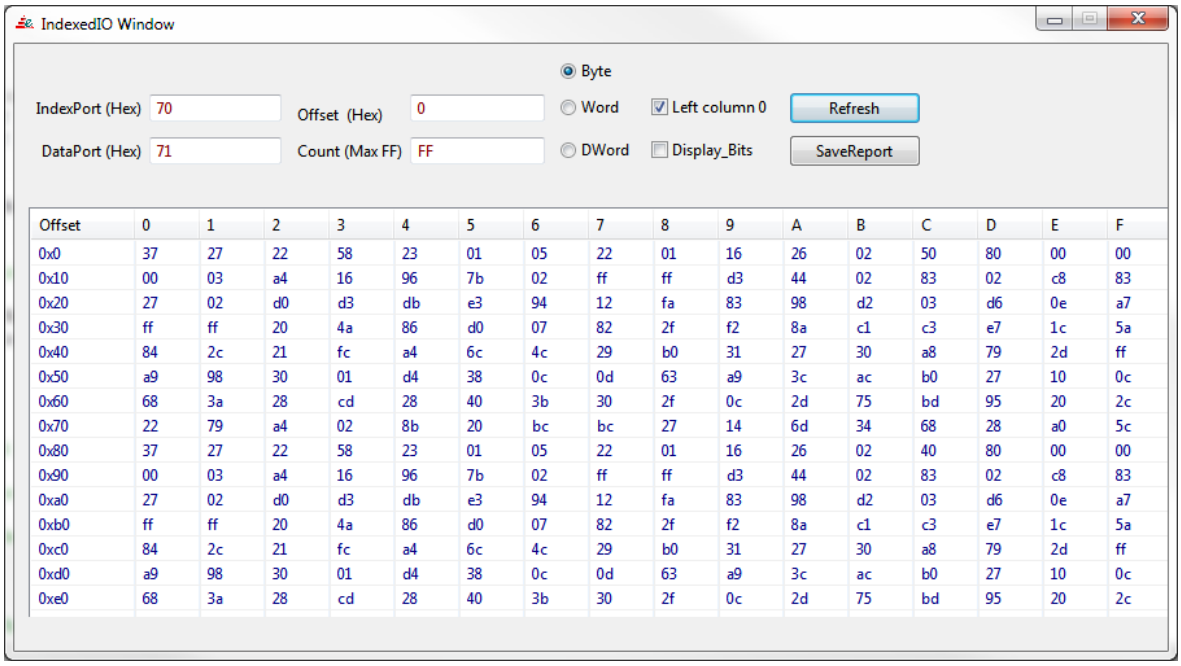
Edit INI File - Users can now use the io.ini to restrict the IO access ports, the io.ini can be edited on clicking the "Edit INI file" button. The INI file need to be modified using the rules specified in o.ini.

IIO

Select: Debug →Windows→IIO Or using the Debug Toolbar Icon [ ]

Only one static IIO Window is possible at a time.

Enter "Index Port" and "Data Port" of the IIO Space and Offset and Size want to observe in the corresponding Textboxes. Data can be modified by mouse click on respective location. The same modification will reflect on target side also. To change the format of display use Radio Buttons Byte, Word and Dword



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0	37	27	22	58	23	01	05	22	01	16	26	02	50	80	00	00
0x10	00	03	a4	16	96	7b	02	ff	ff	d3	44	02	83	02	c8	83
0x20	27	02	d0	d3	db	e3	94	12	fa	83	98	d2	03	d6	0e	a7
0x30	ff	ff	20	4a	86	d0	07	82	2f	f2	8a	c1	c3	e7	1c	5a
0x40	84	2c	21	fc	a4	6c	4c	29	b0	31	27	30	a8	79	2d	ff
0x50	a9	98	30	01	d4	38	0c	0d	63	a9	3c	ac	b0	27	10	0c
0x60	68	3a	28	cd	28	40	3b	30	2f	0c	2d	75	bd	95	20	2c
0x70	22	79	a4	02	8b	20	bc	bc	27	14	6d	34	68	28	a0	5c
0x80	37	27	22	58	23	01	05	22	01	16	26	02	40	80	00	00
0x90	00	03	a4	16	96	7b	02	ff	ff	d3	44	02	83	02	c8	83
0xa0	27	02	d0	d3	db	e3	94	12	fa	83	98	d2	03	d6	0e	a7
0xb0	ff	ff	20	4a	86	d0	07	82	2f	f2	8a	c1	c3	e7	1c	5a
0xc0	84	2c	21	fc	a4	6c	4c	29	b0	31	27	30	a8	79	2d	ff
0xd0	a9	98	30	01	d4	38	0c	0d	63	a9	3c	ac	b0	27	10	0c
0xe0	68	3a	28	cd	28	40	3b	30	2f	0c	2d	75	bd	95	20	2c

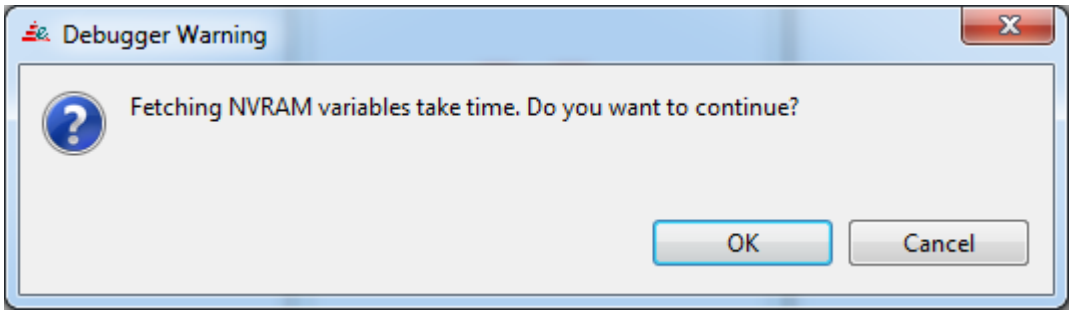
Figure: IIO View.

Getting NVRAM Variables Information

Select: Debug →Windows→NVRAM variables Or using the Debug Toolbar Icon [NV]

Only one static NVRAM Window is possible at a time.

Fetching all NVRAM variables take time. User will be notified before proceeding further as shown below.

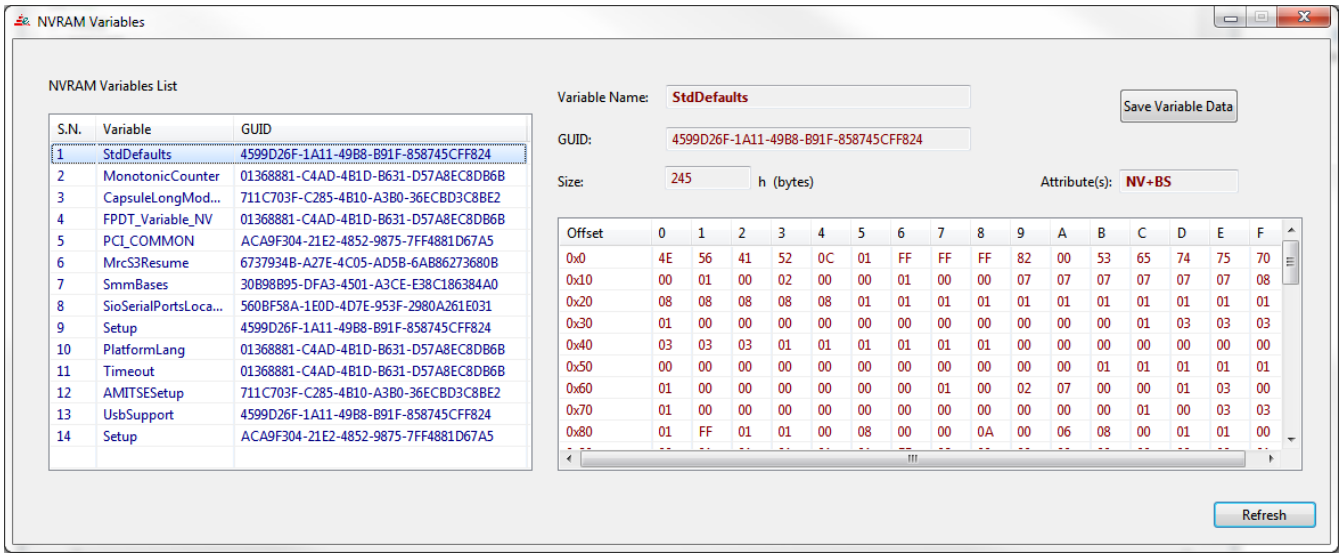


If user clicks ok then debugger will open the NVRAM window. In NVRAM window there is NVRAM variable panel present where user can see the list of NVRAM variable with its GUID information.

If user want to see the content of specific NVRAM variable then user has to double click on that variable name.

After double click, right side of fields in the NVRAM window will automatically update with info such as Variable name, Variable GUID, size of variable in bytes, attribute and content of variable.

Click on “Refresh” button to get updated value of NVRAM variable.



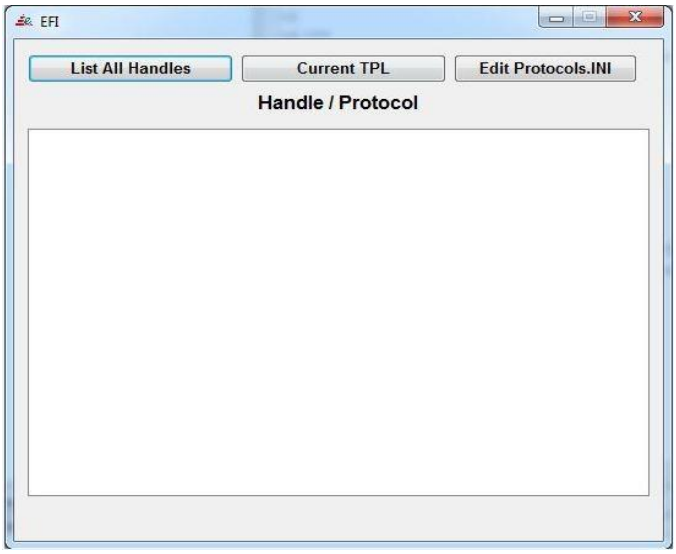
User can save the NVRAM variable data into the file by clicking on “Save Variable Data” button

List Handles and Current TPL

Select: Debug→List Handles Or using the Debug Toolbar Icon [EFI]

This feature will display all the current protocols available in the handles present in the target system at the debug output window and Current TPL value.

This will open the EFI window, which contain List All Handles, Current TPL and Edit INI button.

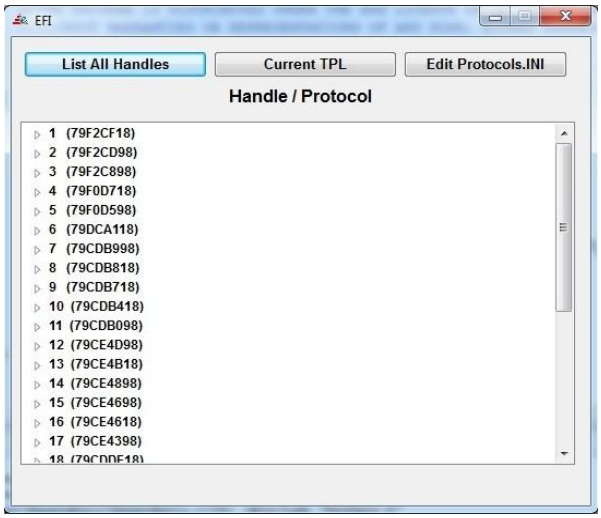


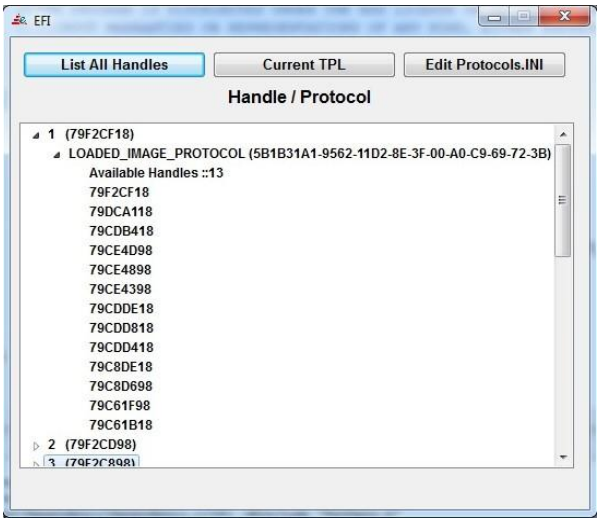
List All Handles:

By click on this button, user can get the list of all handles.

If you want to see, the list of protocol installed on specific handle then expand that handle. It will give list of protocol name with their GUID.

In addition, user can see the list of handle for specific protocol, simply by expanding that specific protocol. After expanding the specific protocol, it display list of available handle with handle count.





Edit INI:

This Button is for editing the protocol.ini file, which contain the list of protocol, their GUID and nickname of the protocol. The format of the ini file is described below. If the user wishes to customize the ini file, then the same format is to be followed and some important rules which are listed later in this section. In the protocols.ini, only one set of Get Handles can be supported. Protocols.ini needs to be updated if handle name displayed as “Unknown” with GUID.

File Format

The file format of MSR.ini will be as follows.

[SectionName]

[SubSectionName]

GUID = Protocol Name = Protocol Nick Name

GUID = Protocol Name = Protocol Nick Name

GUID = Protocol Name = Protocol Nick Name

.

.

[/EndSubSection]

[/EndSection]

For Example.

[CPU]

[GUID]

0x5B1B31A1 0x9562 0x11D2 0x8E 0x3F 0x00 0xA0 0xC9 0x69 0x72 0x3B = LOADED_IMAGE_PROTOCOL = Image

[/GUID]

[/CPU]

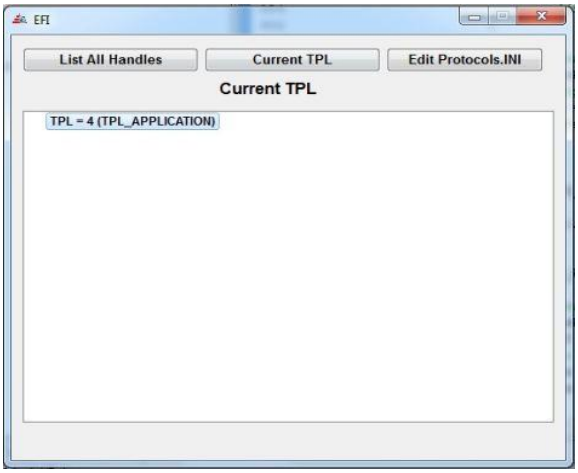
NOTE: List Handles feature available in DXE phase not in PEI phase .

Current TPL:-

User can see the current TPL value at DXE phase

In order to see the current TPL value, user can use this button. By clicking on this button, EFI Window will display the current TPL value.

Below screenshot is showing the current TPL view

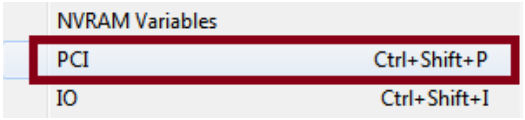


NOTE: Current TPL feature available from DXE phase Onward Not in PEI phase.

Working with PCI

Users can access the PCI by -

- Select: *Debug → Windows → PCI*



- Or using Hotkey: ***CTRL+SHIFT+P***

- Or using the Debug Toolbar Icon [PCI]

Users need to first Scan PCI to populate the available PCI List. Clicking on 'Sacn PCI' button will list the available PCI devices BDF, Vendor ID and Dev ID.

In From and To text boxes user can specify the bus numbers range used to Scan PCI. The PCI List will display the corresponding available PCI devices information.

Selecting a BDF entry in the PCI List will display the PCI device's Configuration Space.

This is 256 bytes that are addressable by knowing the 8-bit PCI bus, 5-bit device, and 3-bit function numbers for the device (commonly referred to as the *BDF* or *B/D/F*, as abbreviated from *bus/device/function*).

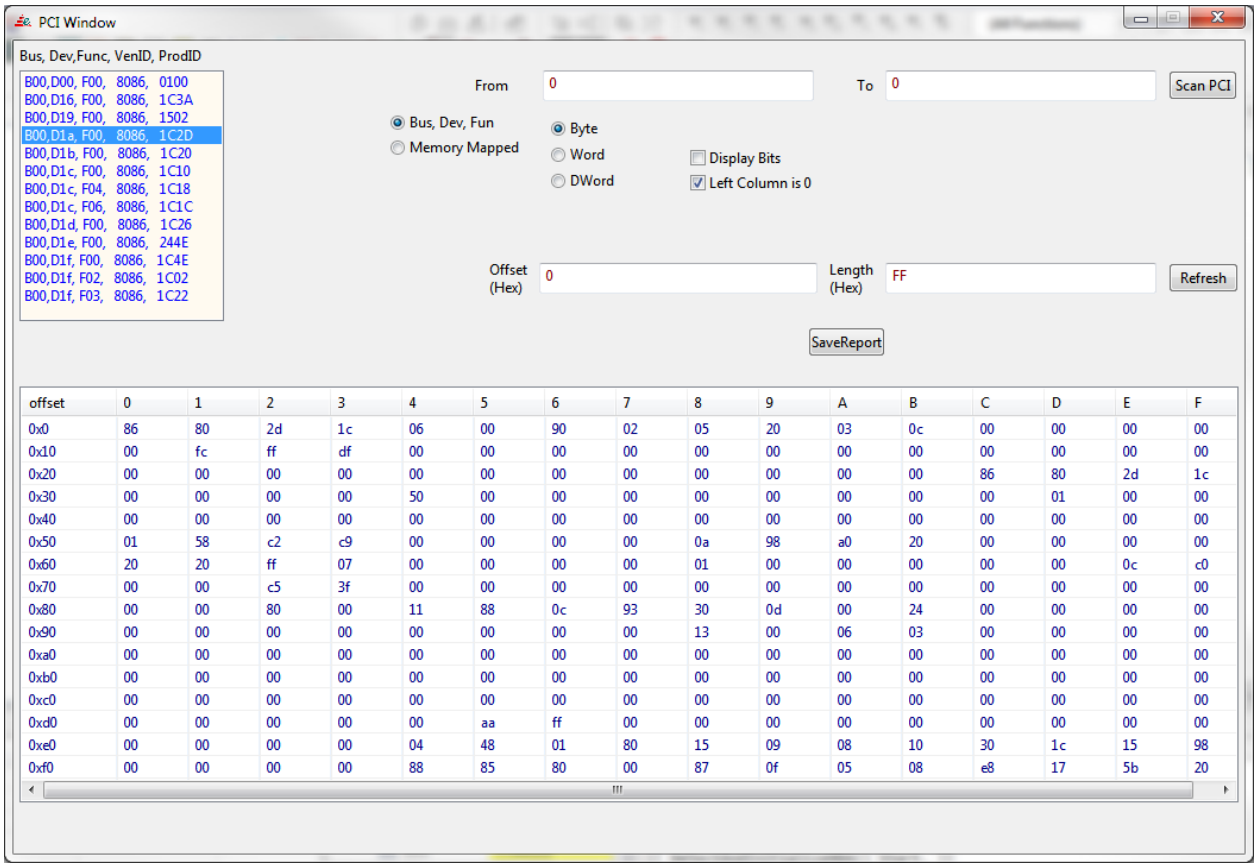


Figure: PCI View.

Options

Edit - User can select the desired offset from the grid and edit it, select something outside the grid or press Enter to write the edited value to Target.

Display - The PCI Configuration Display format can be changed by selecting Radio buttons Byte, Word, Dword. Displays in Big Endian Format.

Memory Mapped Support - To enable memory mapped support, select the grid you would like to use as input for the Memory Mapped view and check the radio button Memory Mapped.

The value in the selected grid will be used as the Memory start address and the data from that Memory address will be shown in the PCI Config display.

Display Bits - displays the values in Binary format.

NOTE: Only one PCI Window is possible at a time, this view is not Auto updatable while stepping, user need to select 'Scan PCI' and select the PCI entry from list to see the updated Config space.

Or, select Refresh button.

Save Report - Dumps the Config data to a Text file.

Performance Measurements

AMI provides solutions to view the POST performance Data of an AptioV BIOS boot session, one such method is by using the PerformanceMeasurement eModule and "AMIPerfRecordDump" Windows Application. Debugger's Performance Measurements View behaves similarly, with the option to avoid the need to boot to Windows to view the Performance Data.

Select :Debug →Windows→Performance Measurements" Or using the Debug Toolbar Icon [ ]

Performance Measurements window will display the current Boot's performance Measurement Data when Debugger host is run in Performance Mode.

NOTE: Performance Measurements will be displayed only in Performance Mode and not in Debug Mode. When a new Debug session is started on the AptioVDebugger if the Performance Measurement window is opened before the initial communication with Target, the Host is said to be in 'Performance Mode', else it is in 'Debug Mode'.

NOTE: In 'Performance Mode' source-level-debugging is not supported, hence the target will not halt on any breakpoint, it will boot normally and populate the Performance Data in the window after the Boot is completed.

Steps to Enter Performance Mode?

- Start a new Debugging session on the Host
- Open Performance Measurements window from "Debug→Windows→Performance Measurements"
- Now Power ON the Target
- After boot the Performance Measurements will be displayed in the window

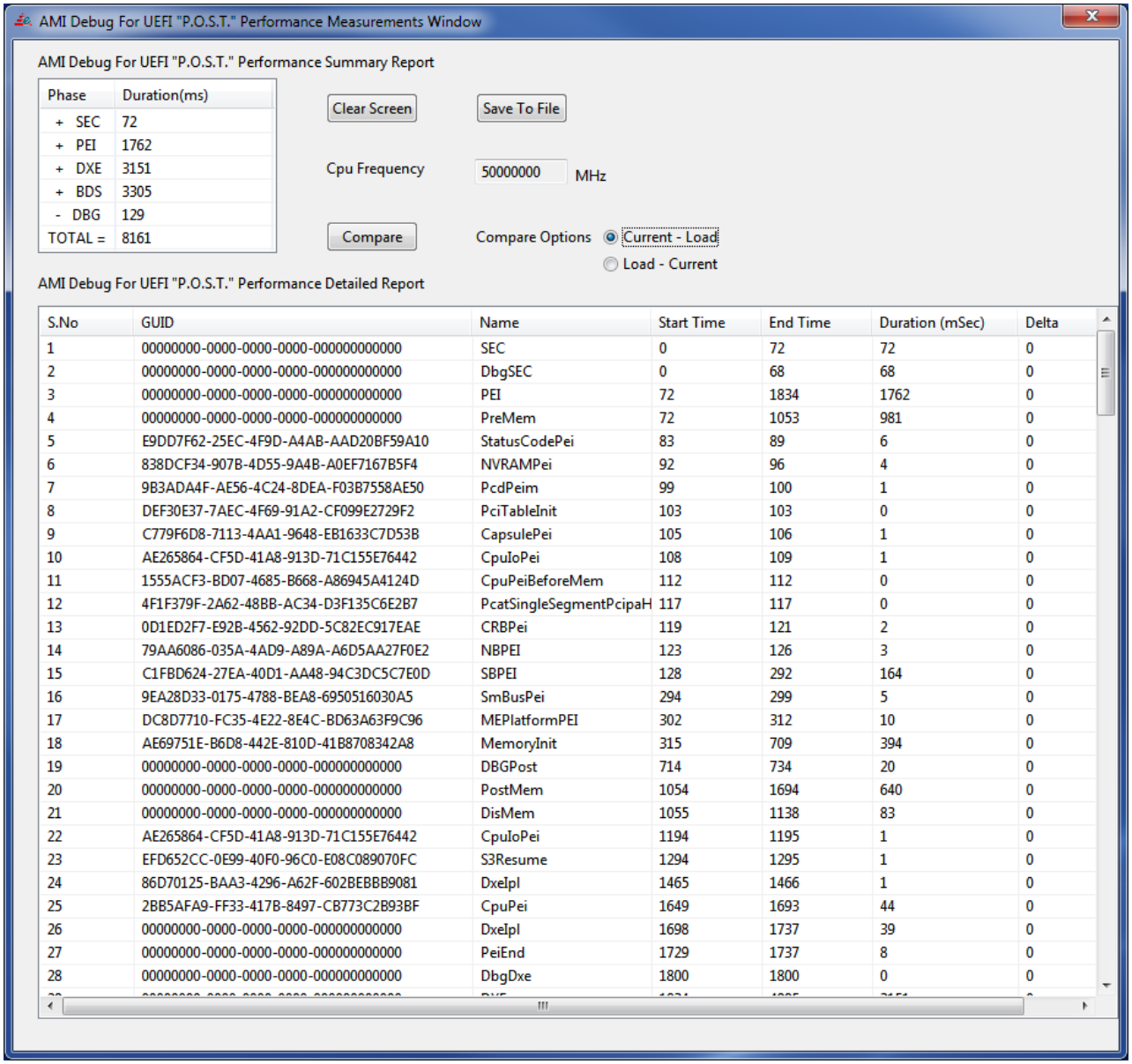


Figure: Performance Measurement View.

NOTE: Debugger Depends on the PerformanceMeasurement eModule to help fetch and display the Performance Data.

Debugger's Performance Display can be used as an alternate to AMI's Windows Application "AMIPerfRecordDump", this application dumps the Performance Data to a file when run from Windows. For more information on 'AMIPerfRecordDump' application, refer to the PerformanceMeasurement.chm in PerformanceMeasurement eModule.

Understanding and using the Performance Measurements View and its Options

The Performance Measurements View consists of two sections –

Performance Summary Report

This section displays the overall Performance Summary with highlights on the various phases (SEC, PEI, DXE and BDS), The added section 'DBG' covers the time used by Debugger eModule for its usage, like Debugger initialization in various phases, USB Initialization for Host-target Communication, etc.

To summarize the Total time taken for the current Boot Session, the time taken by various phases are added together and the debugger's time is subtracted. [TOTAL = SEC + PEI + DXE + BDS – DBG].

Performance Detailed Report

This section displays the individual time taken by specific phases, drivers, special tasks etc. All the Drivers will have their GUIDs listed under the GUID column and Phases and special tasks will not have any GUID.

S.NO – Serial Number

GUID – Globally Unique Identifier

NAME – Name of the Driver or Phase

START TIME – the Relative start time of the phase\driver in mSec

END TIME – the Relative end time of the phase\driver in mSec

DURATION – The difference between Start time and end time in mSec

DELTA – populated on use of the Compare option, described under Compare description

The names of the tasks used by Debugger will have the 'DBG' prefix. The value of the DBG in Summary is a sum of all the debugger tasks.

CPU Frequency

This displays the Target's CPU frequency of the current session.

Save To File

This option saves or dumps the displayed Data to a File, the saved file is formatted similar to the output of the 'AMIPerfRecordDump' windows application. The saved file can be used as an input to the compare option.

Compare

This option compare the current sessions's performance Records with the records from a previously saved session and populates the 'DELTA' column in the Performance Measurements window. Input can be a file saved from the Debugger's Performance Measurement→Save To File or an output file of the 'AMIPerfRecordDump' windows application.

Sub options 'Current – Load' will populate DELTA column based on the 'current session time' – 'load session time' and similarly 'Load - Current' populates the DELTA column based on the 'load session time' – 'current session time'.

ClearScreen

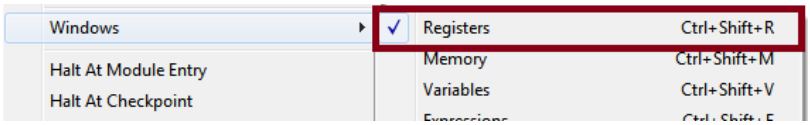
This option clears the performance Data from the window.

NOTE: Limitations of Debugger's Performance Measurements are the same as the 'AMIPerfRecordDump' application. For more info refer to PerformanceMeasurement.chm in PerformanceMeasurement eModule.

Working with Registers

Users can Open or Close the Registers View using:

- VeBMenu: Debug → Windows → Registers



- Or using HotKey: ***CTRL+SHIFT+R***
- Or using Debugger Toolbar Icon [AX]

The Registers view will only be updated when the Target is in a halted state. The Registers window contains two columns. *Name* - Register Name & *Value* - the current value of the Register in Hexadecimal

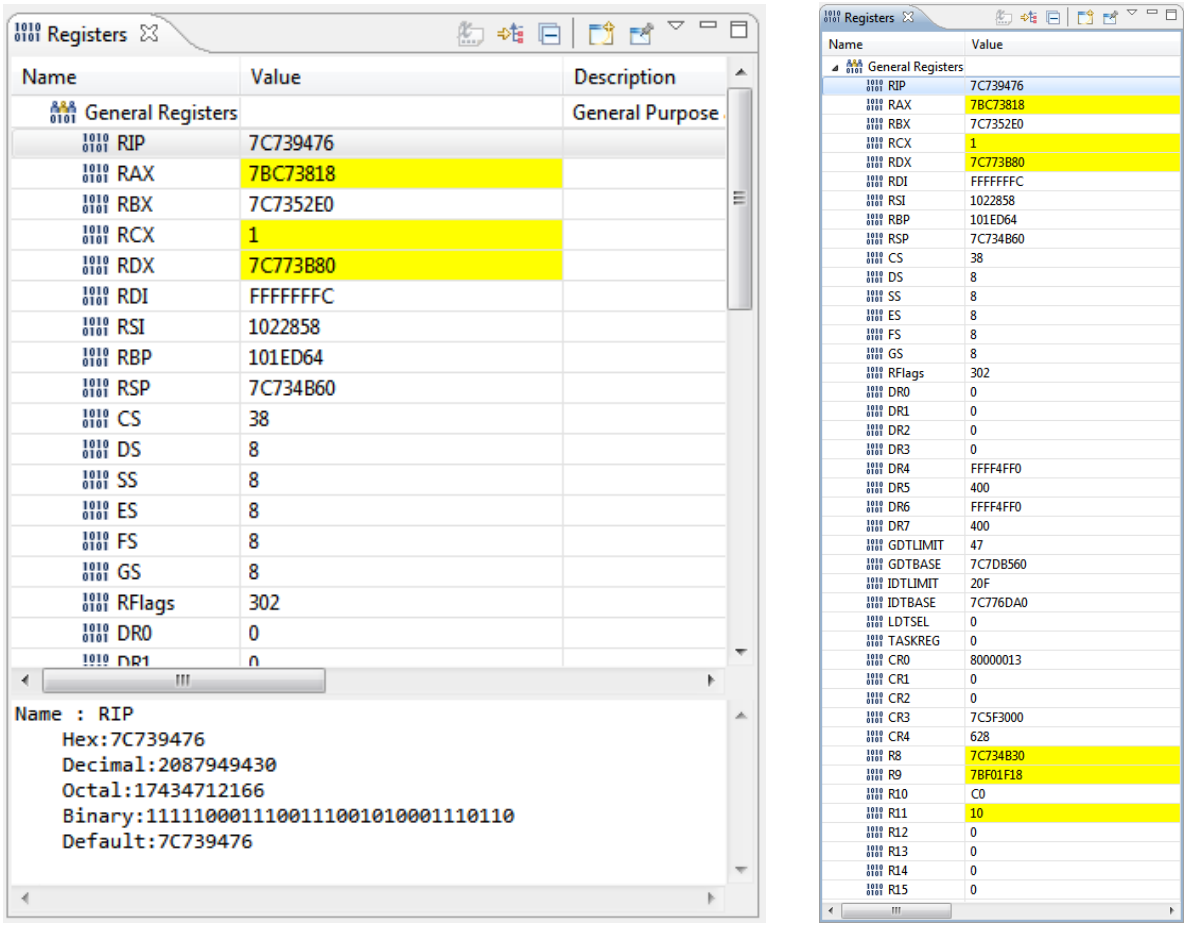


Figure: Registers View

To modify the value of a register, Click on the Value cell, and type a new value or edit the old value (Hex Only).

- To save the new value, press ENTER.

- To discard the new value before saving, press ESC.

On breakpoints or on stepping, the register values that have changed will be highlighted in yellow.

View Registers in Expressions View

It is also possible to add and view a Register's Value in the Expressions window. Right Click on the Register you want to add and select 'Watch' option, this will add the selected register to the Expressions view.

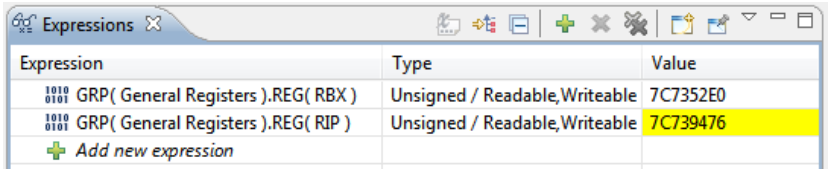


Figure: View Registers in Expressions View

Registers can also be manually added to the Expressions View by selecting 'Add New Expression' and entering

GRP(General Registers).REG(RegName)

where RegName is the name of the register as shown here

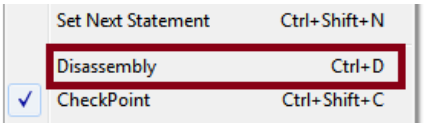
GRP(General Registers).REG(RAX)

Entries in the Register View can be selected to have more detailed information to be displayed in the Detail Pane.

Working with Disassembly

Users can Open\Close the Disassembly View using:

- VeB Menu: Debug→Disassembly



- Or using Hotkey - ***CTRL+D***

- Or using Debug Toolbar Icon [ ]

  Click Disassembly on the Debug menu to open the Disassembly window. Selecting this option when the Disassembly window is already open, will close it. The Disassembly window displays executable code in assembly language.

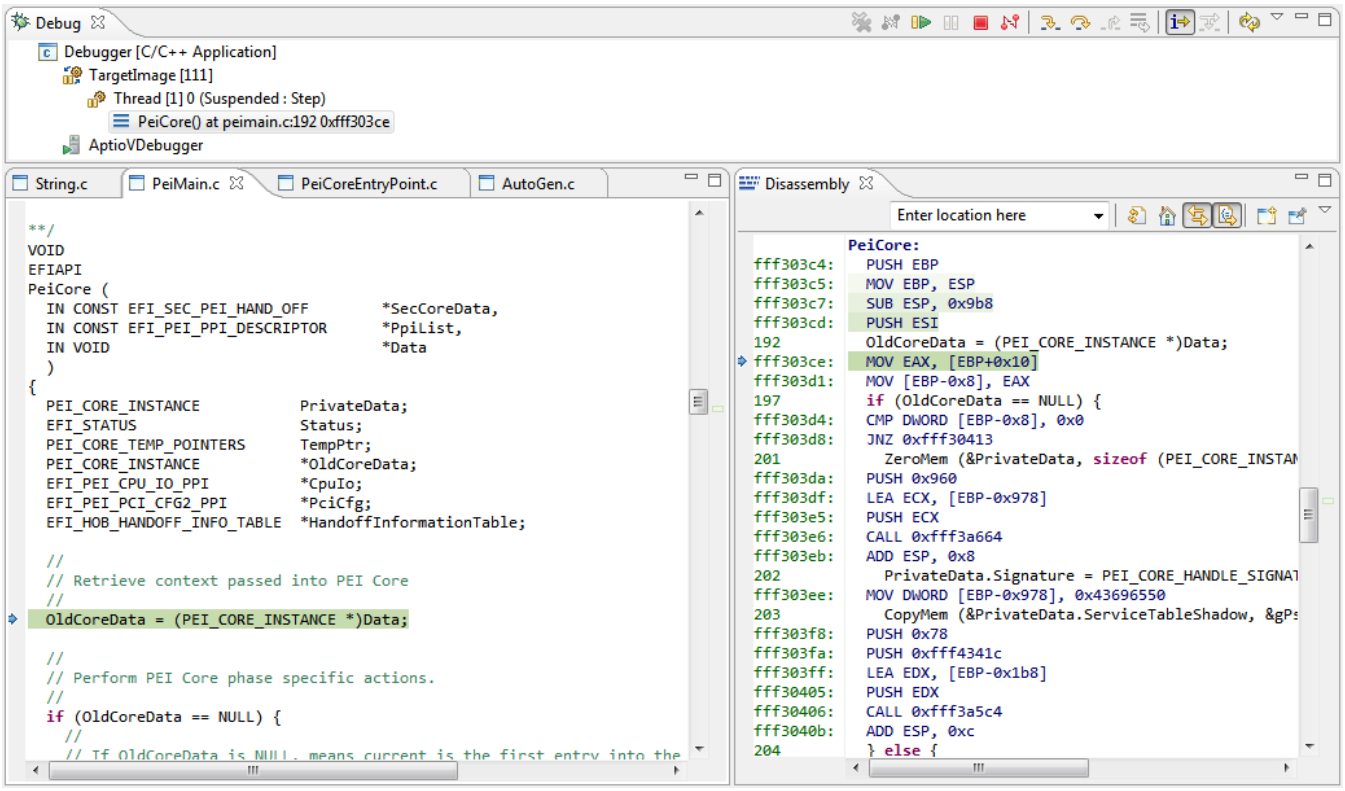
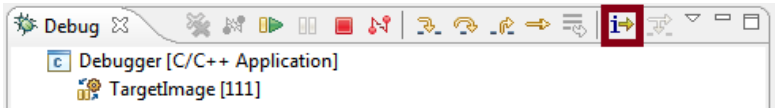


Figure: Disassembly View

In the Disassembly window, you can do the following:

Instruction Stepping Mode

To activate the Instruction step mode click on the **Instruction Stepping Mode** icon on the Debug view's toolbar as shown in following figure. The Debugger switches to the instruction stepping mode automatically when the Disassembly view has Focus.



This will change debug stepping behavior such that you will now step by processor instructions, rather than source lines.

***NOTE**: when Instruction Stepping Mode is enabled in the disassembly view, Step-over operations will behave as Step-into operations. Setting Breakpoints in the disassembly view may not be accurate.*

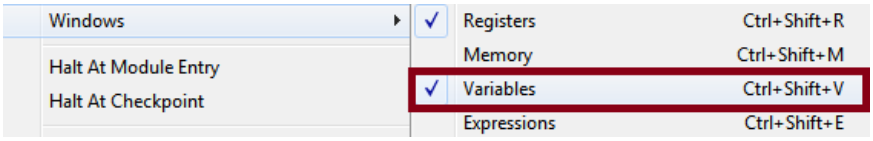
Disassemble Custom Memory

To disassemble a different section of memory, in the Enter Location box, type the address of the memory you want to disassemble. (You can press ENTER after typing the address) The Disassembly window displays code from the respective address.

Working with Variables (Local Variables Only)

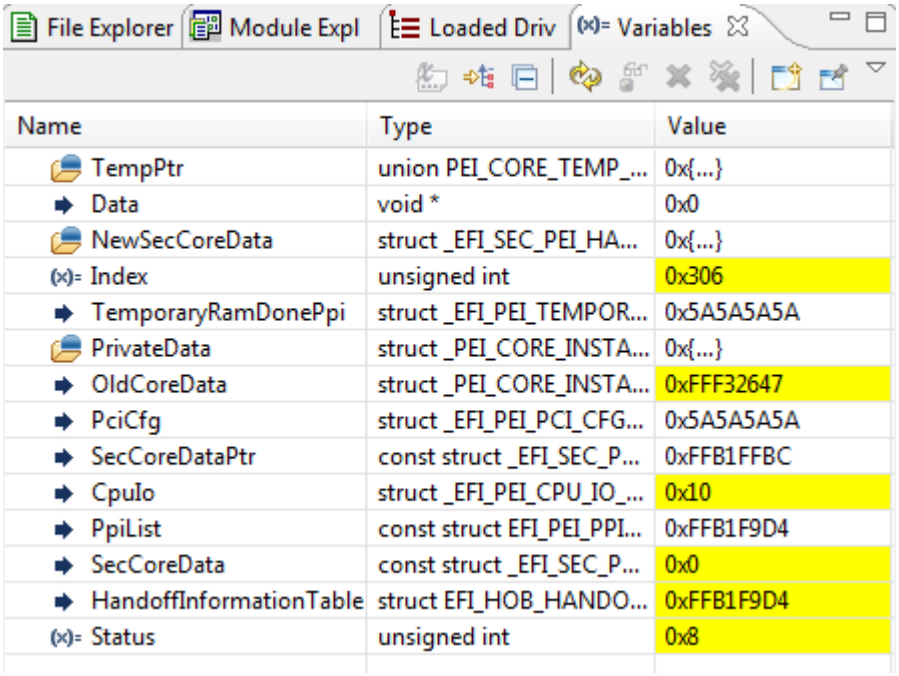
Users can access the Variables View by -

- Select : Select: *Debug →Windows→Variables*



- Or using Hotkey: ***CTRL+SHIFT+V***

- Or using the Debug Toolbar Icon [ ]



The variables window displays only the local variables in the current scope. The variable window contains three columns. The Name, Type and Value columns.

Name column displays the name of each local variable. If a variable has children (like with structures etc), the variable window will provide that variable in a tree view that can be expanded\collapsed.

Value column shows value of that variable in Hexadecimal format.

Type column shows the type of each variable.

Each variable is displayed in the format that is proper for its own data type. Data structures have their type names in the Type column.

NOTE: Variable View will auto-update on every step or halt.

If the value of a variable is changed since its last update, the value will be highlighted.

Editing a Variable

Users can edit the variable's value by selecting the value under Value Column and edit it.

NOTE: Only Hexadecimal inputs are allowed.

Additional Options

Users can see the Memory content and Edit it using the Memory window, to do that users need to right-click on the variable and select "view memory" option as in the picture below. This will use the 'Value' of the variable as the start address for memory display. For more information refer to Memory View.

Watch Variables

If a variable (Parent) is selected for Watch, that variable will be displayed in the Expressions View.

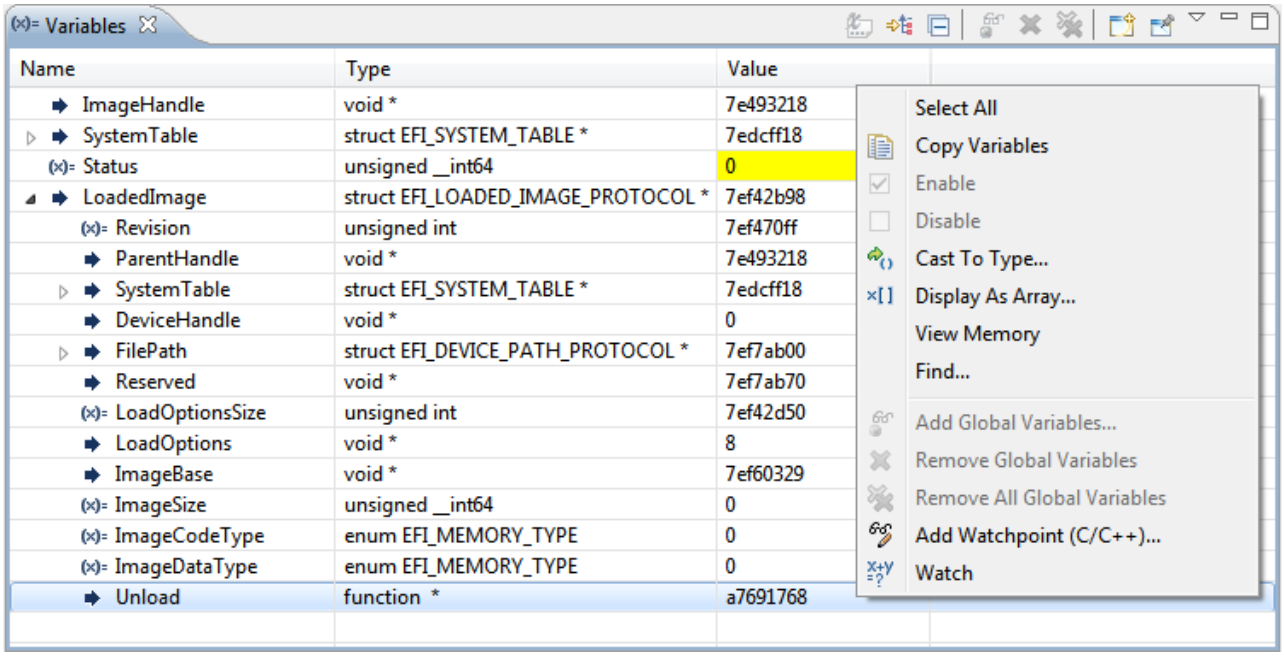


Figure: Variables View.

[NOTE] For the View Memory option, the value of the variable will be taken as the input address to show memory content.

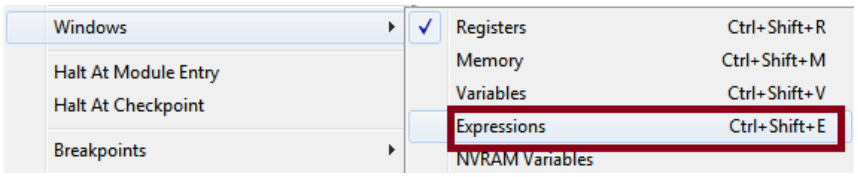
NOTE: To view the Address of the variable, type command 'var' in AptioVDebugger Console.

> NOTE: If variable window is not listing all variable, then make sure that OPTIMIZATION is off.

Adding Expressions (Local and Global Variables)

Users can access the Expressions View by -

- Select : Select: *Debug →Windows→Expressions*



- Or using Hotkey: ***CTRL+SHIFT+E***
- Or using the Debug Toolbar Icon [ ]

This will open the Expression view Window. Global Variables, Specific Variable Data can be inspected in the Expressions View. It acts as a watch window to observe global variables**.** The Expressions view is similar to the Variables view, except in Expressions the Variable needs to be entered by selecting ( ). Add new Expression ( ) allow us to add any new Expression or global variable or local variable to the Expression window.

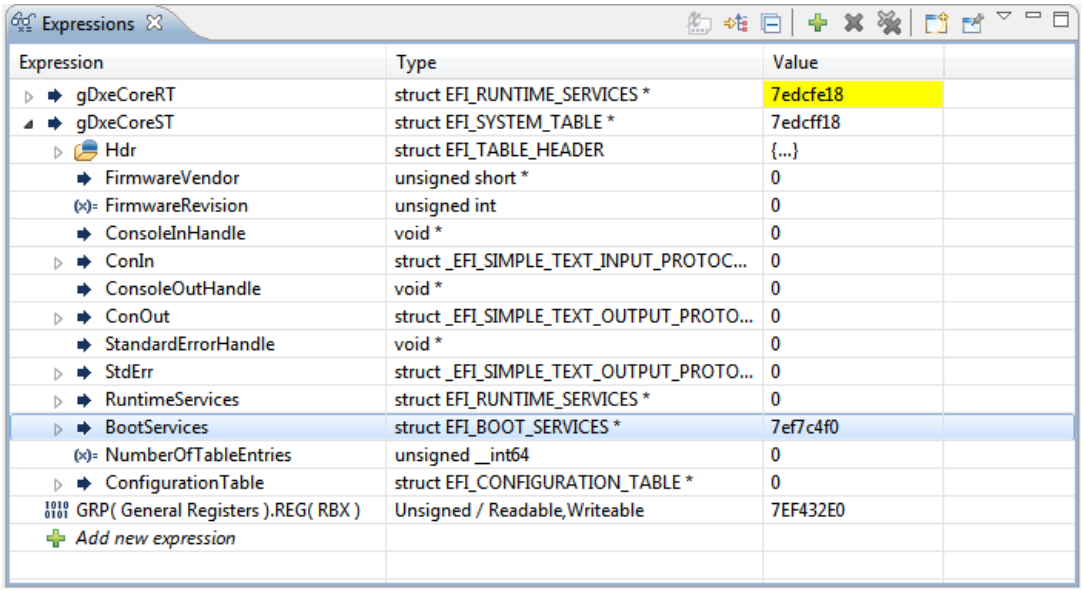


Figure: Expressions View

Any Watch's selected from Variable, Register window will be displayed in the Expressions View.

NOTE: The scope of the Global Variable is the Module's range, but Local variable scope is the function only.

Current debugger supports any combination of the following operators as part of the expression view

- - pointer operator

  Example:

  *a

  *a->b

*(a->b)

- . - Structure operator

  Example:

  a.b

  a.b.c

- -> - arrow operator

  Example:

  a->b

  a->b->c

- [] - array operators

  Example:

  a[b]

  a[1]

  a[b[1][c]]

- & - address operator

  Example:

  &a

  &a->b.c

  & a[b[1][c]]

- sizeof - sizeof operator

  Example:

  sizeof(a)

  sizeof(a->b.c)

  sizeof(a[b[1][c]])

The current limitations are:

- casting

  Any type casting is not supported.

- Arithmetic computations and logic

  Example:

  a+b

  10+1

  A || B

- sizeof(type name)

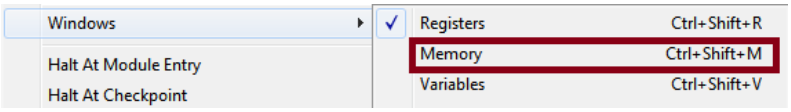
  Example

  sizeof(int)

Working with Memory View

Users can access the Memory View by -

- Select : Select: *Debug →Windows →Memory*



- Or using Hotkey: ***CTRL+SHIFT+M***

- Or using the Debug Toolbar Icon [ ]

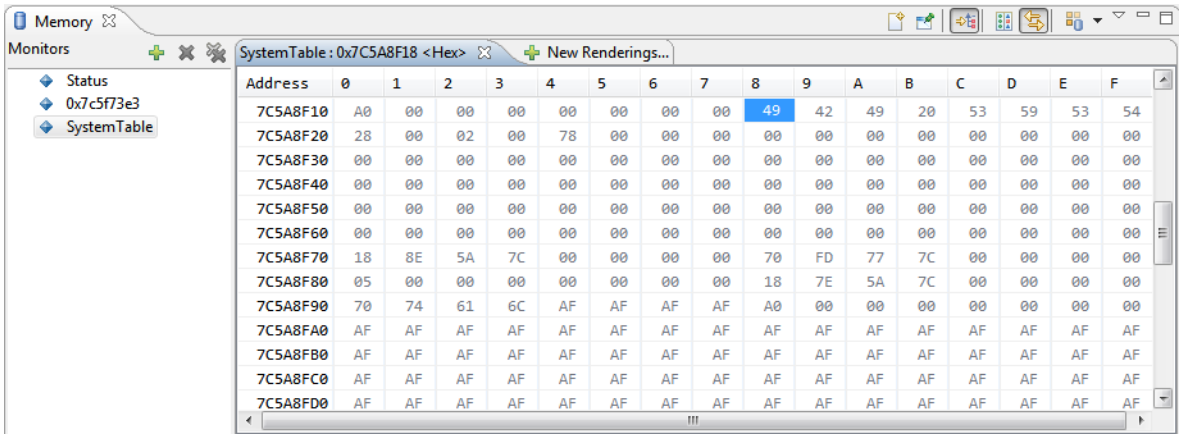


Figure: Memory View.

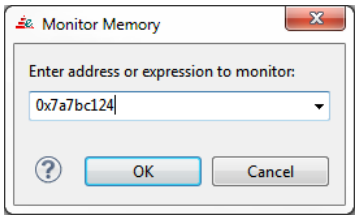
With the Memory view, you can look at the contents of memory at a specific address

To add a new memory monitor from the Memory view:

1. Click the Memory view **Add Memory Monitor** push button (✚). This button is located in the **Monitors** pane.

Or Right click on Memory View and select Add Memory monitor

2. In the Monitor Memory dialog box, enter the address or expression in the field (expressions must evaluate to an address). This entry need not be case sensitive while entering Addresses, but needs to be case-sensitive while using Expressions like Variable Names. Alternatively, if you have previously monitored the address or expression when debugging this application, choose it from the pull down list.



3. Click **OK**.

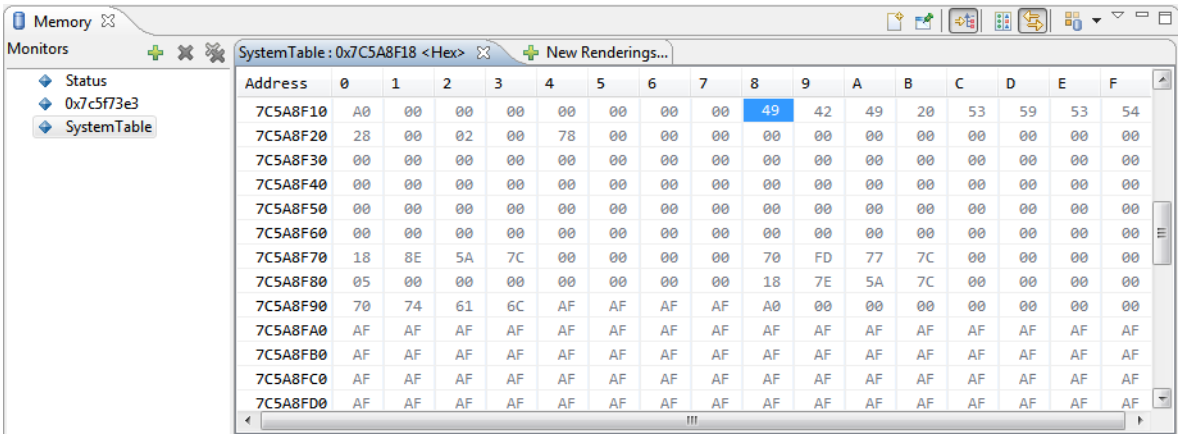


Figure: Memory View.

To write to memory, click inside the Memory window and type new data.

Right Click on the Memory monitor and, Remove Memory monitor will close the Memory monitor of specific address

Memory Monitor Inputs

The Memory monitor can accept Addresses (Hex and Integer), Variables, and Registers.

Addresses

If the entry starts with '0x' it is considered a Hexadecimal Address Input, not case sensitive

Eg: 0x7a7bc124 (or) 0x7A7BC124

If the entry starts with a Numerical (0~9), it is consideres as an Integer Input

Eg: 34572198

If the Entry is an Address, the Memory area from that address on target will be displayed in Memory View.

Registers

If the Entry starts with '$', it is considered a Register Input, not case sensitive

Eg: $RIP (or) $rip (or) $RAX (or) $cr4

If the Entry is a Register Input, the value of the register will be used as the Input Address and Memory area from that address on target will be displayed in Memory View.

Variables

If the entry does NOT start with "0x" or Numerical (0~9) or "$", it is considered as a Variable input

Eg: Status (or) SystemTable

If the Entry is a Variable Input, the value of the Variable will be used as the Input Address and Memory area from that address on target will be displayed in Memory View.

NOTE: If the variable name entered is not available, it will consider the value of the variable as a '0x00000000'.

NOTE: All Variable entries are Case-sensitive.

Rendering Options

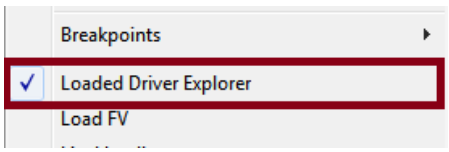
After adding the memory monitor, you can choose the memory format that you want to display in the **Renderings** pane.

1. Click the Memory view **Add Rendering(s)** push button (). This button is located in the **Renderings** pane.
2. Select the memory rendering that you want to display for the memory monitor and click **Ok**

Loaded Drivers Explorer

Users can toggle the Loaded Driver Explorer by -

- Select: *Debug → Loaded Driver Explorer*



- Or using the Debug Toolbar Icon [ ]

The Loaded Driver Explorer Lists the Drivers in the loaded order they are loaded on the target side.

Users can view the Loaded driver and its associated files in a Explorer tree format (shown below). On expanding the Driver tree, users can open any file associated with a driver by double clicking on the file.

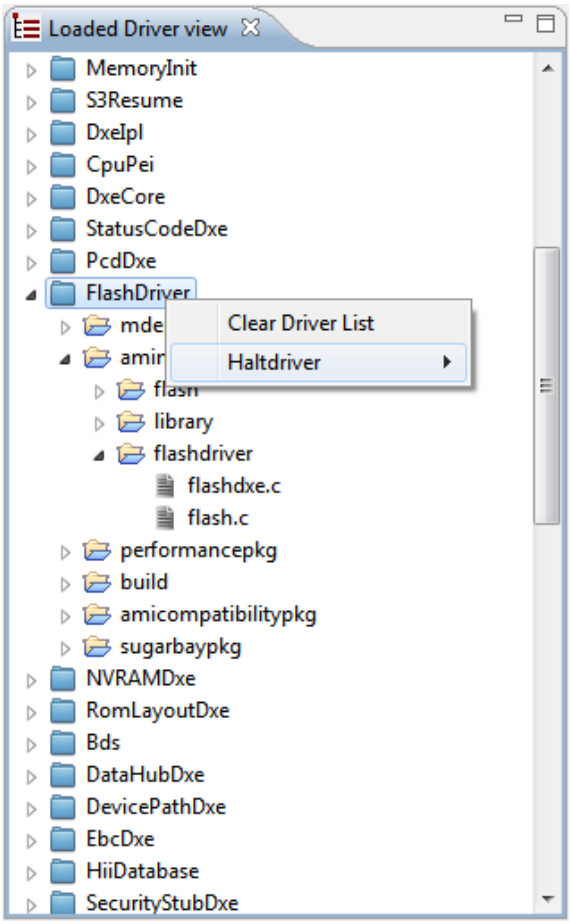
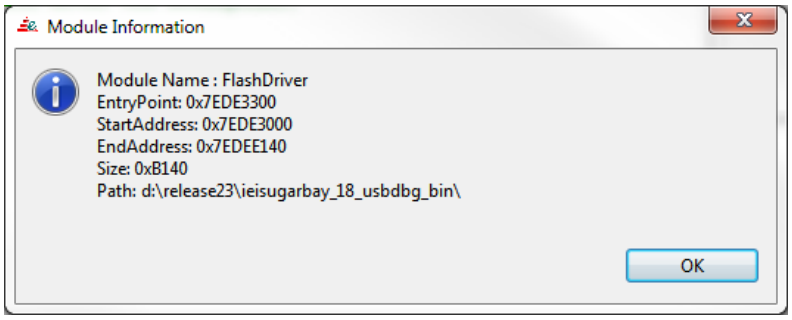


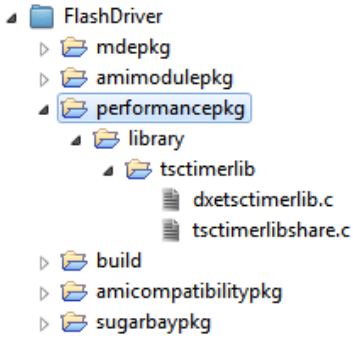
Figure: Loaded Drivers View.

Options

Module Information - Users can view the driver details of a loaded driver by double clicking on the Driver name. The details of that driver will be displayed in a popup window.



Auto Folder Expand - Double clicking on any node will fully expand that node.



Right Click Menu - In addition, users can clear the loaded driver list or enable the halt at ALL or Specific driver from loaded driver explorer, simply by right click on any driver then following option will come

- Clear Driver List
- HaltDriver

Clear Driver List - This option will clears the loaded driver explorer from loaded driver explore.

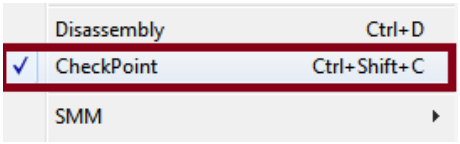
HaltDriver - If you want to set halt at driver Entry then click on Halt at Driver option. It will give three option.

- Halt At This Driver Entry – This is for setting halt at the selected driver Entry only.
- Halt At All Driver Entry – This is to enable Halt on All Driver's Entry.
- Halt Driver Off –This is for disabling all halts at Driver Entry.

Checkpoint View

Users can toggle the Checkpoint View by -

- Select: Debug -> Checkpoint

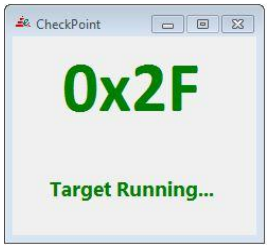


- Or using Hotkey: ***CTRL+SHIFT+C***

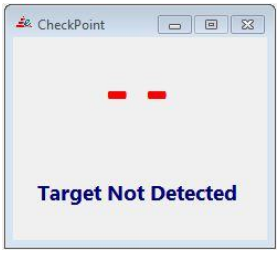
- Or using the Debug Toolbar Icon [ ]

The Checkpoint window will display the Progress code Checkpoints while Target is in Running\Booting state and will display the Port 80 Checkpoint When Target is in Halted State. Checkpoints when halted will be displayed in RED and while running will be displayed in GREEN.

NOTE: While the target is in a Running state, ONLY the Statuscode\Progresscode checkpoints will be shown in the Checkpoint view. Direct write to IO port 80 cannot be shown when in Running state.



When Target is not connected checkpoint window will display with message as “Target Not Detected”. Below screenshot showing when target not detected.

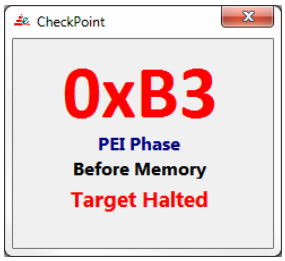


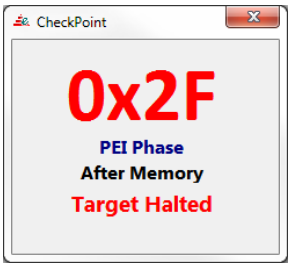
Additional Target Info

When the Target is in halted state, the Checkpoint window will display the state of target, phase of Bios (such as PEI phase, DXE phase or SMM phase) and display the relevant info in each phase,

such as in PEI phase -

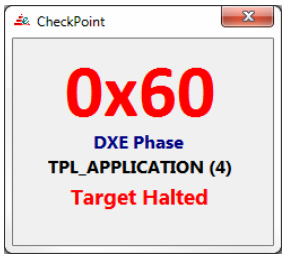
current execution is before memory or after memory





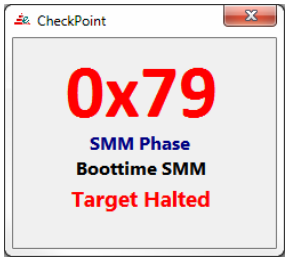
In DXE Phase -

Value of the Current TPL (Task Priority Level)



In SMM Phase -

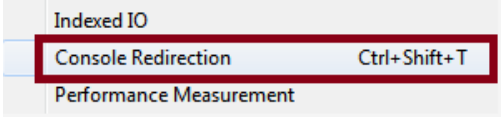
Boottime SMM or Runtime SMM



Console Redirection Support

Users can toggle the Console Redirection View by -

- Select: *Debug → Windows → Console Redirection*



- Or using Hotkey: ***CTRL+SHIFT+T***

- Or using the Debug Toolbar Icon [ ]

The **Console Redirection** view captures the video redirection if supported by the target. Allows the users to view Setup, Shell etc without the need for an external display device connected to the Target. The View allows bi-directional communication over keyboard. Whatever changes made in the console view reflects in the target system as well.

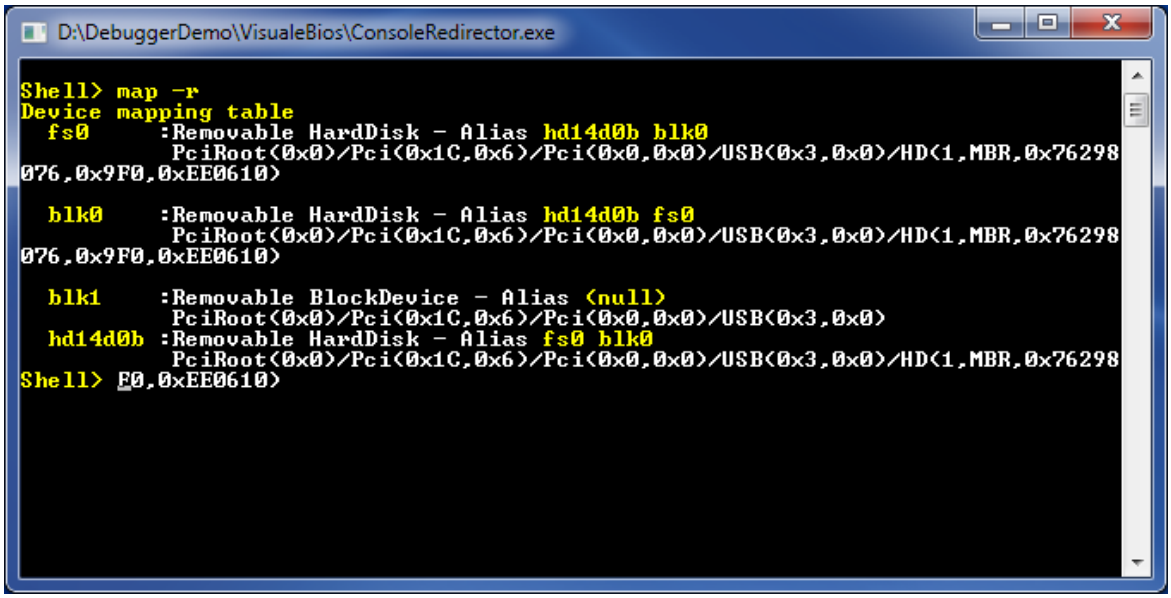
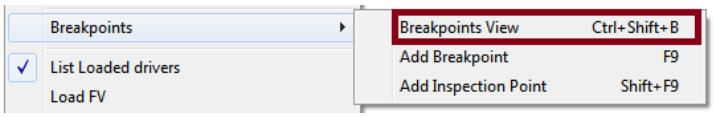


Figure: Console Redirection

Using Breakpoints

The User can Open\Close the Breakpoints View:

- Using VeB Menu Item: *Debug →Breakpoints →Breakpoint View*



- Or using Hotkey: ***CTRL+SHIFT+B***

Or using the Debug Toolbar Icon [ ]

Breakpoints view displays the breakpoints and inspection points set for the Current Debug session.

This window displays the breakpoint's type, location of breakpoint, Hit count and Inspection Options and Conditions with count info.

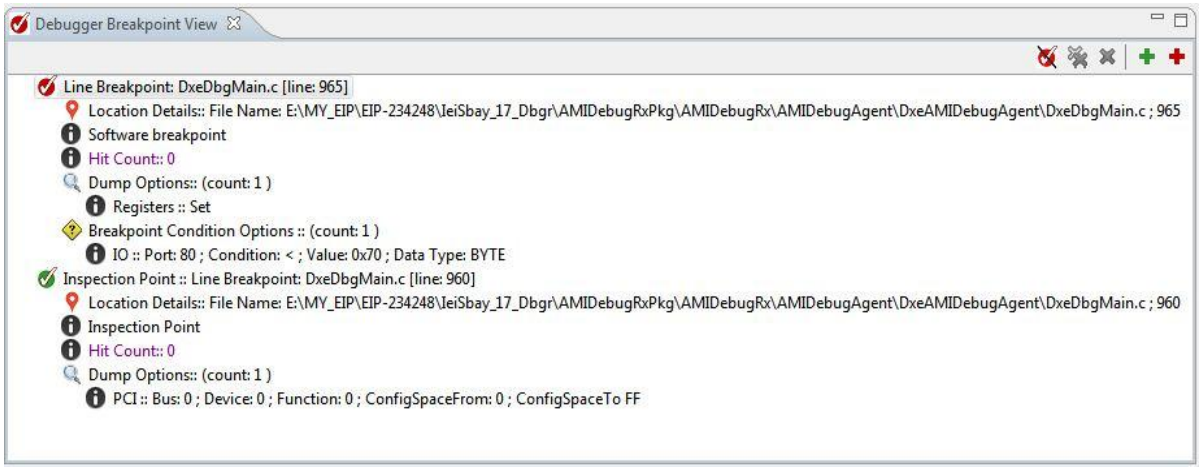
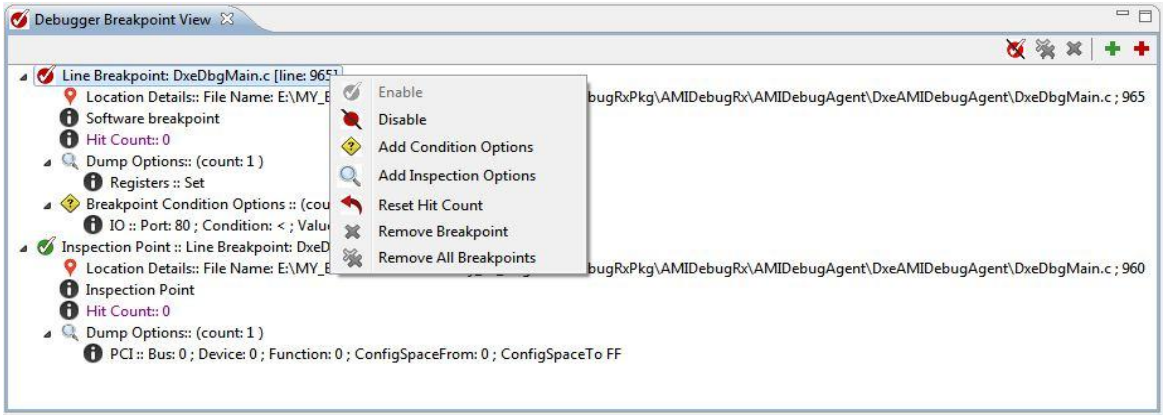


Figure: Breakpoints View

User can disable or enable a particular breakpoint or all breakpoints. User can add conditions, inspection options and reset the hit count for any breakpoint.



User can use different button available in breakpoint view to Add a New Breakpoint or Inspection point. Buttons in the breakpoint view and their usage is explained below.

 - Used to Add a New Breakpoint. Clicking on this button will open the "Add Breakpoint" window. For More Info see Adding Custom Breakpoints.

 - Used to Add a New Inspection point. Clicking on this button will open the "Add Inspection Point" window. For more Info see Adding Inspection Options.

 - Used to disable all the set Breakpoints and Inspection points.

 - Used to Remove all Set Breakpoints and Inspection points.

 - Used to Remove selected Breakpoint or Inspection point

Add Breakpoint View

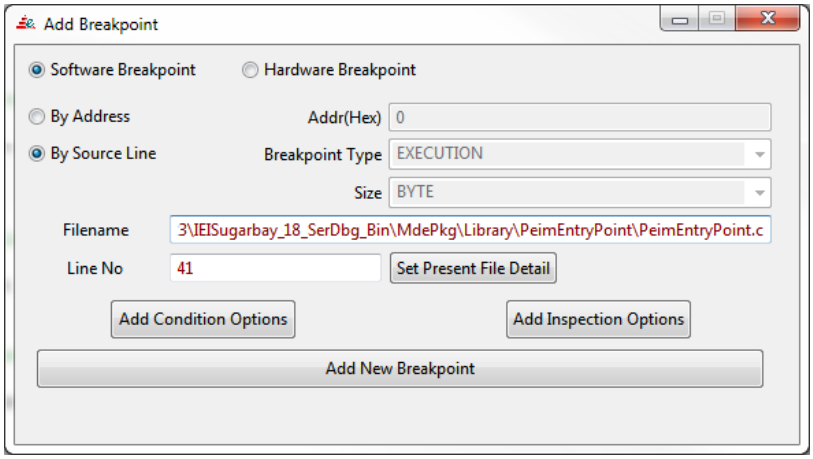
Users can add a new breakpoint by -

- Select: Debug → Breakpoints→Add Breakpoint

- Or using Hotkey: ***F9***

- Or using the Debug Toolbar Icon []

Adding a new breakpoint will open the Add breakpoint window

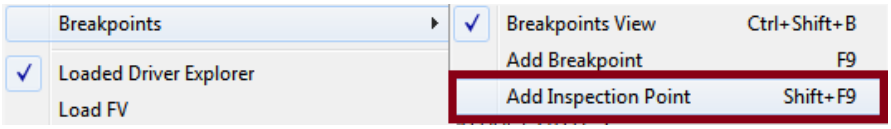


To add a new Hardware or Software breakpoint refer to Adding Custom Breakpoints.

Add Inspection Point View

Users can add a new Inspection Point by -

- Select: Debug → Breakpoints→Add inspection point



- Or using Hotkey: ***SHIFT+F9***
- Or using the Debug Toolbar Icon [ ]

Inspection Point is like breakpoint. Using Inspection Point View, user can see and dump the requested information such as PCI data, IO data, Variable data, context Register data and memory data to console window also the same could be log into a file at Inspection Point.

In Inspection Point Window, user can select the inspection point either by selecting “By Source line” or “By Address” option.

By Source line

Enter the file name and line number into corresponding text boxes to set inspection point using “By Source Line” option.

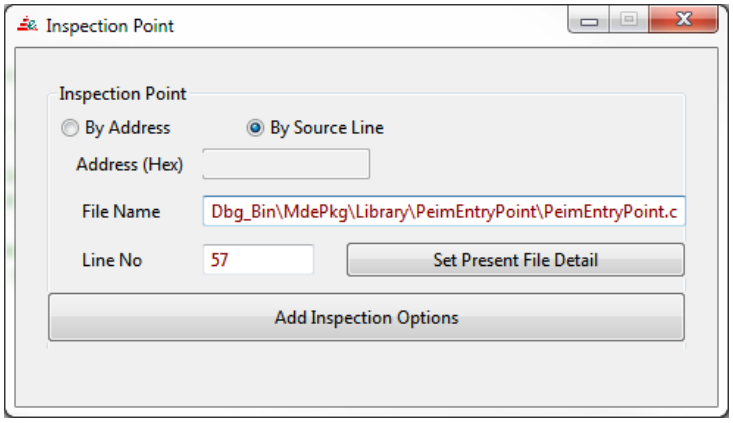


Figure: Inspection point by source line

By Address

Enter the address into “Address” text box at which user wants to inspect data.

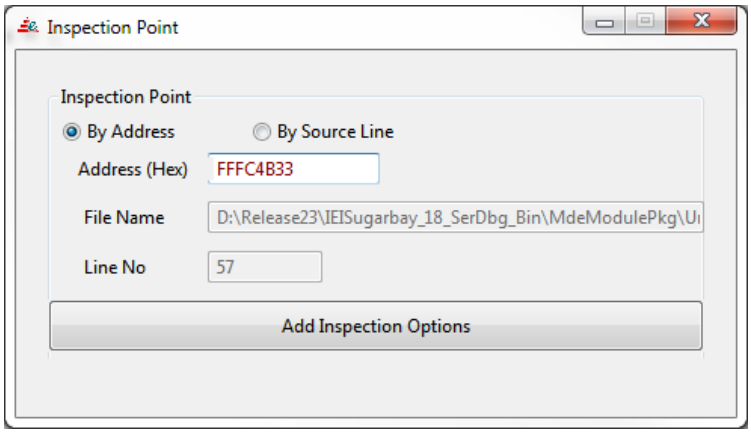


Figure: Inspection point by Address

Click on “Add Inspection Point” button. This will open Dump Option window with different options to dump different data such as PCI device option for dumping PCI configuration data at inspection point.

Select Option, which you want to dump the information and provide corresponding data.

Click on “Add” button.

User can see the dumped information into AptioV console window. Also this information automatically saved into inspection.txt file

Dump Option window

This window consist of different options to dump different data at inspection point. User can select one or more option. Following information user can dump at inspection point.

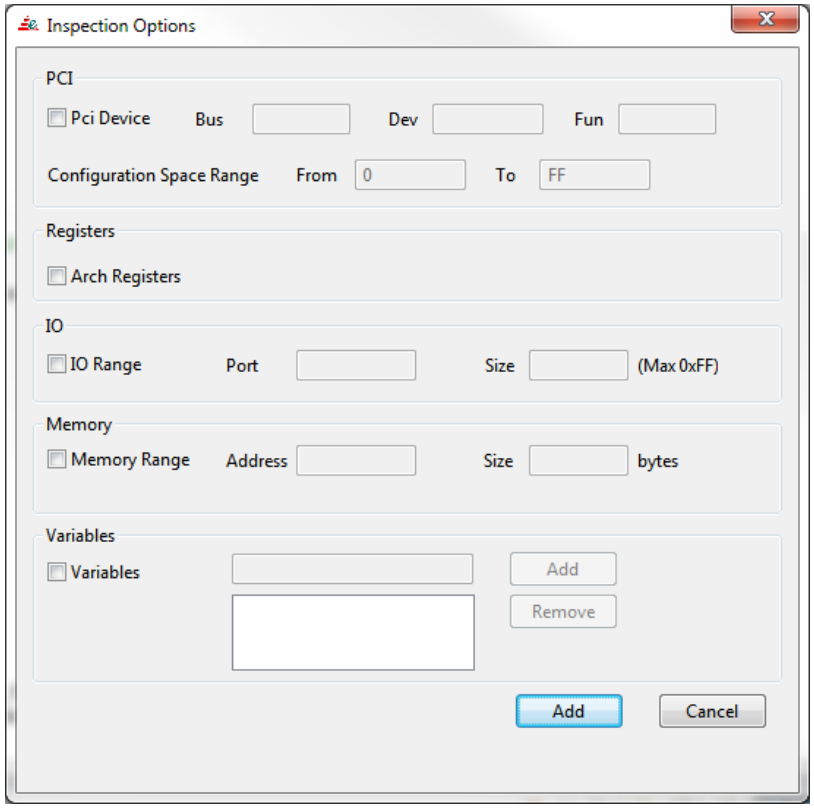


Figure. Dump Option window

PCI Device:

This option used to dump specified PCI configuration information at inspection point. In order to select this option, user has to provide Bus, Device, Function number and config space range in its corresponding text box.

Variable:

This option used to dump specified variable information at inspection point. To select this option, user has to provide variable name.

User can add or remove the variable from dumping by clicking on Add or Remove Button.

Inspect All Registers:

This option is to dump all register data information at inspection point.

IO Port:

This option used to dump specified IO port information at inspection point. To select this option, user has to IO port number and size of data, which want to dump.

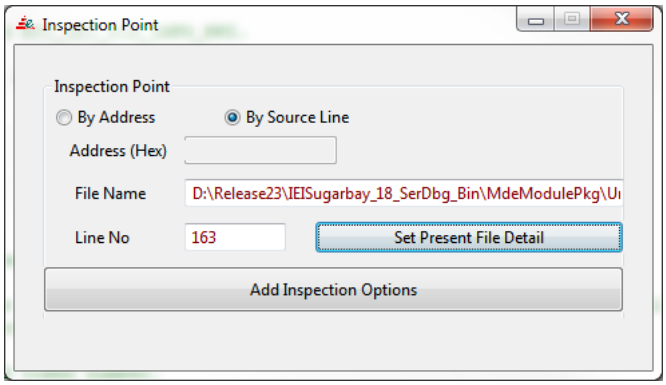
Memory Range:

This option is to dump data present at specified memory. To select this option, user has provide Address and Size in the corresponding text box.

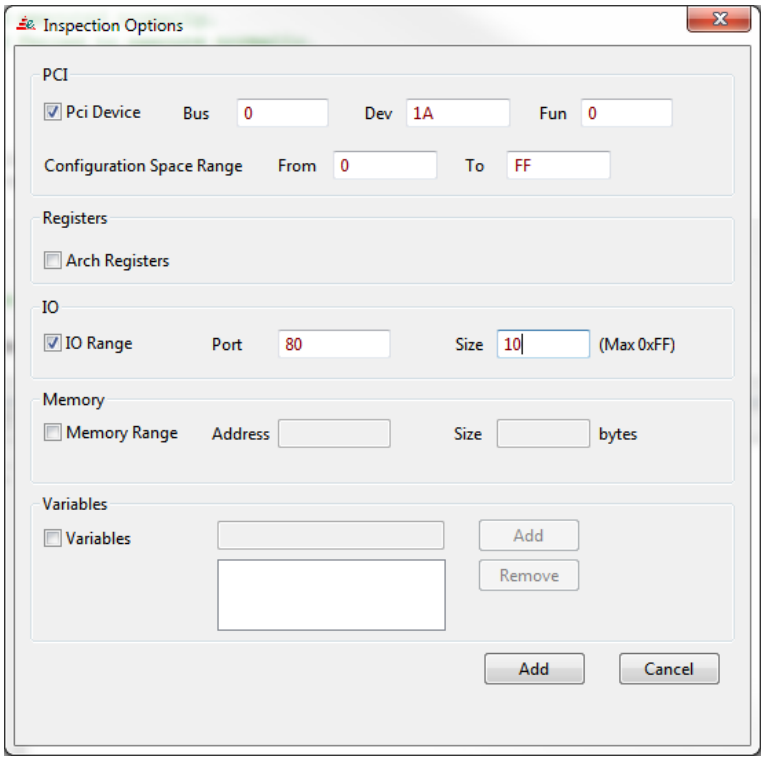
For example:

If we want to dump some information says, PCI configuration of Bus =0, deivce =1A, function=0 with Config space range from 0 to 0xff and IO port data on execution of line number 163 in the Pcd.c file then we will use inspection point feature as follow.

First, we will select the By Source Line option and providing the line number and location of file with name of file.

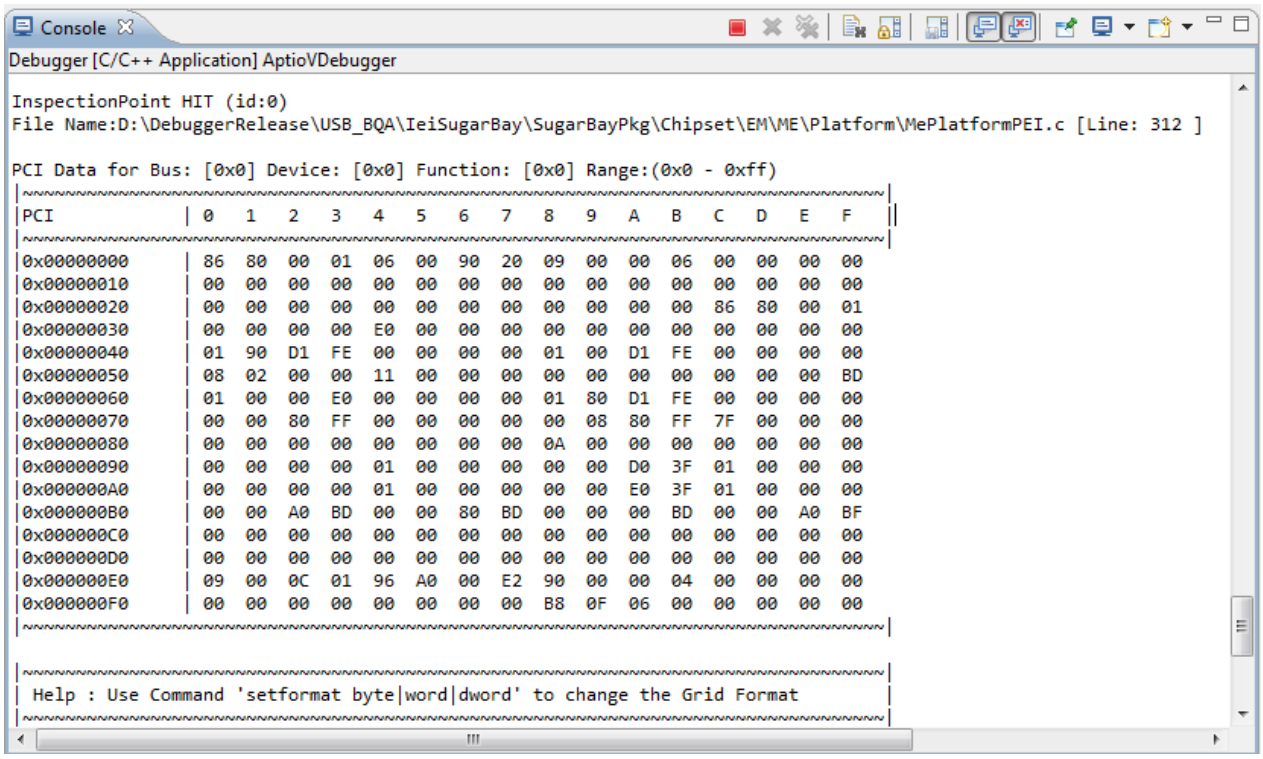


Then we will click on “Add Inspection Point” it will open Dump Option window. We will select PCI and IO port providing data bus=0, device=0, function=0 with config space range from 0 - 0xff and port 80 with size 10.

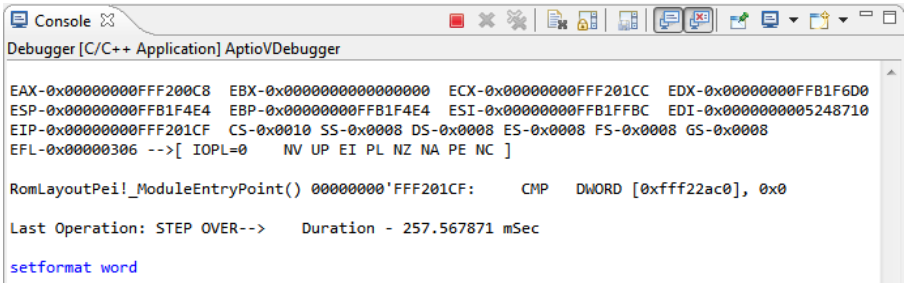


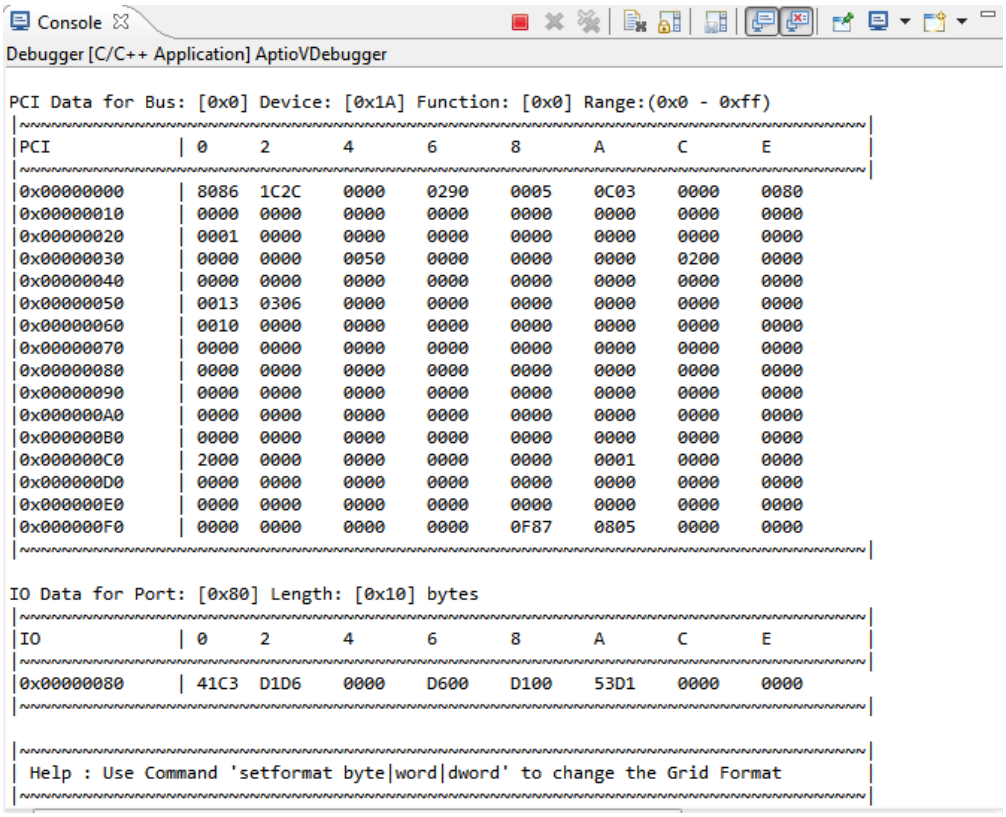
Click on “Add” button.

We can see dumped information in AptioV Debugger Console window. Also this info will be dumped into inspection.txt file, found in the VeB root folder.



User can change the displayed format of the Grid (byte/word/dword) by using the below command in AptioVDebugger Console window.





In addition, user can edit existing inspection point using breakpoint view simply by right clicking on the inspection point and selecting 'Edit Inspection Option'. For more detail See Breakpoints View.

☛	*Note: At Inspection Point, target will not halt the execution state unlike breakpoint.*

BreakPoints - Setting & Usage

Setting a breakpoint or Inspection point at Boot time can be done using any of the methods mentioned below. First let us understand what is the difference between a breakpoint and an inspection point.

Understanding Breakpoint vs Inspection Point

Breakpoint - is a point at which the Debugger will halt the execution on the Target.

* If the Breakpoint has Conditions associated with it (See Conditional breakpoints), the Breakpoint will halt the execution only if all the Conditions are satisfied.

* If the Breakpoint has Inspections associated with it (See Inspection Points), the Inspection Options will be displayed on the AptioVDebugger Console window and the execution will be halted.

Inspection Point - is a point at which the Inspections options are displayed on the AptioVDebugger Console window, but the execution of the Target will NOT be halted.

Breakpoint @ Compile Time (In Code)

Setting a breakpoint in the code can be done by adding any of the below mentioned code in the place where the user wishes the target to break –

- __debugbreak(); Or
- __asm int 3;

> ☛ *NOTE: __asm int 3; is valid only in 32 bit code. In x64 the Assembly instructions need to be in an .asm file.*

Breakpoint @ Boot Time

Using Editor Pane (Source & Disassembly Views)

One way to set a breakpoint on any source or Disassembly line, is by right clicking on the Left Pane of Source View or Disassembly view and selecting Toggle Breakpoint', as shown below.

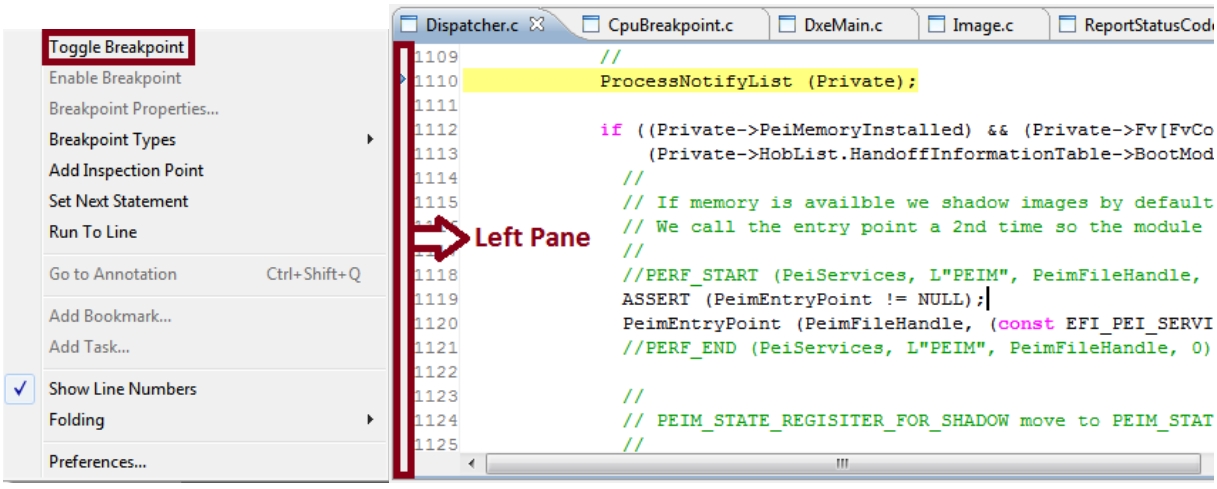


Figure: Source View - Toggle Breakpoint

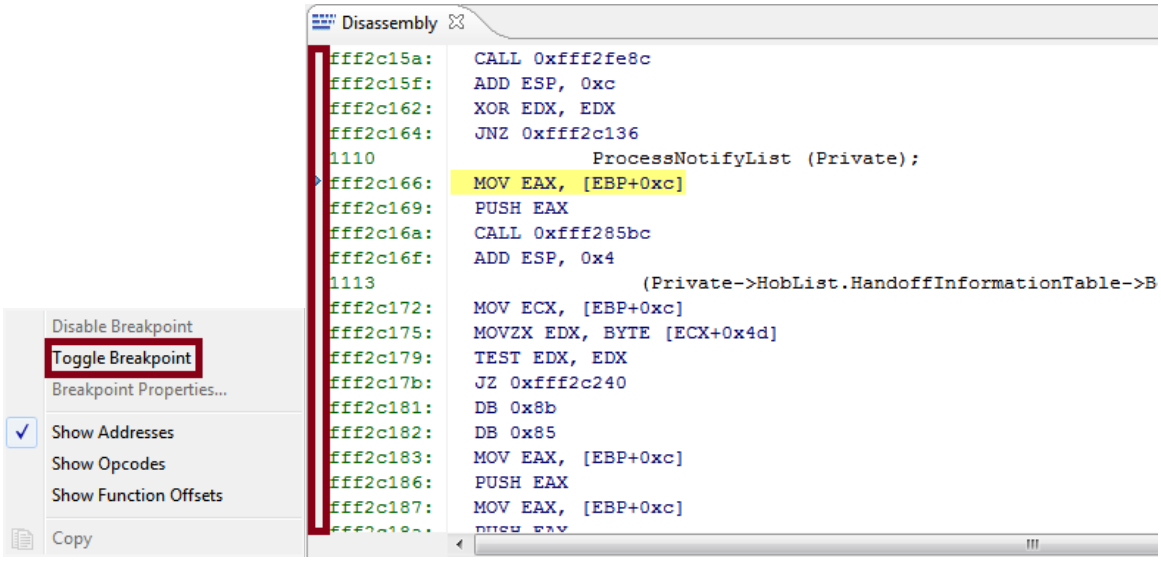


Figure: Disassembly View - Toggle Breakpoint

Using this method the breakpoints set in PEI phase before Memory initialization will be automatically set as Hardware breakpoints. Software breakpoints will be set only after memory is initialized.

Another way to quickly Toggle a breakpoint is by “double clicking” on the left pane, next to the desired source line in Source View or Disassembly View.

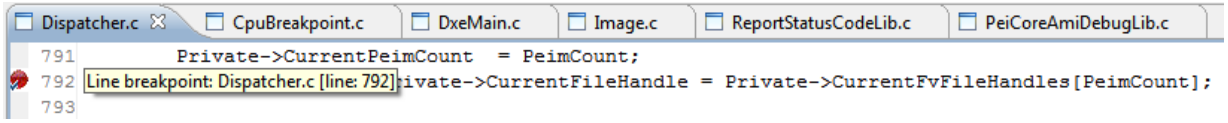


Figure: Line Breakpoint in Source View

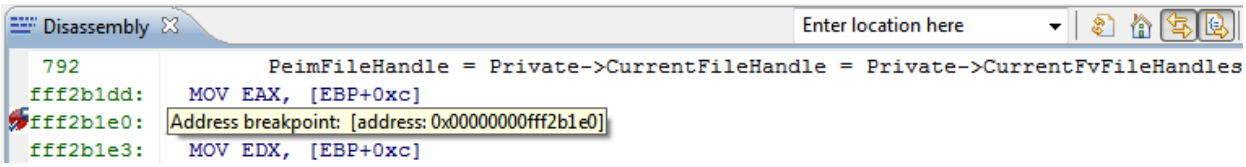


Figure: Address Breakpoint in Disassembly View

NOTE: When setting breakpoints in PEI phase before Memory is initialized, Debugger will automatically set a Hardware Breakpoint using the Debug registers, it is recommend that users do not set more than 3 breakpoints since debugger uses 1 breakpoint for internal stepping purposes.

Setting a Custom Breakpoint:

Users can set custom Breakpoints (HW or SW) or Custom Inspection Points, to halt on a known desired location in the BIOS boot process, for detailed info refer to Adding Custom Breakpoints.

Using Halt at Driver Entry

The user can configure the Debugger to break on the entry point of a Driver while it is being loaded. The user can set the Debugger to halt on the desired entry point using the following options

'ALL' - Debugger will halt on all the Driver's entrypoint (in PEI|DXE|SMM)

'NEXT' - Debugger will halt only on the next loaded driver entry.

'DRIVER NAME' - Debugger will halt on the Entry point of the driver name provided by the user. Multiple driver names can be setup to halt.

Refer to the Halt Driver View for detailed info and usage examples.

Using Halt at Checkpoint

The user can configure the Debugger to break on the occurrence of a Checkpoint. The user can set the Debugger to halt on the desired checkpoint using the following options:

'ALL' - Debugger will halt on all the Checkpoints

'NEXT' - Debugger will halt only on the next loaded Checkpoint .

'CHECKPOINT' - Debugger will halt on the Checkpoint provided by the user. Multiple Checkpoints can be setup to halt.

Refer to the Halt Checkpoint for detailed info and usage examples.

Initial Breakpoints

The Debugger will automatically break on two initial breakpoints by default,

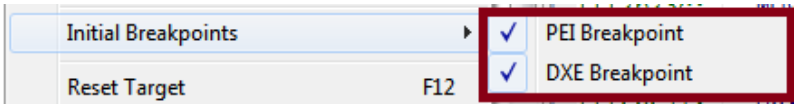
'Pei Initial Breakpoint' - Will halt on the Entrypoint of PeiCore.

'Dxe Initial Breakpoint' - Will halt in DxeCore after Dxe Debugger initialization.

These Initial breakpoint can be toggled using:

- VeB Menu: Debug → Initial Breakpoints →PEI Breakpoint

  Debug → Initial Breakpoints →DXE Breakpoint



- Or using the Debug Toolbar Icons [PEI BP DXE BP]

  The Initial Breakpoints will be enabled by Default.

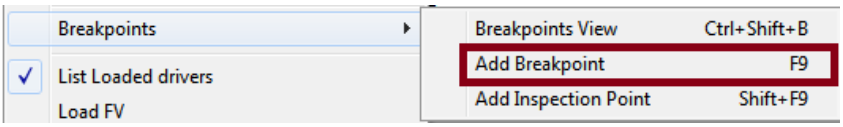
  PEI Initial breakpoint, when disabled also will break once at PeiCore Entry on first boot.

  DXE Initial Breakpoint will happen only when the option is enabled.

Adding Custom Breakpoints

In order to add a new custom breakpoint or condition on customized breakpoint user can do by using:

- VeB Menu: *Debug → Breakpoints → Add New Breakpoint*



- Or using the Hotkey: ***F9***

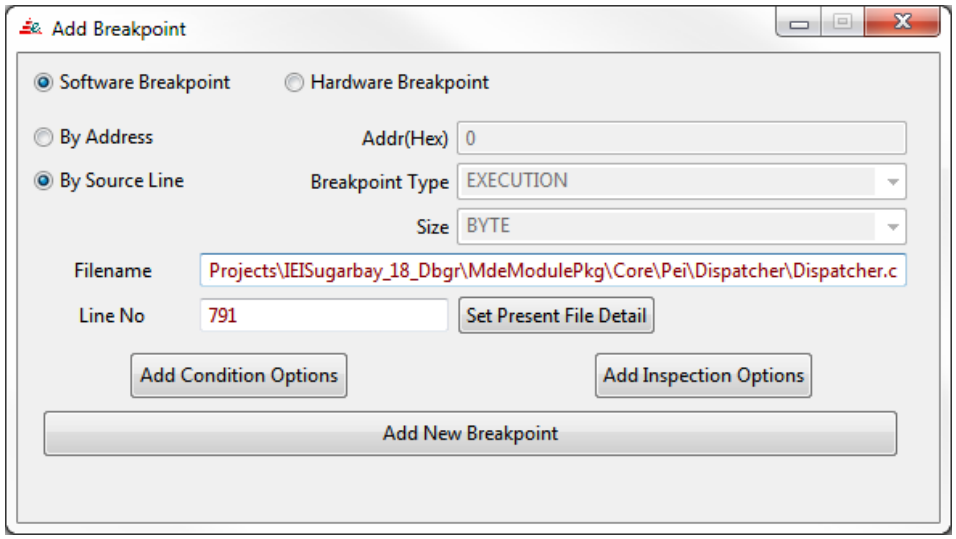
- Or Selecting the Debug toolbar Icon [ ]

This will open the "Add Breakpoint" window which contain options to

Set Hardware/Software breakpoint

- With Conditions (Optional)

- With Inspection Options (Optional)

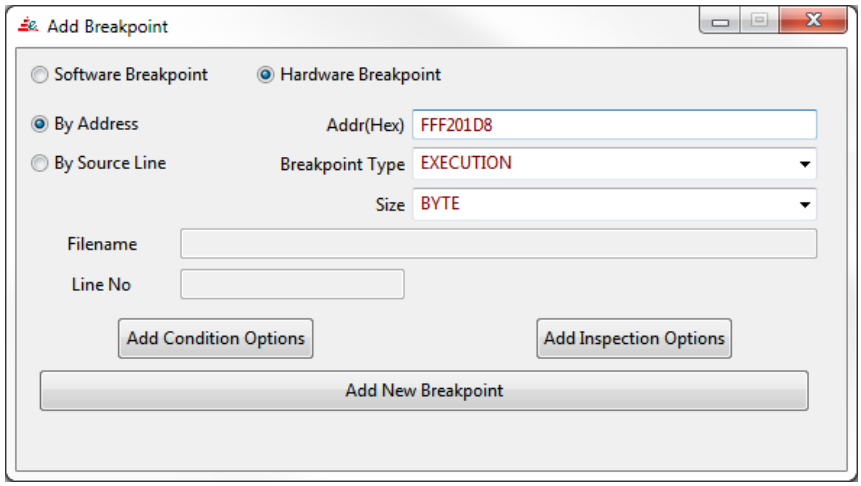


Adding a Custom HW Breakpoint

Custom HW break points can be set by selecting the hardware breakpoint option in the Custom breakpoint view. Hardware Breakpoint can be set By Address or By SourceLine.

HW Breakpoint: @ Address

When setting Hardware Breakpoint by Address, the 'By Address' radio button needs to be selected, The Address, breakpoint type and the size need be entered. Then click on "Add New breakpoint" window. These break points are triggers whenever the Address specified is being accessed based on the breakpoint type specified.



The supported breakpoint types are

EXECUTION – This type of breakpoint will trigger on execution of specified memory address, that is when the specified memory address is the current Instruction Pointer (IP).

DATA WRITE – This type of breakpoint will trigger only when writing data at specified memory address or changing the content present in specified memory address.

IO READ ONLY – This type of breakpoint will trigger when reading from specified IO port address happen.

IO WRITE ONLY – This type of breakpoint will trigger only when writing data to the specified IO Port address.

DATA READ AND WRITE NO EXEC – This type of breakpoint will trigger whenever reading or writing of data at specified Memory address happens, But not when the Address is the Current Instruction.

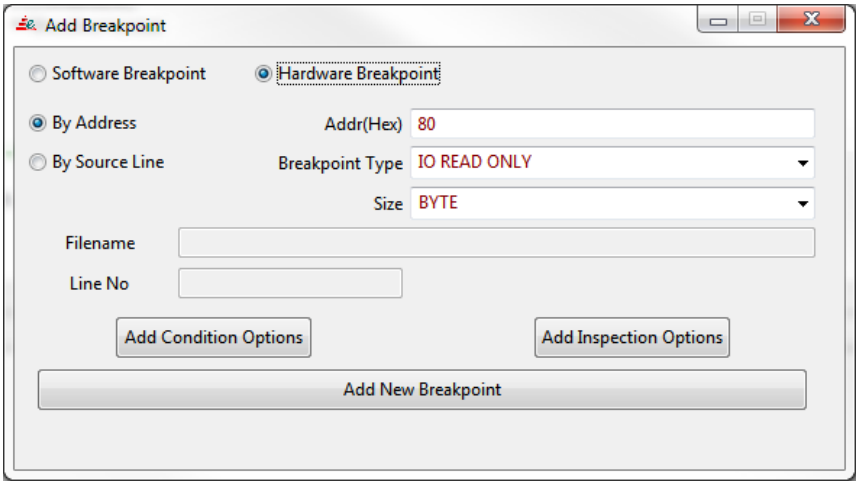
User can see the breakpoints set in breakpoint view window.

Users can also add the breakpoint from Breakpoint view window by clicking on Red plus symbol Refer Breakpoint View.

IO HW Breakpoint Example:

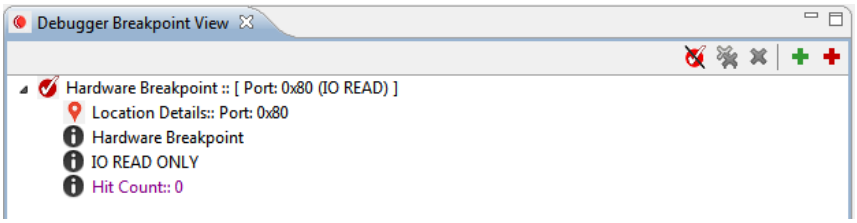
To set an IO READ ONLY Hardware breakpoint at IO Port 0x80

Select hardware breakpoint, address as 0x80, Breakpoint type as IO READ ONLY and size as BYTE.



Click on “Add New Breakpoint” button.

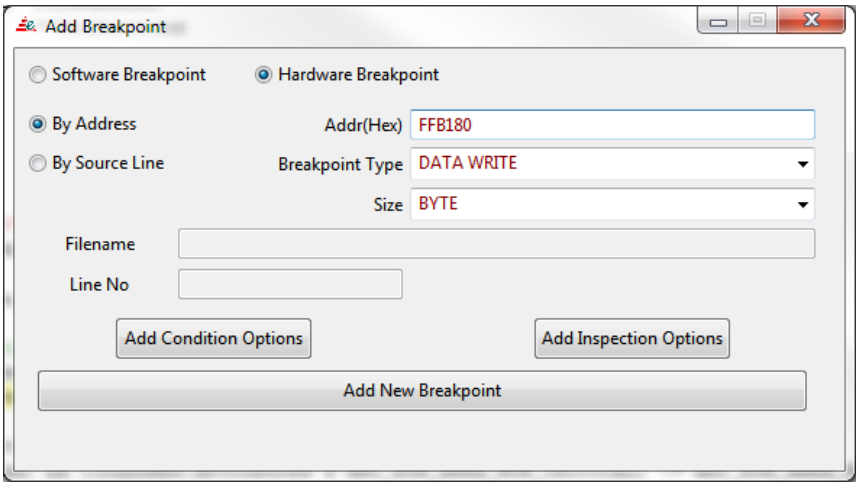
You can see the breakpoint is set in breakpoint view window.



DATA WRITE HW Breakpoint Example:

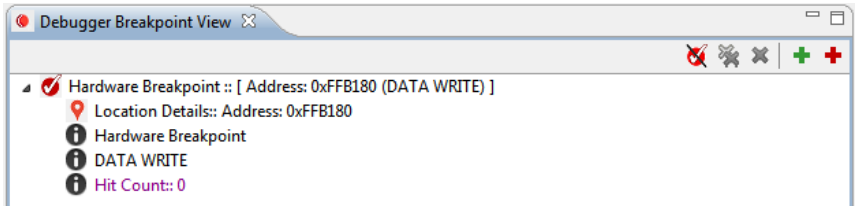
If you want to set the breakpoint, which has to trigger whenever someone changing the first two byte content at memory address 0x00FFB180 then follow the below steps.

Select hardware breakpoint, address as 0x00FFB180, Breakpoint Type as DATA WRITE and size as WORD.



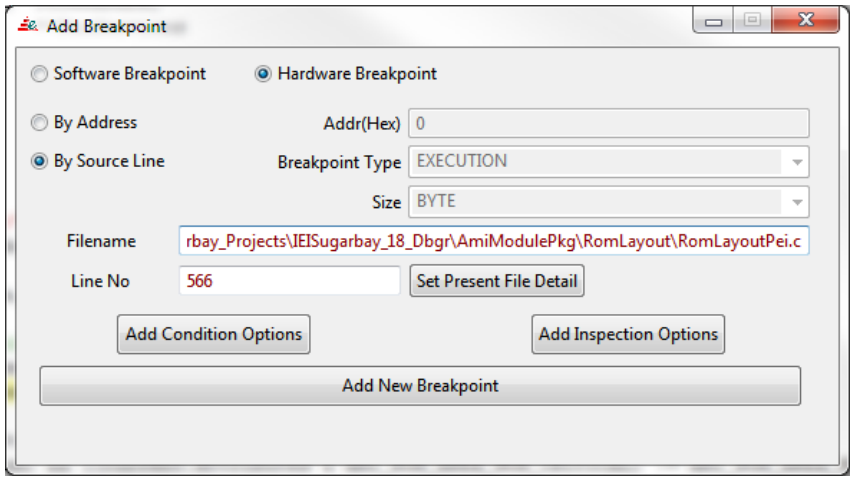
Click on “Add New Breakpoint” button.

You can see the breakpoint is set or not in breakpoint view window.

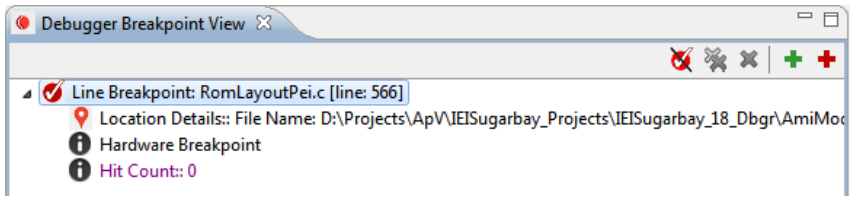


HW Breakpoint: @ SourceLine

When setting Hardware Breakpoint by Address, the ‘By Address’ radio button needs to be selected, The Address, breakpoint type and the size need be entered.



User can click on the source line required in source view and use the 'Set Present Fle Detail' to populate the Filename and Line No fields.

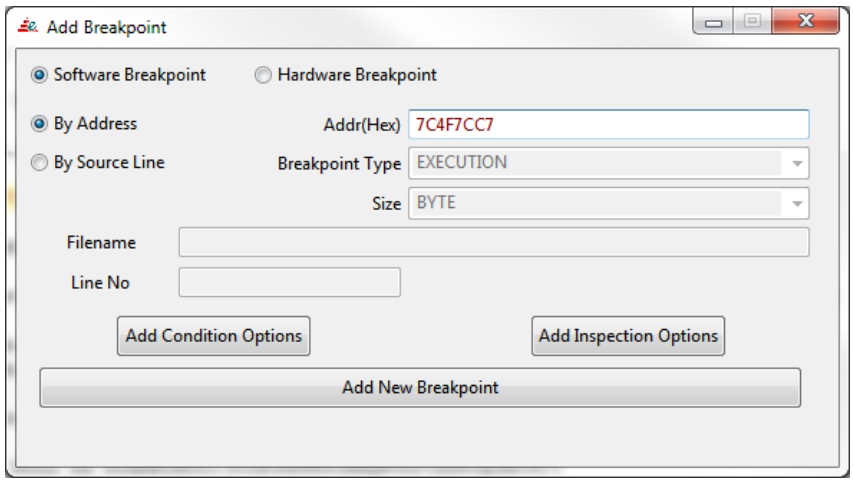


Adding a Custom SW Breakpoint

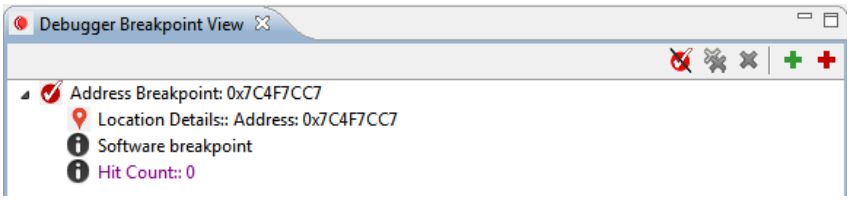
Custom SW break points can be set by selecting the software breakpoint option in the Custom breakpoint view. Software Breakpoint can be set By Address or By SourceLine.

SW Breakpoint: @ Address

A Valid Memory Address is the required input for setting a Software breakpoint at an Address.

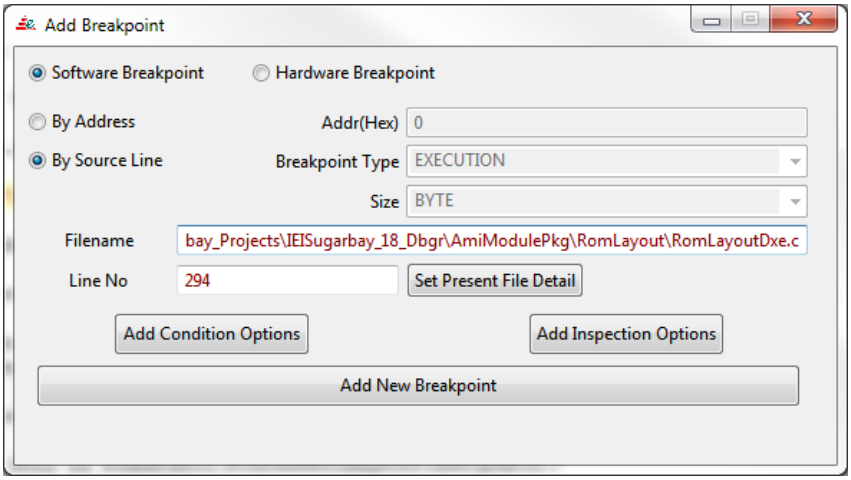


Click on “Add New Breakpoint” button to set the Breakpoint.

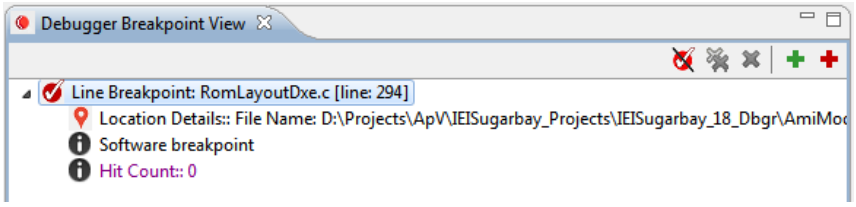


SW Breakpoint: @ SourceLine

The filename and line number may be input as parameters if the breakpoint needs to be created using By Source Line option.



User can click on the source line required in source view and use the ‘Set Present Fle Detail’ to populate the Filename and Line No fields.



NOTE: Adding Software breakpoint @Address or @Sourceline in PEI Phase before Memory is Initialized, will automatically be set as Hardware Breakpoints.

Inspection Points - Setting & Usage

Setting an Inspection point at Boot time can be done using any of the methods mentioned below. First let us understand what is the difference between a breakpoint and an inspection point.

Understanding Breakpoint vs Inspection Point

Breakpoint - is a point at which the Debugger will halt the execution on the Target.

* If the Breakpoint has Conditions associated with it (See Conditional breakpoints), the Breakpoint will halt the execution only if all the Conditions are satisfied.

* If the Breakpoint has Inspections associated with it (See Inspection Points), the Inspection Options will be displayed on the AptioVDebugger Console window and the execution will be halted.

Inspection Point - is a point at which the Inspections options are displayed on the AptioVDebugger Console window, but the execution of the Target will NOT be halted.

Inspection Point @ Boot Time

Using Editor Pane (Source View)

One way to set an Inspection point on any source line, is by right clicking on the Left Pane of Source View and selecting 'Add Inspection Point', as shown below.

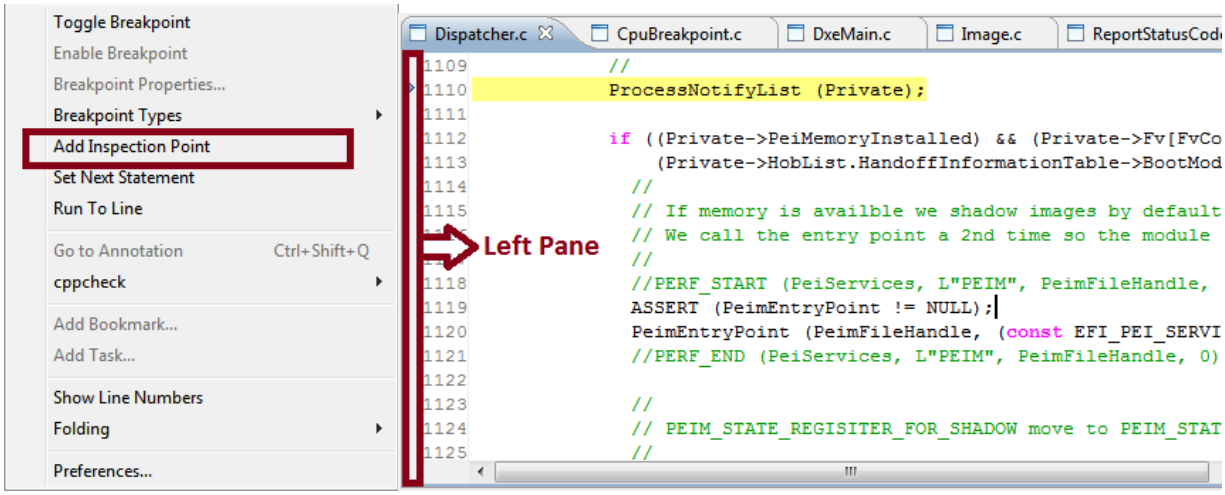
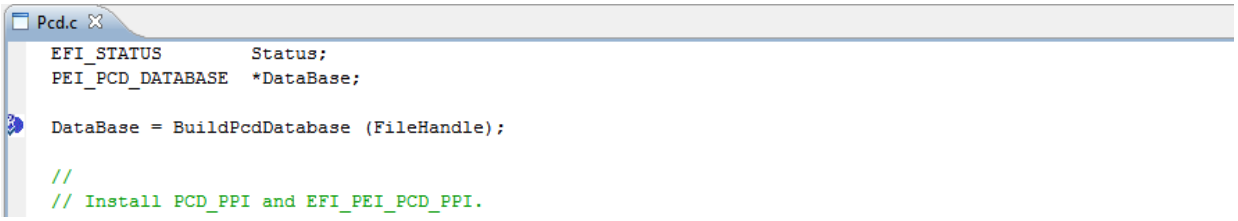


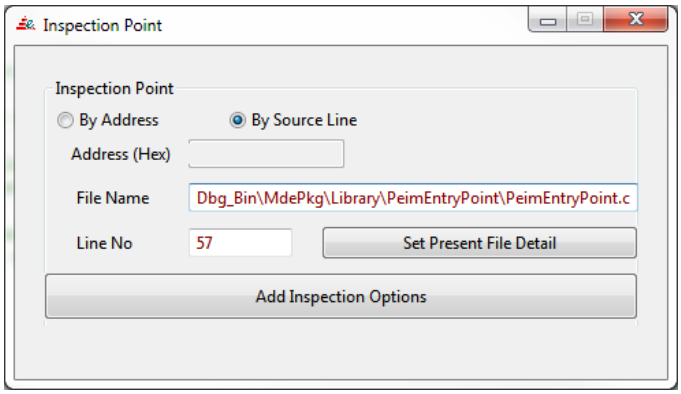
Figure: Source View - Add Inspection Point

On setting the Inspection Point, the inspection option icon will be set by the corresponding source line, like below.

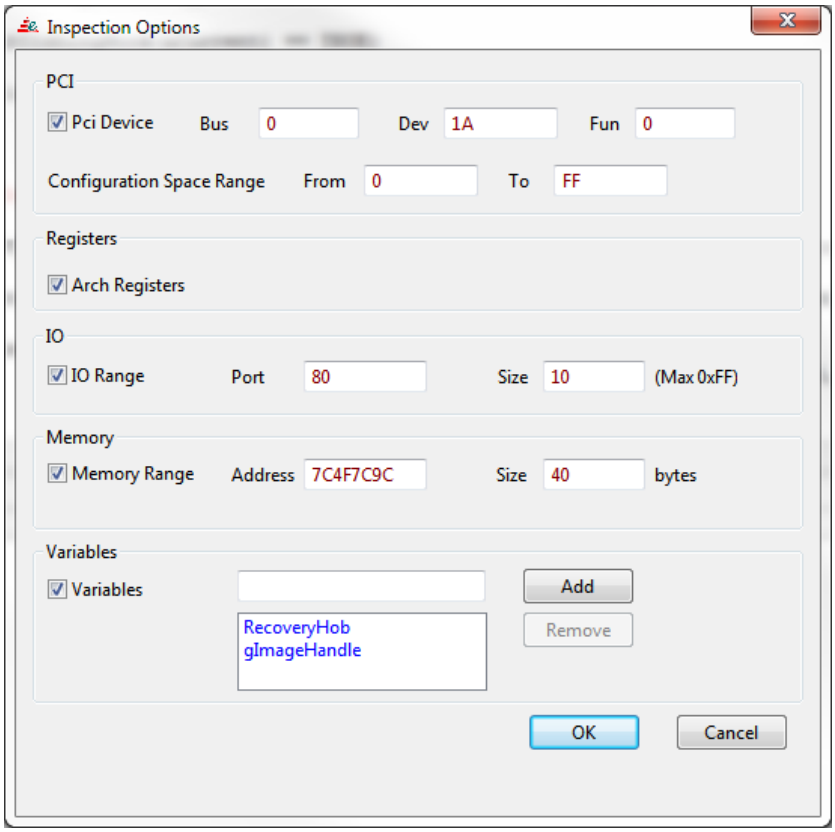


Using HotKey (SHIFT+F9)

Another way to set quick Inspection point is by using the Hot key (SHIFT+F9). This will open the Inspection point window, like below



On selecting By Address or by Sourceline option, the user can select the 'Add Inspection Options' button to open the Inspection Options window used to select the Options.



Similar to breakpoints, using this method, the Inspection points set in PEI phase before Memory initialization will be automatically set as Hardware Inspection points. Software Inspection points will be set only after memory is initialized.

NOTE: When setting *Inspection points* in PEI phase before Memory is initialized, Debugger will automatically set a Hardware *Inspection point* using the Debug registers, it is recommend that users do not set more than 3 combined breakpoints +*Inspectionpoints* since debugger uses 1 Debug register for internal stepping purposes.

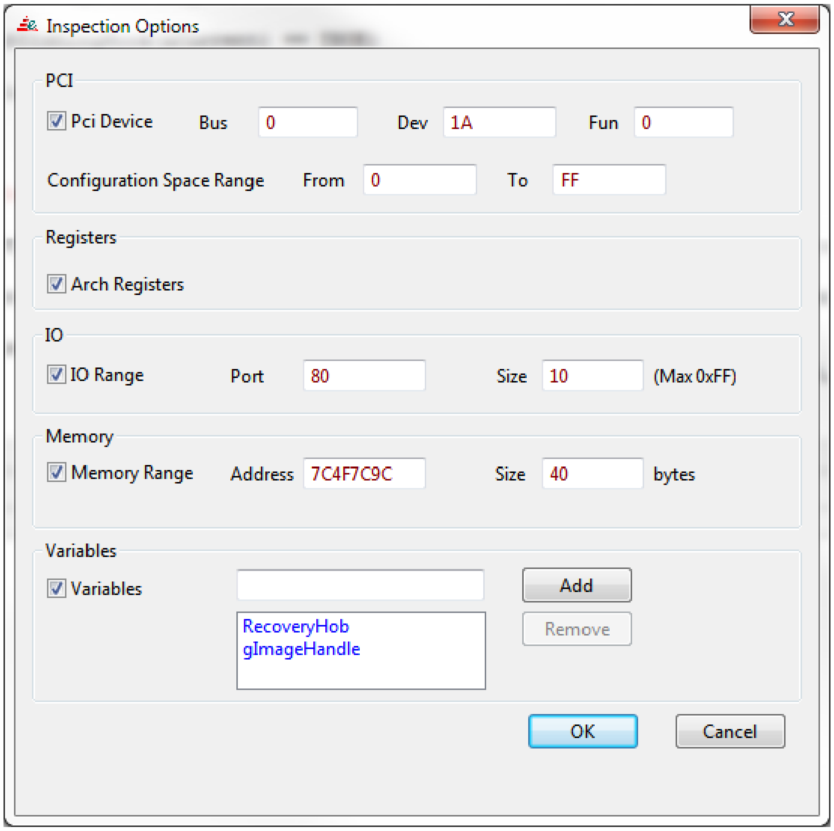
Adding Inspection Options to a Breakpoint

Inspection Options can be set on any set Breakpoint (Hardware or Software). When the breakpoint is 'HIT', the Inspection Options will be displayed on the AptioVDebugger Console and Target's execution will be halted. Users can set Inspection options to Breakpoints while setting the Breakpoint itself by using the Add Breakpoint View (or) can add Inspection Options to existing Breakpoints using the Breakpoints View. Inspection Options can be set to HW and SW breakpoints.

While setting a breakpoint (refer Adding a Custom Breakpoint), before clicking on Add New Breakpoint, the user need to follow the below steps.

In order to set the Inspection Options to a Breakpoint use "Add Inspection Options" button from "Add Breakpoint" window.

Clicking on "Add Inspection Options" button will open the Inspection Options Window. The "Inspection Options" window contains the Inspection Options available to be set such as



PCI

By selecting this option, the Debugger will dump the PCI configuration of the specified BDF and Offset. The output will be displayed on the AptioVDebugger Console Screen.

Sample PCI Output

```
PCI Data for Bus: [0x0] Device: [0x1A] Function: [0x0] Range:(0x0 - 0xff)
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|PCI            | 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F                          |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
|0x00000000     | 86 80 2C 1C 00 00 90 02 05 00 03 0C 00 00 80 00
|0x00000010     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000020     | 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000030     | 00 00 00 00 50 00 00 00 00 00 00 00 00 02 00 00
|0x00000040     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000050     | 13 00 06 03 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000060     | 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000070     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000080     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x00000090     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x000000A0     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x000000B0     | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x000000C0     | 00 20 00 00 00 00 00 00 00 01 00 00 00 00 00 00
```

```
|0x000000D0        | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x000000E0        | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|0x000000F0        | 00 00 00 00 00 00 00 00 87 0F 05 08 00 00 00 00
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|

|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
| Help : Use Command 'setformat byte|word|dword' to change the Grid Format     |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
```

NOTE: Use Command 'setformat byte|word|dword' to change the Grid Format

Registers

By selecting this option, Debugger will dump the entire register set on the AptioVDebugger Console screen.

IO

User can Dump the IO space starting from a specified Port to a specified offset.

Memory

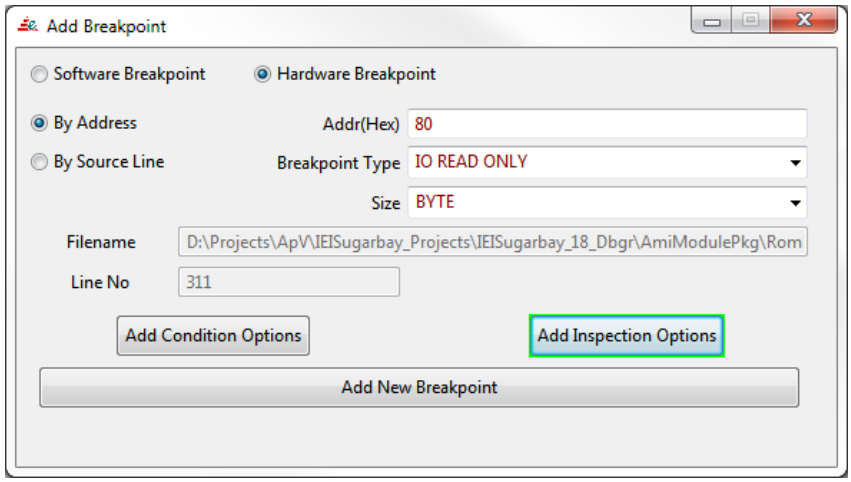
User can Dump the Memory space starting from a specified Address to a specified offset.

Variables

User can enter the name of a Local or Global variable and its value will be dumped to the AptioVDebugger Console Screen.

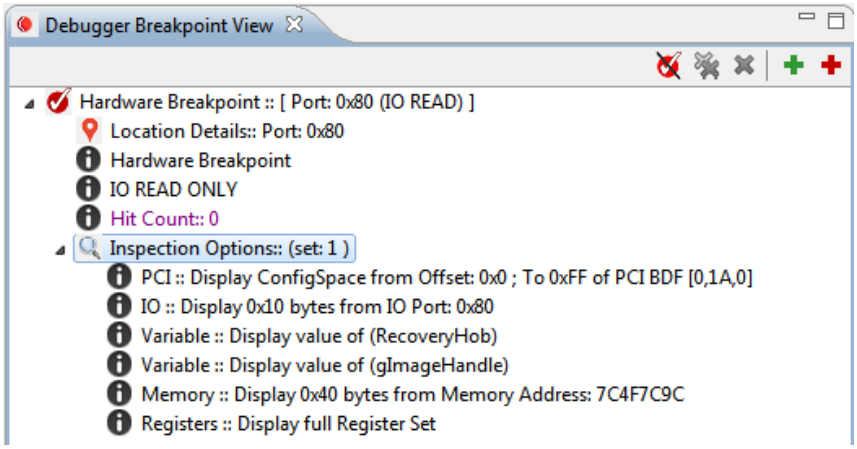
Select the checkbox option, for the required Inspection Options and provide the corresponding data.

Click the Add button in the “Inspection Option” window. After this user can see “Add Inspection Option” button will be highlighted in the Add Breakpoint Window as below



Click on “Add New Breakpoint” button in the “Add Breakpoint” Window.

User can see inspection point with its detail information in breakpoint view window.



For a detailed description on how to set the different Inspection options and ‘how to understand the output’, refer to Inspection Options Window.

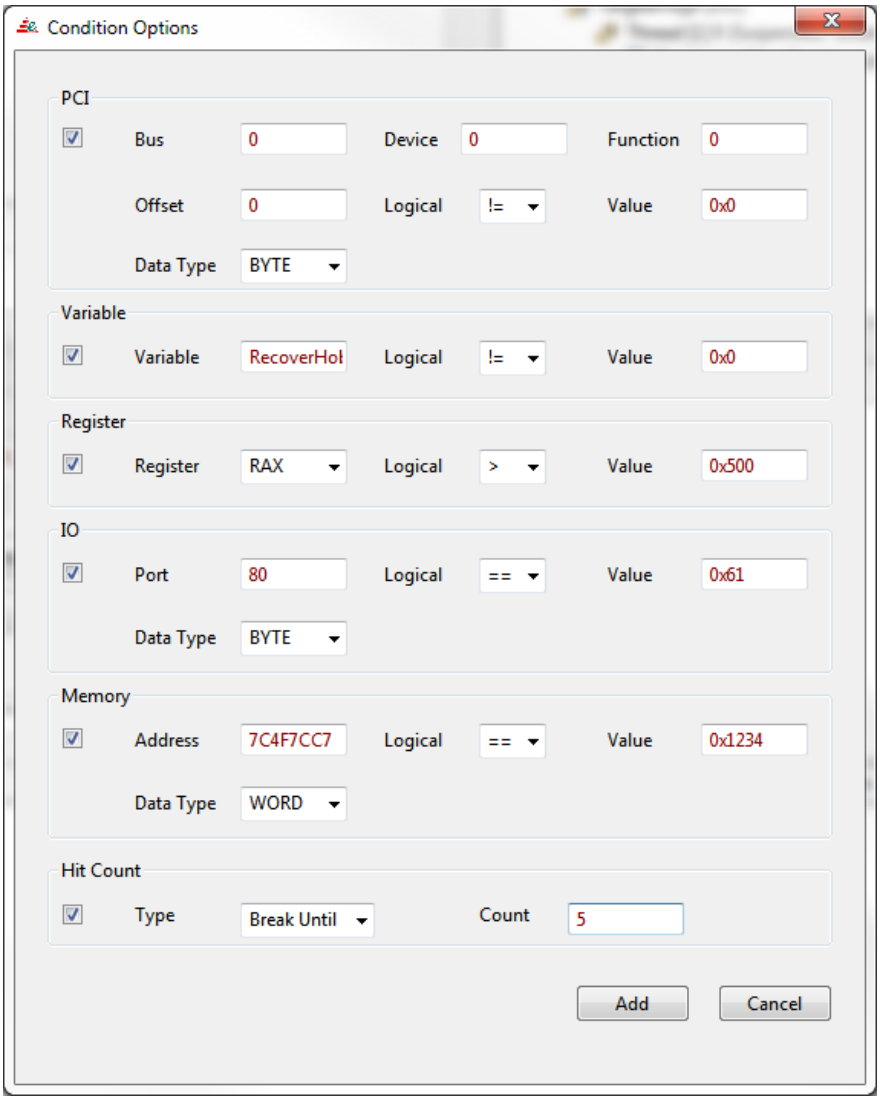
Adding Conditions to a Breakpoint

Conditions can be set on any Breakpoint. When the breakpoint is ‘HIT’, the target will either Halt execution or continue execution based on the conditions. Users can set Conditions to Breakpoints while setting the Breakpoint itself by using the Add Breakpoint View (or) can add Conditions to existing Breakpoints using the Breakpoints View. Conditions can be set to HW and SW breakpoints.

While setting a breakpoint (refer Adding a Custom Breakpoint), before clicking on Add New Breakpoint, the user need to follow the below steps.

In order to set Conditions to a Breakpoint, use “Add Condition Options” button from “Add Breakpoint” window.

Clicking on “Add Condition Options” button will open the Condition Options Window.



User can select different condition (one or more) to halt at specified breakpoint if those conditions satisfy. After selecting the condition option, user has to provide appropriate input as below explained.

PCI

By selecting this option, user can set the PCI configuration condition for breakpoint.

In order to set PCI configuration condition user has to provide values for the mentioned fields, such as bus, device, function, offset, logical condition, value and Data Type.

Variable

By selecting this option, user can set the particular variable, choose the logical operation to be performed and the value with which it has to be compared.

Register

By selecting this option, user can set the particular register, choose the logical operation to be performed and the value with which it has to be compared.

IO

By selecting this option, user can set the particular IO, choose the logical operation to be performed and the value with which it has to be compared. Also need to choose data type.

Memory

By selecting this option, user can set the particular memory address, choose the logical operation to be performed and the value with which it has to be compared. Also need to choose data type.

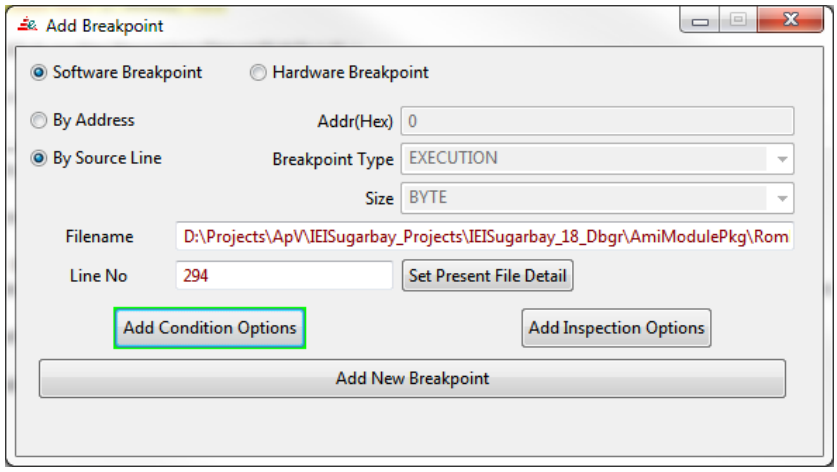
Hit Count

User can select any one of the below option as hit count type.

Break Until (n) – Breakpoint will halt on hit until the count value is <= n, after which the breakpoint hit will not halt at this breakpoint.

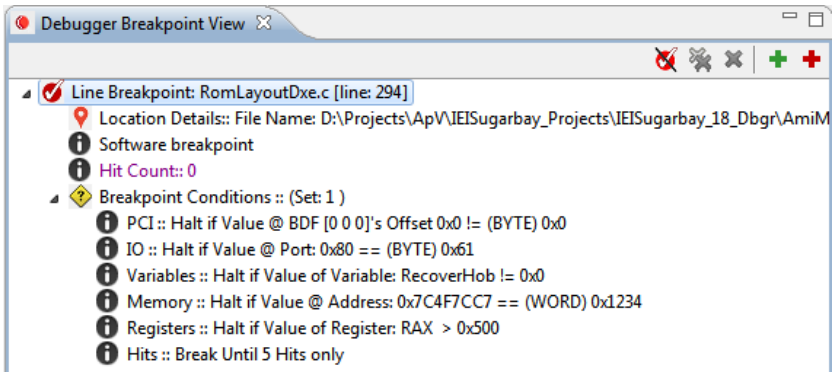
Break After (n) – Breakpoint will not halt on breakpoint hit until count value is > n, after which it will halt at this breakpoint.

Click on Add button in the “Condition Option” window. After this user can see “Add Condition Option” button in the Add Breakpoint window as like below.



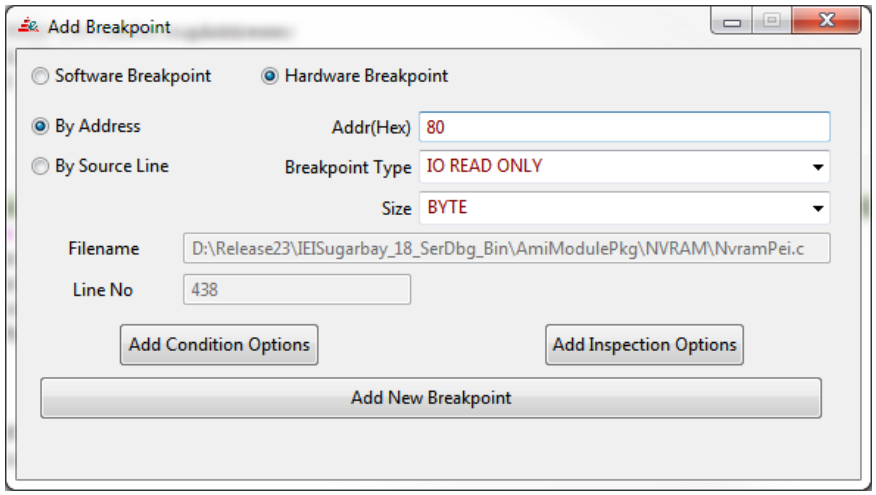
Click on “Add New Breakpoint” button in the “Add Breakpoint” window.

User can see breakpoint with its detailed information in breakpoint view window.



Example:

Below screenshot, we are setting a new breakpoint. Now if we want make it as a conditional breakpoint, select ‘Add Condition Options’.



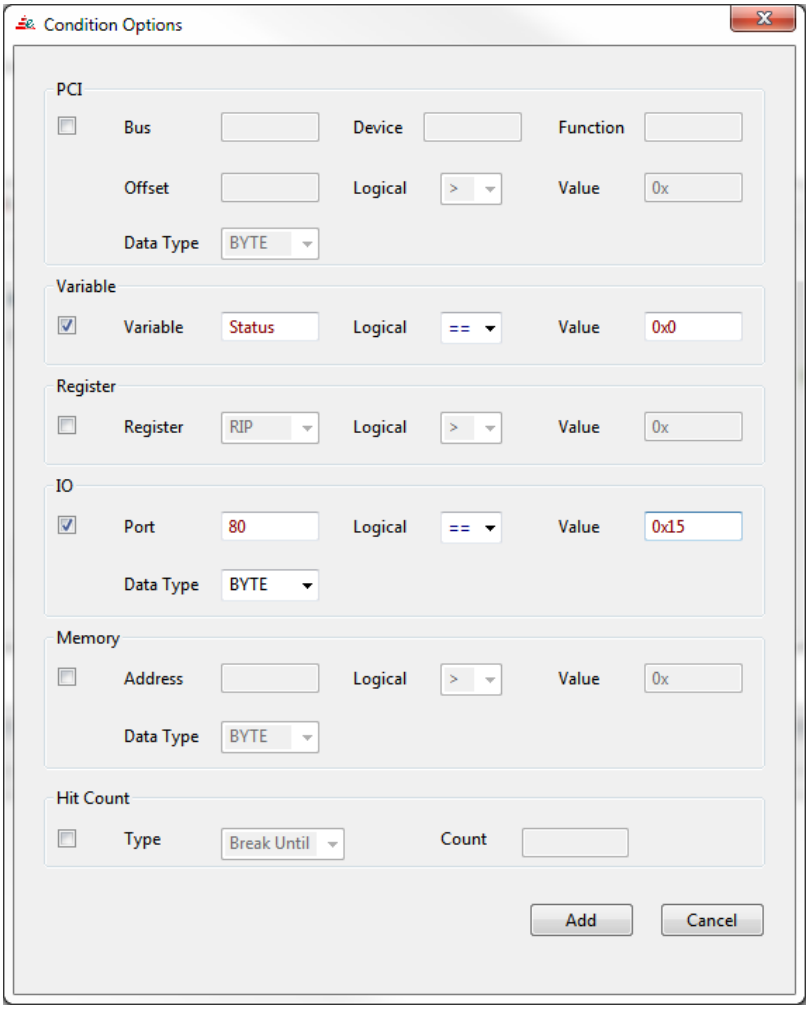
In below screenshot we were setting the condition by clicking on add condition. This will open the conditions window, now we set a sample condition,

Say Condition1 → Local variable Status == 0

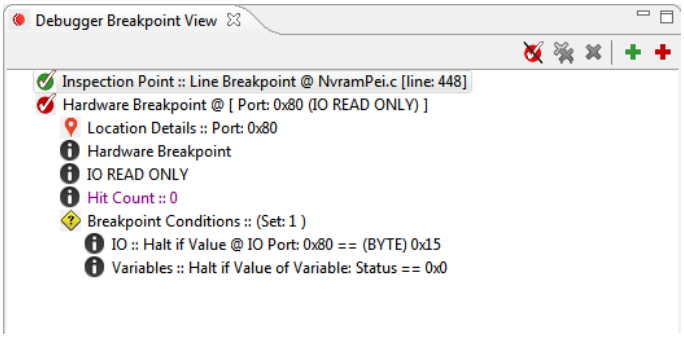
And

Condition 2 → IO Port 80 value is equal to 0x15.

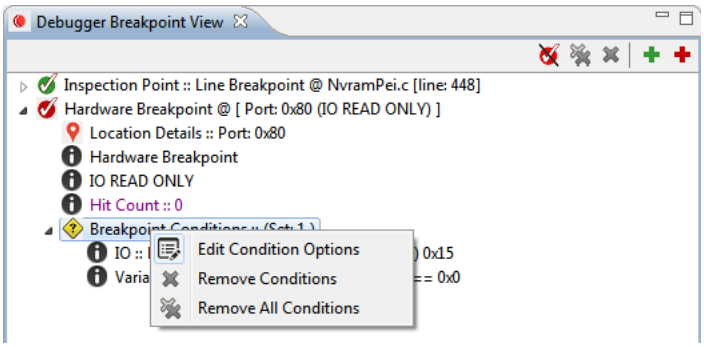
This means above breakpoint will happen only if both the conditions will be satisfied, Otherwise it will not halt.



In addition, we can see and add condition or inspection point on existing breakpoint using breakpoint view. Simply by right click on breakpoint then click on Add Condition/Inspection option to add Condition/Inspection point. Below screenshot is showing how to add condition or inspection point to existing breakpoint.

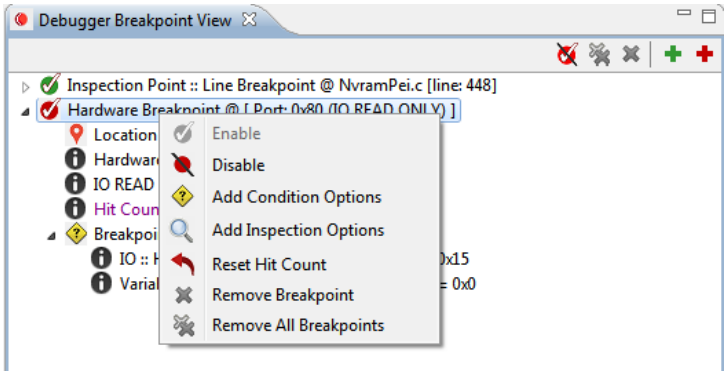


User can modify or edit the set Conditions from the Breakpoint View by Right clicking on the Conditions Set, like below



And selecting 'Edit Condition Options', this will again pop up the Condition Options window with the set conditions. User can edit the previously set conditions or add/remove conditions.

Users can also add a new set of conditions (allows multiple conditions of the same type) by right clicking on the breakpoint itself and selecting 'Add Condition Options', like below

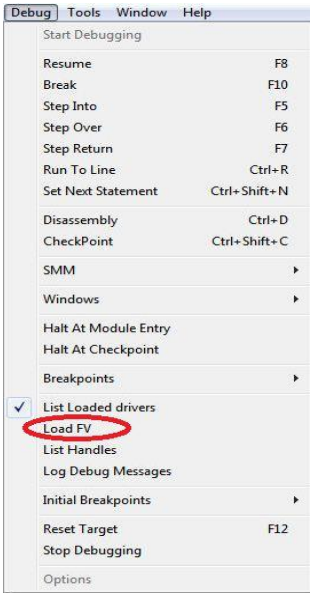


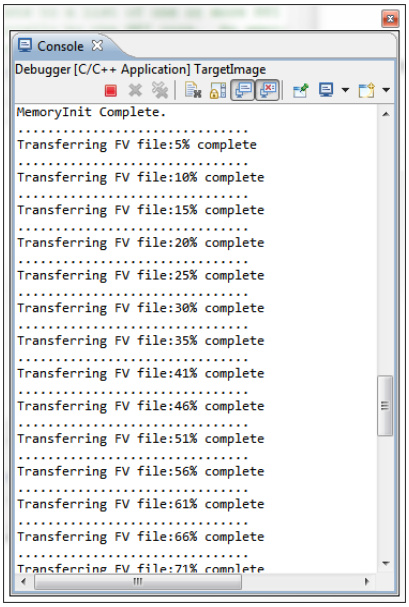
Load Firmware Volume

Select "Debug→Load Firmware Volume" Or by selection debugger tool bar icon [ ]

Load FV support enables users to load a new Firmware Volume (FV_MAIN) without needing to re-flash the Target.

- Open Host application and Start a Debug session
- Power On the target
- On PEI Initial breakpoint (or any breakpoint before Memory is initialized)
- Select “Debug→Load Firmware Volume” and in pop-up explorer select the ROM or BIN image
- After Memory is initialized Debugger will load the new FV_MAIN from the ROM or BIN image selected





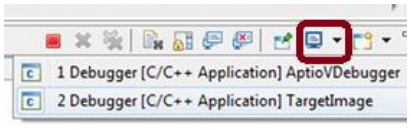
Console Window and Logs

AptioVDebugger uses two consoles windows to display messages

(a) AptioVDebugger: Displays the status messages from the AptioVDebugger host

(b) TargetImage: Displays the Debug Trace message received from the Target BIOS

Users can toggle between both Console views



(or) Stick to one view by pinning the console view.

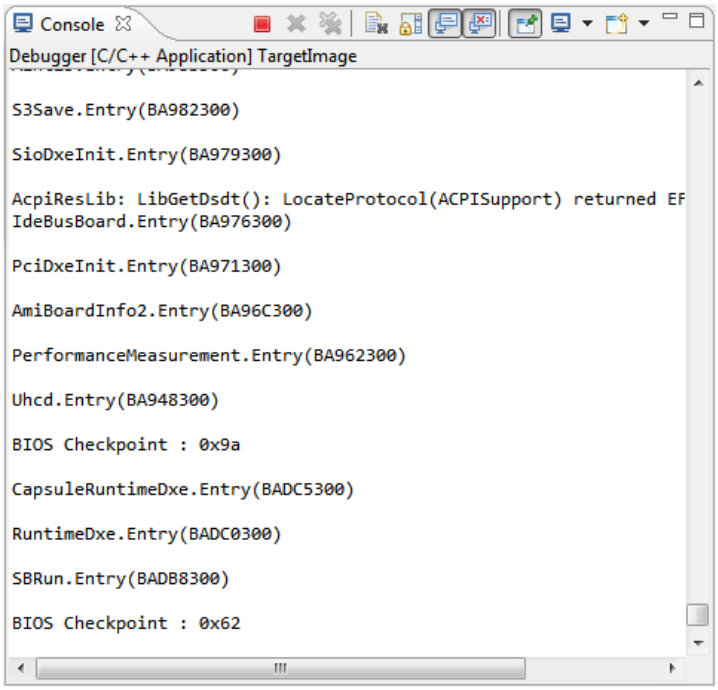


Figure: Console Window displaying the Target Debug Messages (Pinned).

Breakpoint Info

AptioVDebugger displays additional information in the AptioVDebugger Console widow to provide user with the following

- Current Register context
- Current Instruction Line info
- Last Operation Duration

How this data can be interpreted

The Breakpoint Info example is displayed here -

RAX-0x0000000000020020 RBX-0x000000007A74D2E0 RCX-0x00000000FEE00320 RDX-0x0000000000020020

RSP-0x000000007A74CE00 RBP-0x0000000000101D6C8 RSI-0x00000000010215C8 RDI-0x0000000000000000

R8 -0x0000000000000001 R9 -0x000000007A780420 R10-0x0000000000000000 R11-0x00000000FEE00320

R12-0x0000000000000000 R13-0x0000000000000000 R14-0x0000000000000000 R15-0x0000000000000000

RIP-0x000000007A74D70A CS-0x0038 SS-0x0008 DS-0x0008 ES-0x0008 FS-0x0008 GS-0x0008

EFL-0x00000306 -->[IOPL=0 NV UP EI PL NZ NA PE NC]

```
DxeCore!DxeMain() 00000000'7A74D70A:        LEA   R8, [RSP+0xf0]
Last Operation: STEP OVER-->            Duration - 27.421873 mSec
```

The Register Context as shown displays the General Purpose registers and their values,

The Flags Register (EFL above) is interpreted as such -

```
The Flags register is an 8-bit register that holds 1-bit of data for each of
the eight "Flags" which are to be interpreted as follows:

Textbook abbrev. for Flag Name =>   of df if sf zf af pf cf
If the FLAGS were all SET (1),      -- -- -- -- -- -- -- --
they would look like this...   =>   OV DN EI NG ZR AC PE CY
If the FLAGS were all CLEARed (0),
they would look like this...   =>   NV UP DI PL NZ NA PO NC

                      FLAGS            SET (a 1-bit)   CLEARed (a 0-bit)
                ---------------      ---------------- ------------------
                 Overflow   of  =    OV (OVerflow)    NV [No oVerflow]

                Direction   df  =    DN (decrement)   UP (increment)

                Interrupt   if  =    EI  (enabled)    DI (disabled)

                     Sign   sf  =    NG (negative)    PL (positive)

                     Zero   zf  =    ZR   [zero]      NZ  [ Not zero]

          Auxiliary Carry   af  =    AC  [Aux Carry]  NA  [ No AC ]

                   Parity   pf  =    PE  (even)       PO  (odd)

                    Carry   cf  =    CY  [Carry]      NC  [ No Carry]
```

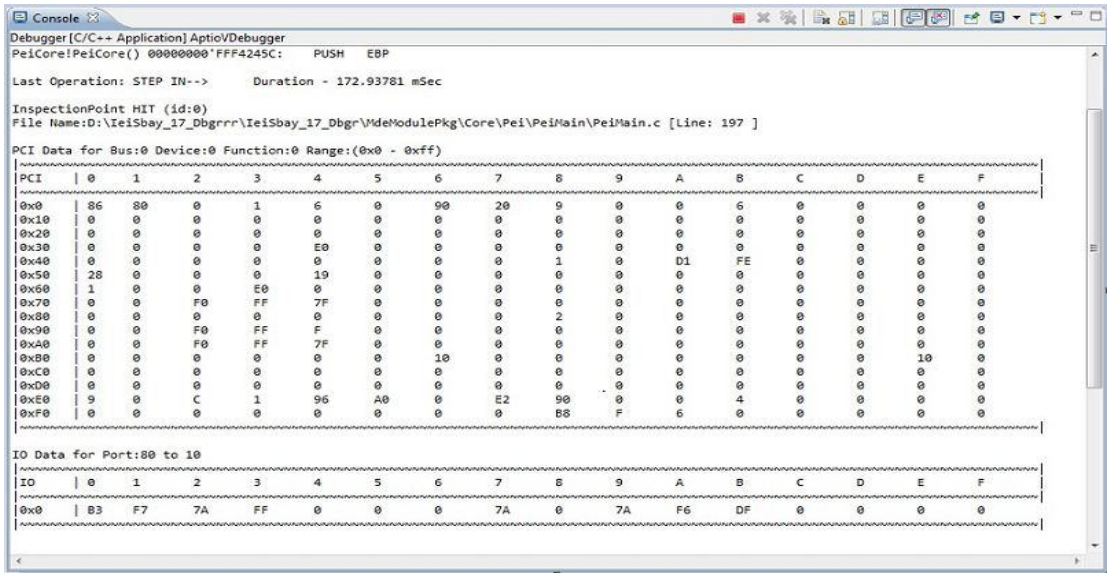
The Instruction Line info is ordered in this format

[Module]![Function()] [IP value]: [Instruction Disassembly]

The Last Operation Dutartion line display the last operation as a Sting (eg;STEP OVER) and the duration is, the time taken from operation given to breakpoint complete.

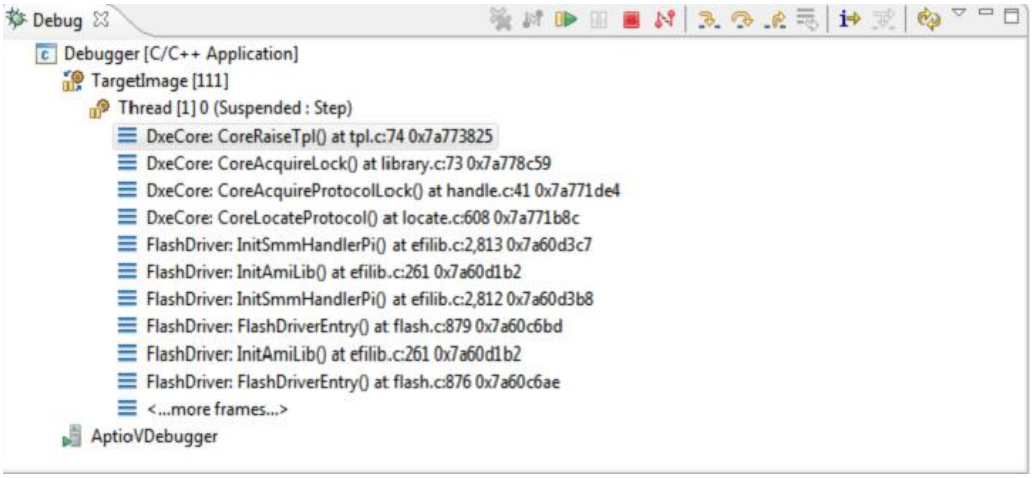
Dumped info

AptioV debugger console window display the user requested dump info at inspection point or breakpoint. It may contain user selected PCI configuration data, Variable data, Memory data, IO data and Register context data.



Call Stack

The Debug View contains the Call stack Displayed under TargetImage Thread. This view cannot be turned off, Debug View window will always display it



The format on the Frame information is -

[Module Name]: [Function Name ()] at [Filename]: [Line No] [Instruction Address]

Clicking on any of the Frames will display the selected File, pointing to the line Number.

Giving a user a clear path of execution if required.

Logging Debug Messages to file

Select "Debug →Debug Message Log" Or using the Debug Toolbar Icon []

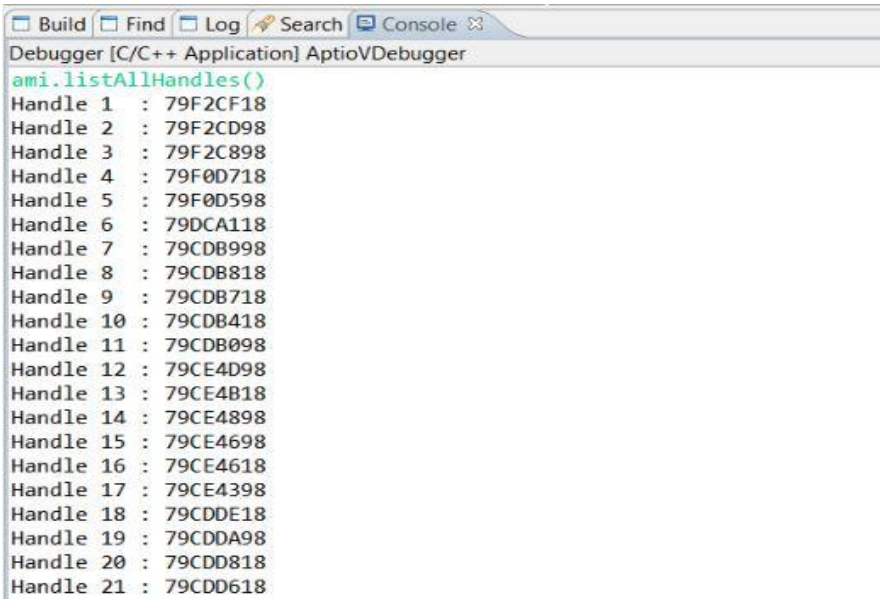
Selecting this option will create a "DebugMessageLog.txt" file in VisualeBios folder and all Debug messages will be logged in it.

*NOTE: By Default user will get DeubgMessageLog.txt without timestamp. However, user can toggle between log with timestamp and log without timestamp using the commands "**timestampon**" and "**timestampoff**" respectively in AptioVDebugger console.*

Scripting Support

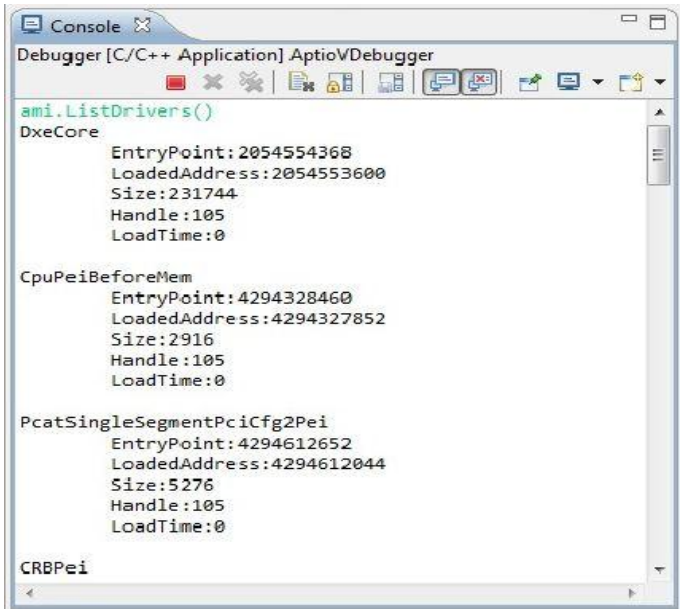
AptioVDebugger console can support the execution of Python Script files, or existing defined Commands and view the output in same console window.

There are already some predefined Scripts and Commands provided with AptioVDebugger, which are defined in MasterScript.py file present in scripts folder of VeB path. User can run these commands or custom written python scripts (including ITP scripts) directly in AptioV debugger console.

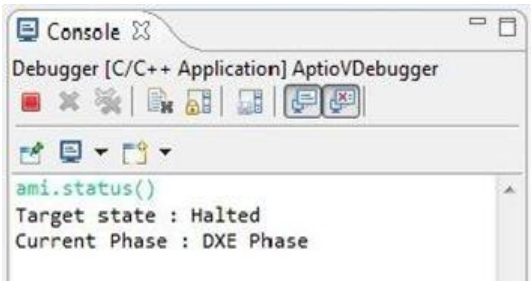


Below are some more example that how to run and see a few commands and outputs in the AptioVDebugger console.

If you want to see the list of loaded driver with their info such as Entry Point, Loaded address, size, handles and load time then you can run command ami.ListDrivers() which is already defined in master script.



If you want to see status of target the run the ami.Status() script command.



User can define their own function by using master script commands. Set of command defined in script can use along with master script. In order to run user defined script, user has to load script file first.

To load the user-defined script and run the user defined function, user has to do following step.

1. First copy python Script file into Script folder, which is present in VeB directory.

2. Use ami. Load ('python file name') then press Enter Key.

3. Now, user can run any function present in user defined script.

In above screenshot hsw_msr.py is a user defined script file, which contain user defined function to read MSR. This user defined function internally uses the master script command i.e. ami.msr(). After loading the user defined script can be run as a function call. Output of this command is displayed in the same window.

6.9.1 AMI Debugger Scripting Commands

Below are the list of AMI scripting commands which can be accessed by importing the AMI class object in the Python Script file.

Load Script File

1. **Load:**

Usage: load (file path)
Argument: file path expressed in string– relative script file path which want to load.
Return: None.
This command is used to load the script file.
Example: ami.load('scripts/pci/cscripts_startup.py')

Execution of Script File

1. **Execute:**

Usage: execute (file path)
Argument: file path expressed in string – relative script file path.
Return: None.
This command is used to execute the specified script file.
Example: ami. execute ('cscript_startup.py')

Loaded Driver

1. **ListDrivers:**

Usage: ListDrivers ()
Argument: None
Return: None. It display the list of loaded driver list.
This command will display the list the name of loaded driver.
Example: ami. ListDrivers ()

2. **LoadedDriverDetails:**

Usage: LoadedDriverDetails (Driver name)
Argument: loaded Driver Name expressed as string
Return: None. It display the specified loaded driver detail.

This command will take an argument as loaded driver name and return detail information of that loaded driver such as Entry Point, Start address, End Address, Size and Handle.
Example: ami. LoadedDriverDetails ('DxeCore')

3. LISTSrcFiles:

Usage: LISTSrcFiles (*arg)
Argument: loaded Driver Name expressed as string
Return: None. It display the associated file list of a loaded driver.
This command will take an argument as loaded driver name and return the associated file list of that loaded driver.
Example: ami. LISTSrcFiles ('DxeCore')

Variable

1. DisplayLocalVar:

Usage: DisplayLocalVar (varName) or DisplayLocalVar ()
Argument: local variable Name or None expressed as string
Return: None. It display the specified variable detail or all local variable detail.
This command will give the specified variable details. If no variable is specified then it will return all local variable details.
Example: ami. DisplayLocalVar ('Status') or ami. DisplayLocalVar ()

2. VarGetValue:

Usage: VarGetValue (varName)
Argument: varName - local/global variable name expressed in string
Return: None. It display the specified local/global variable detail.
This command will take an argument as local/global variable name and display detail information of specified local/global variable.
Example: ami. DisplayLocalVar ('Status')

3. VarSetValue:

Usage: VarSetValue (Var_name, Value)
Argument: local/global variable name expressed as string, variable value expressed as int / Hex /Hex string value

Return: None.

This command will take an argument as local/global variable name and new value which want to set. This function will set the specified variable with specified value.
Example: ami. VarSetValue ('Status', '0x1')
ami. VarSetValue ('Status', 15)

Instruction Mode

1. **usemode:**

Usage: usemode ()
Argument: None
Return: None. It display the Instruction mode.
This command will display the instruction mode.
Example: ami. usemode ()

2. **usemoderaw:**

Usage: usemoderaw ()
Argument: None
Return: None. It display the Instruction mode.
This command will display the instruction mode.
Example: ami. usemode ()

Stepping Operations

1. **RunToAddress:**

Usage: go (address)
Argument: address as Hex String value/ long /Hex value
Return: None.
This command used to resume the execution of target till specified address.
Example: ami. go ('0xffe350d0')
ami.go(0xffe350d0)

ami.go(4293087440)

2. Resume:

Usage: go ()
Argument: None
Return: None.
This command used to resume the execution of target.
Example: ami. go ()

3. Break:

Usage: halt ()
Argument: None
Return: None.
This command is used to halt the target from the current execution (Dynamic Break).
Example: ami. halt ()

4. stepInto:

Usage: stepInto ()
Argument: None
Return: None.
This command will step into the source code while debugging.
Example: ami. stepInto ()

5. stepOver:

Usage: stepOver ()
Argument: None
Return: None.
This command will step over the source code while debugging.
Example: ami. stepOver ()

6. stepReturn:

Usage: stepReturn ()
Argument: None
Return: None.
This command will step return the source code while debugging.
Example: ami. stepReturn ()

Halt Driver

1. **setHaltDriver:**

Usage: setHaltDriver (driver name)
Argument: Driver Name expressed in string
Return: None.
This command is used to set halt at specified driver entry point.
Example: ami. setHaltDriver ('peiCore')

2. **RevmoveHaltDriverByName:**

Usage: removeHaltDriver (driver name)
Argument: Driver Name expressed in string
Return: None.
This command is used to remove halt at specified driver entry point.
Example: ami. removeHaltDriver ('peiCore')

3. **HaltDriverNext:**

Usage: setHaltDriver ('NEXT')
Argument: 'NEXT' string.
Return: None.
This command is used to set halt at next driver entry point. Note: argument should be "NEXT" String.
Example: ami. setHaltDriver ('NEXT')

4. **HaltDriverAll:**

Usage: setHaltDriver ('ALL')
Argument: "ALL" String.

Return: None.
This command is used to set halt at All driver entry point. Note: argument should be “ALL” String.
Example: ami. setHaltDriver (‘ALL’)

5. **RemoveHaltDriverAll:**

Usage: removeHaltDriver (‘ALL’)
Argument: “ALL” string.
Return: None.
This command is used to remove halt at all driver entry point.
Note: argument should be “ALL” String.
Example: ami. removeHaltDriver (‘ALL’)

6. **RemoveHaltDriverNext:**

Usage: removeHaltDriver (‘NEXT’)
Argument: “NEXT” String.
Return: None.
This command is used to remove halt at next driver entry point. Note: argument should be “NEXT” String.
Example: ami. removeHaltDriver (‘NEXT’)

Register

1. **Read Specific Register:**

Usage: reg (register name)
Argument: name of the register which want to read
Return: value of specified register.
This command is used to display the value of specified register.
Example: ami. reg (‘ax’)

2. **Read All Register set :**

Usage: regs ()
Argument: None
Return: value of all register set.
This command is used to display the value of all register set. Register name can be EAX, EBX, ECX, EDX etc.
Example: ami. reg ()

3. Read Extended Register:

Usage: xmregs ()
Argument: None
Return: None. it display value of extended register set.
This command is used to display the value of extended register set.
Example: ami. xmregs ()

4. Write Register:

Usage: reg (reg name, value)
Argument: Register name; value which want to write.
Return: None.
This command is used to write the specified value into the specified register.
Register name can be AX, BX, CX, DX etc. including extended Registers set.
Example: ami. reg ('ip',123)

CPUID

1. cpuid_eax:

Usage: cpuid_eax (eax, ecx)
Argument: eax value expressed as int/long/string, ecx value (optional) expressed as int/long/string
Return: Return the processor identification and feature information.
This command used to get the processor identification and feature information stored in the EAX register.
Example: ami. cpuid_eax ('2') or

ami. cpuid_eax (‘2’, ‘1’)

2. cpuid_ebx:

Usage: cpuid_ebx (eax, ecx)
Argument: eax value expressed as int/long/string, ecx (optional) value expressed as int/long/string.
Return: Return the processor identification and feature information.
This command used to get the processor identification and feature information stored in the EAX register.
Example: ami. cpuid_ebx (2) or
ami. cpuid_ebx (2,1)

3. cpuid_ecx:

Usage: cpuid_ecx (eax, ecx)
Argument: eax value expressed as int/long/string, ecx (optional) value expressed as int/long/string
Return: Return the processor identification and feature information stored.
This command used to get the processor identification and feature information stored in the EAX register.
Example: ami. cpuid_ecx (2) or
ami. cpuid_ecx (2,1)

4. cpuid_edx:

Usage: cpuid_edx (eax, ecx)
Argument: eax value expressed as int/long/string, ecx (optional) value expressed as int/long/string
Return: Return the processor identification and feature information stored.
This command used to get the processor identification and feature information stored in the EAX register.
Example: ami. cpuid_edx (2) or
ami. cpuid_edx (2,1)

Halt at Checkpoint

1. **setHaltCheckpoint:**

Usage: setHaltCheckpoint (Checkpoint)
Argument: Checkpoint (it may be ALL, NEXT in string format or specific checkpoint value in int/Hex/Hex String format)
Return: None.
This command is used to set the halt at Checkpoint. User can set the halt at specific checkpoint or next or all checkpoint.
For setting halt at next checkpoint, argument should be “NEXT”.
For setting halt at all checkpoint, argument should be “ALL”.
Example: ami. setHaltCheckpoint (‘0xB3’) or
ami. setHaltCheckpoint (‘NEXT’) or
ami. setHaltCheckpoint (‘ALL’) or
ami. setHaltCheckpoint (0xB3)

2. **Remove Halt at Checkpoint:**

Usage: removeHaltCheckpoint (checkpoint)
Argument: Checkpoint (it may be ALL, NEXT or specific checkpoint value)
Return: None.
This command is used to remove the halt at Checkpoint. User can remove the halt at specific checkpoint or next or all checkpoint.
For removing halt at next checkpoint, argument should be “NEXT”.
For removing halt at all checkpoint, argument should be “ALL”.
Example: ami. removeHaltCheckpoint (‘0xB3’) or
ami. removeHaltCheckpoint (‘NEXT’) or
ami. removeHaltCheckpoint (‘ALL’)

Breakpoints

1. **Create S/w breakpoints:**

Usage: brnew (address or file name and line number)
Argument: File name(string) with line number(int) or only address(Hex/int/long/Hex String)
Return: None.
This command is used to set software breakpoint at specified address or line breakpoint.
While setting line breakpoint, file name should be with absolute path.
Example:

ami. brnew (0xB31234) or

ami. brnew ('D: \sugarbay\DbgSerInitLib\DbgSerInitLib.c ', 118) or

ami. brnew ('0xB31234')

2. Create H/w breakpoints:

Usage: brnew (address or file name with line number, breakpoint condition)
Argument: File name(string), line number(int) and breakpoint condition(string) or address(hex/long/int) and breakpoint condition(string)
Return: None.
This command is used to set H/w breakpoint at specified address or source line with specified condition.
Condition can be exec, ioread, iowrite, datawrite, noexec.
Example: ami. brnew ('0xB31234', 'exec') or

ami. brnew ('0xB31234', 'exec',)

3. Remove breakpoints:

Usage: brremove (breakpoint id)
Argument: breakpoint id (int)
Return: None.
This command is used to remove the breakpoint of specified id.
Example: ami. brremove (2)

4. List breakpoints:

Usage: br ()
Argument: None.
Return: None.
This command is used to display the list of breakpoints.
Example: ami. br ()

5. Enable breakpoints:

Usage: brenable (breakpoint id)
Argument: breakpoint id
Return: None.
This command is used to enable the breakpoint of specified id.
Example: ami. brenable (2)

6. Disable breakpoints:

Usage: brdisable (breakpoint id)
Argument: breakpoint id
Return: None.
This command is used to disable the breakpoint of specified id.
Example: ami. brdisable (2)

7. Breakpoints info:

Usage: brget (breakpoint id)
Argument: breakpoint id
Return: None.
This command is used to display detail information of specified breakpoint.
Example: ami. brget (2)

8. Add Inspection point:

Usage: AddInspOption (breakpoint id, inspection point type)
Argument: breakpoint id(int) and inspection point type(string)
Return: None.
This command is used to set the inspection point of specified type at specified breakpoint.
Inspection type can be pci, Io, reg, vars etc
Example: ami. AddInspOption (2,'pci')

9. Remove inspection point:

Usage: RemoveInspOption (breakpoint id)

Argument: breakpoint id
Return: None.
This command is used to remove the inspection point at breakpoint of specified id.
Example: ami. RemoveInspOption (2)

PCI

1. PCI Count:

Usage: PCICount ()
Argument: None
Return: Total pci count.
This command is used to get total pci count.
Example: ami. PCICount ()

2. PCI List:

Usage: PCIList (Lower bus range, higher bus range)
Argument: Lower bus range(int/hex/hex string), higher bus range(int/hex/hex string)
Return: List contain bus, device, fun, device id, vendor id.
This command is used get the list PCI devices present in specified range.
Example: ami. PCIList (1,250)

3. Read PCI config space:

Usage: pci_config (bus, dev, fun, reg, size)
Argument: bus number, device number, function number, register number, size of data want to read (Int/long/hex)
Return: None.
This command is used to read the specified size of data from pci configuration space of specified bus, device, function, register.
Example: ami. pci_config (1, 1, 1, 0, 4)

4. Write PCI config space:

Usage: pci_config (bus, dev, fun, reg, size, value)

Argument: bus number, device number, function number, register number, size of data want to write, value which want to write(int/hex/long)
Return: None.
This command is used to write the specified value to the specified size of data into pci configuration space of specified bus, device, function, register.
Example: ami. pci_config (1, 1, 1, 0, 4, 1234)

IO

1. **Read byte from IO port:**

Usage: port (port number)
Argument: port number expressed as Int/hex value/Hex String.
Return: None.
This command is used to read the 1 byte data from the specified port number.
Example: ami. port (0x80)

2. **Read word from IO port:**

Usage: wport (port number)
Argument: port number expressed as Int/hex value/ Hex String.
Return: None.
This command is used to read the 2 byte data from the specified port number.
Example: ami. wport (0x80)

3. **Read Double word from IO port:**

Usage: dport (port number)
Argument: port number expressed as Int/hex value/ Hex String.
Return: None.
This command is used to read the 4 byte data from the specified port number.
Example: ami. dport (0x80)

4. **Write byte to IO port:**

Usage: port (port number, value)
Argument: port number expressed as Int/hex/hex string, value expressed as Int/hex value /Hex String
Return: None.
This command is used to write the specified value to byte of the specified Port number.
Example:
ami. dport (0x80,0x12) - for writing

5. **Write word to IO port:**

Usage: wport (port number, value)
Argument: port number expressed as Int/hex/ Hex String, value expressed as Int/hex/ Hex String
Return: None.
This command is used to write the specified value to 2 byte of the specified Port number.
Example:
ami. dport (0x80,0x1234)

6. **Write double word to IO port:**

Usage: dport (port number, value)
Argument: port number expressed as Int/Hex/ Hex String, value expressed as Int/hex/ Hex String
Return: None.
This command is used to write the specified value to 4 byte of the specified Port number.
Example: ami. dport (0x80,0x123423)

Indexed IO

1. **Read indexed IO port:**

Usage: indexedIO (index, data, start, count)
Argument: index-port address, data port address, offset, count-number bytes to read. (Arguments can be int/hex/long/Hex String)
Return: number of byte read.
This command is used to read the specified byte data from the specified index port address, data port and offset.
Example: ami. indexedIO (0x70, 0x71, 4, 2)

2. Write indexed IO port:

Usage: indexedIO (index, data, start, count, value)
Argument: index-port address, data port address, offset, count-number bytes to read, value which want to write. (Arguments can be int/hex/long/Hex String)
Note: if the value is of two bytes, provide it byte by byte separating it by comma
Return: True or false.
This command is used to write the specified value to specified byte data of the specified index port address, data port and offset
Example: ami. indexedIO (0x70, 0x71, 4, 2,0x12,0x34)

MSR

1. Validate MSR:

Usage: validateMSR (msr value)
Argument: msr value which want to validate (Hex/int/Hex String).
Return: None.
This command is used to validate the specified msr. Also it will display the msr valid or not.
Example: ami. validateMSR (0x17)

2. Read MSR:

Usage: msr (msr value)
Argument: msr value which want to read (Hex/int/Hex String).
Return: msr value.
This command is used to read the specified msr. Also it will return the msr value.
Example: ami. msr (0x17)

3. Write MSR:

Usage: msr (msr value, new value)
Argument: msr value, new value which want to write. (Hex/int/Hex String).
Return: True or false.
This command is used to write the specified value to specified msr.
Example: ami. msr (0x17,12)

Memory

1. Read Memory:

Usage: mem (start address, size)
Argument: start address(Hex string/long/Hex value) from which want to read data, size- size of data want to read(int/hex/long).
Return: content at specified memory.
This command is used to read the specified size of data from specified memory location. Also it will return the msr value
Example: ami. mem ('0xffa00000',12)
ami. mem (4288675840,12)
ami. mem (0xffa00000,12)

2. Write Memory:

Usage: mem (start address, size, new value)
Argument: start address(Hex string/long/Hex value), size(int), value which want to write.
Return: true or false.
This command is used to write specified value into specified size of specified memory location.
Example: ami. mem ('0xffa00000',12,0)
ami. mem (4288675840,12,0)
ami. mem (0xffa00000,12,0)

Custom breakpoint

1. **H/w breakpoint on IO read:**

Usage: BPonIORead (address, size)
Argument: address of breakpoint(long/Hex value/Hex string).
Size of data to be read(int)
Return: None.
This command is used to add the hardware breakpoint on IO read at specified address.
Example: ami. BPonIORead ('0x80', 1)

2. **H/w breakpoint on IO write:**

Usage: BPonIOWrite (address, size)
Argument: address of breakpoint (long/Hex value/Hex string).
Size of data to be write (int)
Return: None.
This command is used to add the hardware breakpoint on IO write at specified address.
Example: ami. BPonIOWrite ('0x80', 1)

3. **H/w breakpoint on execution:**

Usage: BPonExec (address, size)
Argument: address of breakpoint (long/Hex value/Hex string).
Size of data to be read (int)
Return: None.
This command is used to add the hardware breakpoint on execution at specified address.
Example: ami. BPonExec ('0xffe350f4', 2)

4. **H/w breakpoint on data write:**

Usage: BPonDataWrite (address, size)

Argument: address of breakpoint (long/Hex value/Hex string).
Size of data to be write (int)
Return: None.
This command is used to add the hardware breakpoint on data write at specified address.
Example: ami. BPonDataWrite ('0xffe350f4', 2)

5. **H/w breakpoint on read write with no execution:**

Usage: BPonDataReadWriteNoExec (start address, size)
Argument: start address (long/Hex value/Hex string) , size (int).
Return: None
This command is used to add the hardware breakpoint on data read/write of specified address at no execution.
Example: ami. BPonDataReadWriteNoExec ('0xffe350f4', 2)

Initial Breakpoint

1. **Enable PEI Breakpoint:**

Usage: setInitBreak (True)
Argument: True
Return: None.
This command is used to enable the initial breakpoint at Pei core entry point.
Example: setInitBreak (True)

2. **Disable PEI Breakpoint:**

Usage: setInitBreak (False)
Argument: False
Return: None.
This command is used to disable the initial breakpoint at Pei core entry point.
Example: setInitBreak (False)

3. **Enable DXE Breakpoint:**

Usage: setDXEInitBreak (True)
Argument: True
Return: None.
This command is used to enable the initial breakpoint at DXE core entry point.
Example: setDXEInitBreak (True)

4. **Disable DXE Breakpoint:**

Usage: setDXEInitBreak (False)
Argument: False
Return: None.
This command is used to disable the initial breakpoint at DXE core entry point.
Example: setDXEInitBreak (True)

SMM

1. **Enable/Disable the halt at SMM entry point:**

Usage: smmentrybreak (True/False)
Argument: True – for enable and False- for disable
Return: None.
This command is used to enable/disable the halt at SMM entry point.
Example: smmentrybreak (True) –for enable smm entry break
smmentrybreak (False) –for disable smm entry break

2. **Enable/Disable the halt at SMM exit point:**

Usage: smmexitbreak (True/False)
Argument: True – for enable and False- for disable
Return: None.
This command is used to enable/disable the halt at SMM exit point.
Example: smmexitbreak (True) –for enable smm entry break
smmexitbreak (False) –for disable smm entry break

Console Redirection

1. **Open/close console redirection window:**

Usage: consoleSupport (True/False)
Argument: True – open and False - for close
Return: None.
This command is used to open/close the console redirection window. Console redirection allows to redirect the console message from target to Host and Host will display in Console redirection Window and vice versa.
Example: ami. consoleSupport (True)

Disassembly

1. **Disassem:**

Usage: asm (start address, end address [optional])
Argument: Start address (Hex string/Hex Value/long/'$') $-for current instruction
end address [optional] (Hex string/Hex Value/long) or Offset [optional] (int)
Return: None
This command is used to display the assembly instructions for a source at specified address.
Example: ami. asm ('0xffe350f4') or ami. asm ('0xffe350f4' , ' 0xffe3ffff ') or
ami. asm ('0xffe350f4', 10)

Call Stack

1. **Callstack:**

Usage: calls ()
Argument: None
Return: None
This command is used to display the details of call stack in AptioVDebugger console.
Example: ami. calls ()

Reset Target

1. **Reset:**

Usage: reset ()
Argument: None
Return: None
This command is used to reset the target system.
Example: ami. reset ()

NVRAM

1. **NVRAM variable:**

Usage: NVRAMList ()
Argument: None
Return: None
This command is used to display list of NVRAM variable in AptioVDebugger console.
Example: ami. NVRAMList ()

2. **Read NVRAM variable:**

Usage: ReadNVRAMVar (name, GUID)
Argument: name of variable (string), GUID of variable (string)
Return: True- on success and false – on failure
This command is used to display detail information of specified NVRAM variable.
Example: ami. ReadNVRAMVar ('Setup', '4599D26F-1A11-49B8-B91F-858745CFF824')

Load FV

1. **Load FV:**

Usage: LoadFV (file path)
Argument: Rom or BIN file path (string)
Return: None
This command is used to load firmware volume.
Example: ami. LoadFV ('E: \EIP\ 1ASOH030_skylake\1ASOH030.rom')

List Handle

1. **List all handle:**

Usage: listAllHandles ()
Argument: None
Return: None
This command is used to list all handles installed.
Example: ami. listAllHandles ()

2. **List handle by GUID:**

Usage: listHandles (GUID of protocol)
Argument: GUID of protocol (string)
Return: None
This command is used to list all handles which installed protocols supplied in the arguments as protocol GUID.
Example: ami. listAllHandles ('5B1B31A1-9562-11D2-8E-3F-00-A0-c9-69-72-3B')

3. **List Protocols By Handle No:**

Usage: listProtocols (handle number)
Argument: handle number - index of the handle (int)
Return: None

This command is used to list all the protocols with its name and GUID installed on Specified handle.

Example : ami. listProtocols (1)

Log Debug Message

1. **LogDbgMsg:**

Usage: commands_logging ()

Argument: None

Return: None

This command is used to log the trace messages in DebugMessageLog.txt file.

Example: ami. commands_logging ()

Trace and Checkpoint

1. **Trace Message:**

Usage: PrinttraceMessage (True/False)

Argument: True - enable, False - disable

Return: None

This command is used to enable/disable the log the trace messages in DebugMessageLog.txt file.

Example: ami.PrinttraceMessage (True)

2. **Checkpoint:**

Usage: PrintChkPt (True/False)

Argument: True - enable, False - disable

Return: None

This command is used to update the check point with a specified value by updating port 80.

Example : ami. PrintChkPt (True)

Note: Strings should be enclosed in single quotes(‘ ’).

6.9.2 Using ITP Scripts in AMI Debugger

Below are the list of ITP scripting command which can be accessed by using AMI class object. Following example will shows the how to access load command.

Bits Operation

1. **bits:**

Usage: bits (object, offset, size, newValue)

Arguments: object - It can be a valid register name expressed as a string, an arbitrary integer, or a BitData object or newly created AMI datatypes

offset - A valid expression yielding the bit index into the specified value.

size - A valid expression yielding the size, in bits, of the bit field.

newValue - If specified, a numerical value to be assigned to the bit field.

Returns: If newValue is None, returns a BitData object containing the requested bits.

If newValue is not none and register is an integer, then returns a BitData object containing the updated integer.

Otherwise, the register or provided BitData object is changed in place and nothing is returned.

This command is used to View or change a range of bits in a value. It can also be used for in-place changes of BitData and AMI Data Types and Register values.

Example: x=BitData(16,0x14)

print(itp.bits(x,0,4))

Disassembly

1. **asm:**

Usage: asm (address, args)

Arguments: address - valid address expressed as string

args - it can be a valid integer or end address as string

Returns: None

This command helps to display the assembly instruction.

Example: itp.asm ('ffe350d0')

Itp.asm ('ffe350d0', 10)

Invalidate Cache

1. **invd:**

Usage: invd ()
Arguments: None
Returns: None

Invalidate IA32 processor cache.

Example: itp.invd ()

2. **wbinvd:**

Usage: wbinvd ()
Arguments: None
Returns: None

Write back modified data from the IA32 caches to main memory and invalidates the caches.

Example: itp.wbinvd ()

UseMode

1. **usemode:**

Usage: usemode ()
Arguments: None
Returns: None

This command helps user to display the current IA32 instruction mode.

Example: itp.usemode ()

2. **usemoderaw:**

Usage: usemoderaw ()
Arguments: None
Returns: instruction mode

This command returns the current IA32 instruction mode as a numerical value.

Example: itp.usemoderaw ()

Timer

1. **Timer:**

Usage: Timer ()
Arguments: None
Returns: None
This command performs basic timer operation.

Example: itp.Timer ()

2. **wait:**

Usage: wait (time)
Arguments: time
Returns: None
Suspend script execution until a break has occurred on the target for the specific thread or the specified number of seconds has passed.

Example: itp.wait (10)
Itp.wait()

3. **time:**

Usage: time ()
Arguments: None
Returns: None
This command is used to print the current system time.

Example: itp.time ()

4. **sleep:**

Usage: sleep (time)
Arguments: time in seconds
Returns: None

This command is used to stop the script execution for the specific seconds. Suspend script execution until a break has occurred on the target or the specified number of seconds has passed.
Example: itp.sleep (10)

Expression

1. **eval:**

Usage: eval (argument)
Arguments: expression to evaluate
Returns: None

This command helps user to evaluate an address or expression and display the result. If the expression is numerical, then it is converted to binary, decimal, and hexadecimal. If the expression is a string and cannot be converted to a number, then the bytes of the string are displayed. If the expression contains a floating point value, then the bytes of the floating point value are displayed.
Example: itp.eval ('$')
Itp.eval ('1000L')

Append

1. **append:**

Usage: append ("filename", "objectname")
Arguments: filename -- Name of the file to append to. Must be enclosed in quotes.
objectname -- Name of the object to get source for. Must be enclosed in quotes.
Returns: None

Appends source code for an object into a file.
Example: x=10
itp.append ('D:\value.txt','x')

Status

1. **status:**

Usage: status ()
Arguments: None
Returns: None
This command is used to display the current target state and current phase.
Example: itp.status ()

2. **hwstatus:**

Usage: hwstatus ()
Arguments: None
Returns: None

Display the current target state, current connection details and firmware version of AMIDebugRx.
Example: itp.hwstatus ()

Log

1. **commands_logging:**

Usage: commands_logging ()
Arguments: None
Returns: None

This command is used to log the trace messages in DebugMessageLog.txt file.
Example: itp.commands_logging ()

2. **stopdallog:**

Usage: stopdallog ()
Arguments: None
Returns: None

This command is used to stop logging the trace messages in DebugMessageLog.txt file.
Example: itp.stopdallog ()

Callstack

1. **calls:**

Usage: calls ()
Arguments: None
Returns: None

This command helps user to observe the details of call stack in APtioVDebugger console.
Example: itp.calls ()

Register

1. **reg:**

Usage: reg (val, [newVal])
Arguments: val – Register name (String format)
newVal – register value to update (int/Hex/Hex String)
Returns: Register value

This command is used to get and alter the value of the particular register.
Example: itp.reg ('AX')
itp.reg ('DS',15)

2. **regs:**

Usage: regs ()
Arguments: None
Returns: None

This command is used to display register set in AptioVDebugger console.
Example: itp.regs ()

3. **xmregs:**

Usage: xmregs ()
Arguments: None
Returns: None

This command is used to display extended register set in AptioVDebugger console.
Example: itp.xmregs ()

4. **display:**

Usage: display (registers)
Arguments: register names
Returns: None

This command is used to display the values of given register names.
Example: itp.display ()
itp.display ('DS')

Stepping operation

1. **step:**

Usage: step (type, steps)
Arguments: type of stepping
steps count
Returns: None

This command is used to perform a single step of program execution.
Example: itp.step (1,4)
Itp.step ()

2. **istep:**

Usage: istep (steps)
Arguments: steps count
Returns: None

This command is used to perform a single step of machine-language instruction.
Example: itp.istep ()
Itp.istep (2)

3. **go:**

Usage: go ([address])
Arguments: address – line address

Returns: None

This command is used to resume the execution and Run to particular address or Line.
Example: itp.go ()

Breakpoints

1. **br:**

Usage: br ()
Arguments: None
Returns: None
This command helps user to observe the available breakpoints in console.
Example: itp.br ()

2. **brremove:**

Usage: brremove (breakpointIDs)
Arguments: breakpointIDs - IDs of zero or more breakpoints to enable. The BreakpointID can be a string or a number starting from 1.
Returns: None

This command helps user to remove breakpoints by id.
Example: itp.brremove (1)

3. **brdisable:**

Usage: brdisable (breakpointIDs)
Arguments: breakpointIDs - IDs of zero or more breakpoints to enable. The BreakpointID can be a string or a number starting from 1.
Returns: None

This command helps user to disable breakpoints by id.
Example: itp.brdisable (1)

4. **brenable:**

Usage: brenable (breakpointIDs)

Arguments: breakpointIDs - IDs of zero or more breakpoints to enable. The BreakpointID can be a string or a number starting from 1.
Returns: None

This command helps user to enable breakpoints by id.
Example: itp.brenable (1)

5. **brget:**

Usage: brget (breakpointIDs)
Arguments: breakpointIDs - IDs of zero or more breakpoints to enable. The BreakpointID can be a string or a number starting from 1.
Returns: None

This command helps user to get details of breakpoints by id.
Example: itp.brget (1)

6. **cause:**

Usage: cause ()
Arguments: None
Returns: None

This command helps to display last breakpoint detection details in console.
Example: itp.cause ()

7. **reset:**

Usage: reset ()
Arguments: None
Returns: None

This command is used to reset the target system.
Example: itp.reset ()

8. **pulsepwrgood:**

Usage: pulsepwrgood ()

Arguments: None
Returns: None

This command is used to reset the target. Pulse the Power Good Reset (pwrgood#) signal to the target(s), causing the system to go through a power cycle.

Example: itp.pulsepwrgood ()

9. halt:

Usage: halt ()
Arguments: None
Returns: None

This command is used to halt the target from the current execution.
Example: itp.halt ()

10. halttimeout:

Usage: halttimeout (timeout)
Arguments: timeout
Returns: None
This command is used to change the timeout (in ms) for the halt command.

Example: itp.halttimeout (10)

11. num_halt_retries:

Usage: num_halt_retries (retry)
Arguments: retry count
Returns: None

This command is used to change the number of retries for the halt command.
Example: itp.num_halt_retries (3)

12. exit:

Usage: exit ()
Arguments: None
Returns: None

This command helps to invoke stop debug action.
Example: itp.exit ()

13. stopmasterframe:

Usage: stopmasterframe ()
Arguments: None
Returns: None

This command helps to invoke stop debug action.
Example: itp.stopmasterframe ()

14. unloadsymbols:

Usage: unloadsymbols ()
Arguments: None
Returns: None

This command helps user unload the loaded symbols.
Example: itp.unloadsymbols ()

Memory

1. mem:

Usage: mem (address,datasize,[val])
Arguments: address - Base address from where to start reading or writing
datasize - 1,2 or 4 based on byte,word,DWord
val - bytes to write, none in case of read
Returns: Read value (if no write value specified)

This command reads or writes to a specified memory based on the arguments passed.
Example: itp.mem ('ffa00010', 8) – read memory
itp.mem ('ffa00000', 1, 15) – write memory

2. **memblock:**

Usage: memblock (address, addressOrCount, datasize,[val])

Arguments: address - Base address from where to start reading or writing (Hex String/long/Hex value)

addressOrCount - If a string, then this is the "to" address marking the end of the memory block (inclusive); otherwise, this is the number of elements to read.

datasize - 1,2 or 4 based on byte,word,DWord

val - bytes to write, none in case of read

Returns: Return a BitData object containing the values from memory if reading the memory; otherwise, returns None.

This command is used to read or write range of values to memory.

Example: itp.memblock (‘0xFFA0000’, 4, 1)

itp.memblock('0xFFE33CB4','0xFFE33CB8',2)

3. **memdump:**

Usage: memdump (address, addressOrCount, datasize)

Arguments: address - Base address from where to start reading or writing

addressOrCount - If a string, then this is the "to" address marking the end of the memory block (inclusive); otherwise, this is the number of elements to read.

datasize - 1,2 or 4 based on byte,word,DWord

Returns: None

This command is used to display the contents of a block of memory.

Example: itp.memdump (‘0xFFA0000’, 4, 1)

4. **memsave:**

Usage: memsave (filename, address, addressOrCount, overwrite)

Arguments: filename - file where the data is to be stored.

address - Address of the first byte to read from.

addressOrCount - To address or count for number of elements save

overwrite – true overwrite file

false append, By default false (it will append)

Returns: None

This command is used to save a range of target memory to a host file.

Example: itp.memsave ('D:\file.txt','0x1000','0x1100',True)

5. **memload:**

Usage: memload (filename, address, addressOrCount)

Arguments: filename - file from where the data is read.

address - Address of the first byte to read from.(Long/Hex/Hex String)

addressOrCount - To address or count for number of elements to load

Returns: None

This command is used to fill a range of target memory with the contents of a host file.

Example: itp.memload ('D:\file.txt','0x1000',’0x1100’)

6. **upload:**

Usage: upload (filename, address, addressOrCount, overwrite)

Arguments: filename : Filename of the object file where the information is stored.

address : Address of the first byte to read from.

addressOrCount : To address or count for number of elements save

overwrite : True to overwrite the file if it exists; otherwise, an error is raised.

Returns: None

This command is used to save contents of selected portions of target memory to a host file.

Example: itp.upload ('D:\file.txt','0x1000',’0x1100’)

7. **memoryscandelay:**

Usage: memoryscandelay (delay)

Arguments: delay time

Returns: None

This command configures a memory scan delay.

Example: itp. memoryscandelay (100)

8. **forcememoryscandelay:**

Usage: forcememoryscandelay (delay)
Arguments: delay time
Returns: None

This command configures a memory scan delay instead of triggered scans. If this is set to true, it will force memory scan delays for all memory access operations, instead of a triggered scan that would normally be done in order to verify the completion of the memory scan.
Example: itp. forcememoryscandelay (100)

CPUID

1. **cpuid_eax:**

Usage: cpuid_eax (eax, [ecx])
Arguments: eax address
ecx address
Returns: Processor information stored in eax register in BitData Format.

Return the processor identification and feature information stored in the EAX register, where the EAX register is initialized to 1.
Example: itp.cpuid_eax (2)

2. **cpuid_ebx:**

Usage: cpuid_ebx (eax, [ecx])
Arguments: eax address
ecx address
Returns: Processor information stored in ebx register in BitData Format.

Return the processor identification and feature information stored in the EBX register, where the EAX register is initialized to 1.
Example: itp.cpuid_ebx (2)

3. **cpuid_ecx:**

Usage: cpuid_ecx (eax, [ecx])
Arguments: eax address
ecx address
Returns: Processor information stored in ecx register in BitData Format.

Return the processor identification and feature information stored in the ECX register, where the EAX register is initialized to 1.
Example: itp.cpuid_ecx (2)

4. **cpuid_edx:**

Usage: cpuid_edx (eax, [ecx])
Arguments: eax address
ecx address
Returns: Processor information stored in edx register in BitData Format.

Return the processor identification and feature information stored in the EDX register, where the EAX register is initialized to 1.
Example: itp.cpuid_edx (2)

PCI

1. **pci_config:**

Usage: pci_config (bus, dev, fnc, reg[, num[, val]])
Arguments: bus - PCI Bus {0-0xff}
dev - PCI Device {0-0x1f}
fnc - PCI Function {0-7}
reg - Register offset {0-0xff (standard) or 0xfff (enhanced)}
num - Number of bytes {1, 2, 4 (default), 8, 16}
val - Write value {0-0xff, 0-0xffff, 0-0xffffffff or
0-0xffffffffffff (8 bytes), or
0-0xffffffffffffffffffffffff (16 bytes)}
All parameters can be given into int/Hex/Hex string format
Returns: Read value (if no write value specified)

This command reads or writes to a specified configuration register based on the arguments passed into the script.

Example: itp. pci_ config (0, 0, 0, 0) – for reading or
itp. pci_ config ('0x0', '0x1', '0x1', '0x0')
itp. pci_ config (0, 0, 0, 0, 0x12) – for writing

2. pci_configb:

Usage: pci_configb (bus, dev, fnc, reg[, val]])
Arguments: bus - PCI Bus {0-0xff}
dev - PCI Device {0-0x1f}
fnc - PCI Function {0-7}
reg - Register offset {0-0xff (standard) or 0xfff (enhanced)}
val - Write value {0-0xff, 0-0xffff, 0-0xffffffff or
0-0xffffffffffff (8 bytes), or
0-0xffffffffffffffffffffffff (16 bytes)}
All parameters can be given into int/Hex/Hex string format
Returns: Read BYTE value of the register offset(if no write value specified)

This command reads or writes a single byte value to a specified configuration register based on the arguments passed into the script.

Example: itp. pci_configb (0, 0, 0, 0) – for reading or
itp. pci_configb (0, 0, 0, 0, 0x12) – for writing

3. pci_configw:

Usage: pci_configw (bus, dev, fnc, reg[, val]])
Arguments: bus - PCI Bus {0-0xff}
dev - PCI Device {0-0x1f}
fnc - PCI Function {0-7}
reg - Register offset {0-0xff (standard) or 0xfff (enhanced)}
val - Write value {0-0xff, 0-0xffff, 0-0xffffffff or
0-0xffffffffffff (8 bytes), or

0-0xffffffffffffffffffffffff (16 bytes)}
All parameters can be given into int/Hex/Hex string format
Returns: Read WORD value of the register offset (if no write value specified)

This command reads or writes a single word value to a specified configuration register based on the arguments passed into the script.
Example: itp. pci_configw (0, 0, 0, 0) – for reading or
itp. pci_configw (0, 0, 0, 0, 0x1234) – for writing

4. **pci_configd:**

Usage: pci_configd (bus, dev, fnc, reg[, val]])
Arguments: bus - PCI Bus {0-0xff}
dev - PCI Device {0-0x1f}
fnc - PCI Function {0-7}
reg - Register offset {0-0xff (standard) or 0xfff (enhanced)}
val - Write value {0-0xff, 0-0xffff, 0-0xffffffff or
0-0xffffffffffff (8 bytes), or
0-0xffffffffffffffffffffffff (16 bytes)}
All parameters can be given into int/Hex/Hex string format
Returns: Read WORD value of the register offset (if no write value specified)

This command reads or writes a single double word value to a specified configuration register based on the arguments passed into the script.
Example: itp. pci_configd (0, 0, 0, 0) – for reading or
itp. pci_configd (0, 0, 0, 0, 0x123421) – for writing

MSR

1. **msr:**

Usage: msr ()
Arguments: MSR address, [optional] MSR value
Returns: MSR value for read; none for write.
Notes: Reads msr (msr_addr); Write: msr (msr_addr, msr_val)

Reads and writes specified values to a specified MSR address. For reads, it returns the value in the specified address. For writes, the value of the second argument of the script is written to the MSR address specified by the first argument of the script.

Example: itp. msr (0x17) – for reading or
itp. msr (0x17,1234) – for writing

6.9.3 Sample Scripts

System Info:

Usage: sysinfo ()
Argument: None
Return: None

This command is used to get the current target system info.
Example: ami.load ('Sysinfo.py') -Load the Sysinfo.py script file
sysinfo ()

PCI Info:

Usage: pciinfo ()
Argument: None
Return: None

This command is used to print the list of all valid pci devices with their detail vendor ID, device ID, class code, Prog Interface.
Example: ami.load ('PciInfo.py') -Load the PciInfo.py script file
pciinfo()

CPUID Info:

Usage: cpuid ()
Argument: None
Return: None

This command is used to print CPU detail such as manufactural detail, cache size, version.
Example: ami.load ('CPUID.py') -Load the CPUID.py script file
cpuid()

SMM Debugging

Halting on SMM Entry\Exit

Select: Debug→SMM→Break on SMM Entry from Menu item or Select SMM Entry from Toolbar Icon [] to halt at SMM entry.

NOTE: In Aptio V BIOS, the AptioV Debugger will halt on SMM entry when InitializeDebugAgent(DEBUG_AGENT_INIT_ENTER_SMI, NULL, NULL) is called.

NOTE: User can select/Remove SMM Entry only when Target is in halted state.

Select : Debug →SMM→Break on SMM Exit to halt at SMM exit

NOTE: In Aptio V BIOS, the AptioV Debugger will halt on SMM exit when InitializeDebugAgent(DEBUG_AGENT_INIT_EXIT_SMI, NULL, NULL) is called.

NOTE: User can select/Remove SMM Exit only when Target is in break state.

SMM Outside Context

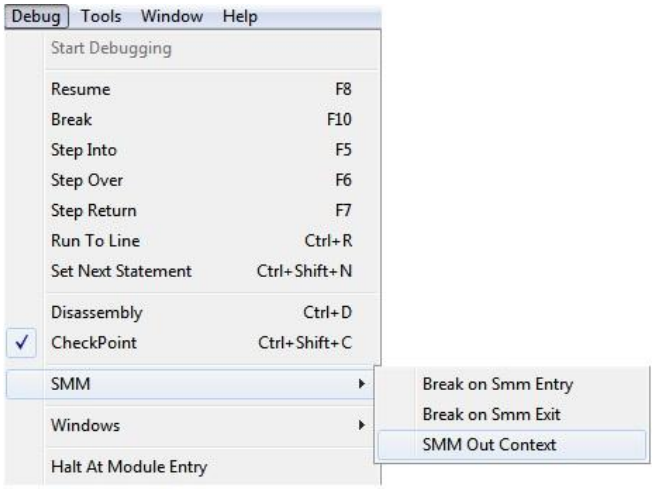
Select: Debug→SMM→SMM Out Context or selection using debugger tool bar icon []

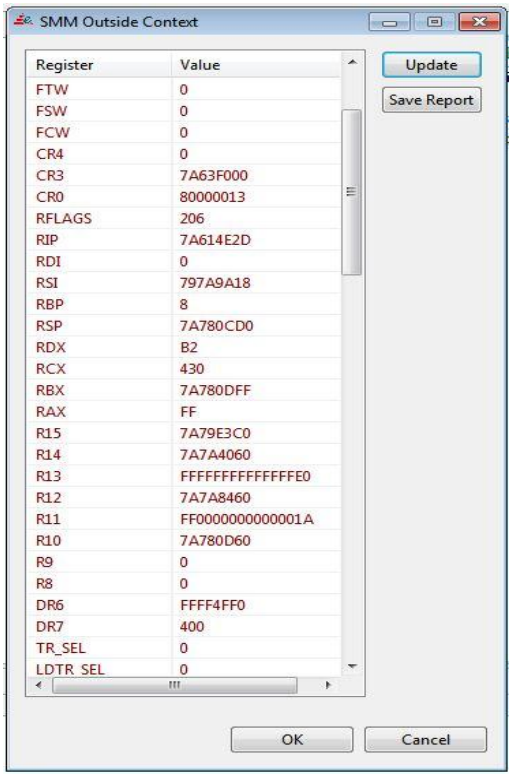
This will open the SMM outside Context window which display the all the registers and their values, when target is in SMM phase.

By using “Update” button we can get updated value of register.

In addition, we can save the all register data in text file by using “Save Report” button.

Below screenshot is showing the SMM outside context.





NOTE: For SMM Out Context, Target should be in SMM mode.

NOTE: On the first halt at SMM Entry debugger shows all zeros in SMM Outside Context window, because the SMM Outside context is not yet initialized by the SMM core. User can be able to see the Outside context from the second SMM Entry halt onwards.

Start Debugging

Click Start Debugging on the Debug menu or select debugger tool bar icon [ ] to start Debugging Session. By selecting this option, Debugger Host Application will be initialized and wait for the target to start. Selecting Start Debugging option gray out the debugging features including stop debugging.

Execution Control

Resume

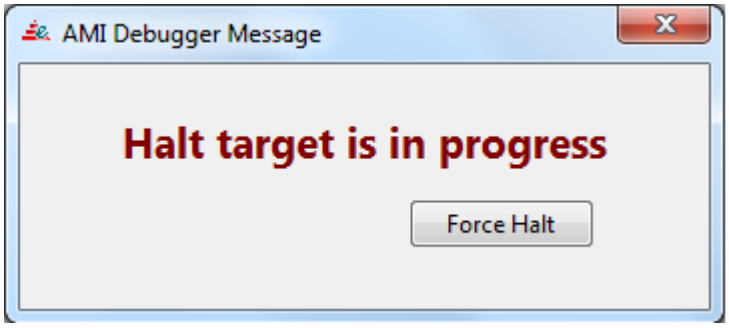
If process is halt at breakpoint, you can resume the execution by selecting Resume on Debug Menu or by pressing **F8** or by selection tool bar icon [ ]. The execution of the application continues until another breakpoint is encountered. In disassembly mode, Resume at Line works similar to Resume.

Break

When the target is running, most debugger actions are unavailable. If you want to stop a running target, you can issue a Break command by clicking on Break on Debug Menu or by selecting tool bar icon [ ] or by using **F10**. This command causes the debugger to break into the target. That is, the debugger stops the target and all control is given to the debugger. If a running target encounters an exception, if certain events occur, if a breakpoint is hit, or if the application closes normally, the target breaks into the debugger and a message appears in the Debugger Command window and describes the error, event, or breakpoint.

Note:

When halt driver is enabled, on pressing break, the host would wait for the next breakpoint to happen from the target with the following message box. If the user want to break the target forcefully as in case of an infinite while loop added for debug or in shell, user may use force halt. Pressing force halt when multiple drivers are getting loaded would lead into inconsistent behavior.



Reset Target

Click Reset Target on the Debug menu or select debugger tool bar icon [ ] or the hotkey **F12** to reset the target

Stepping Controls

Step Into

Click Step Into on the Debug menu or select debugger tool bar icon [] or use the hot key **F5** to execute a single instruction on the target.

If a function call occurs, Step Into enters the function and continues stepping through each instruction.

Step Over

Click Step Over on the Debug menu or select debugger tool bar icon [] or use the hot key **F6** to execute a single instruction on the target. If the instruction is a function call, the whole function is executed .Step over treats the function call as a single step. When the debugger is in Assembly Mode, stepping occurs one machine instruction at a time. When the debugger is in Source Mode, stepping occurs one source line at a time.

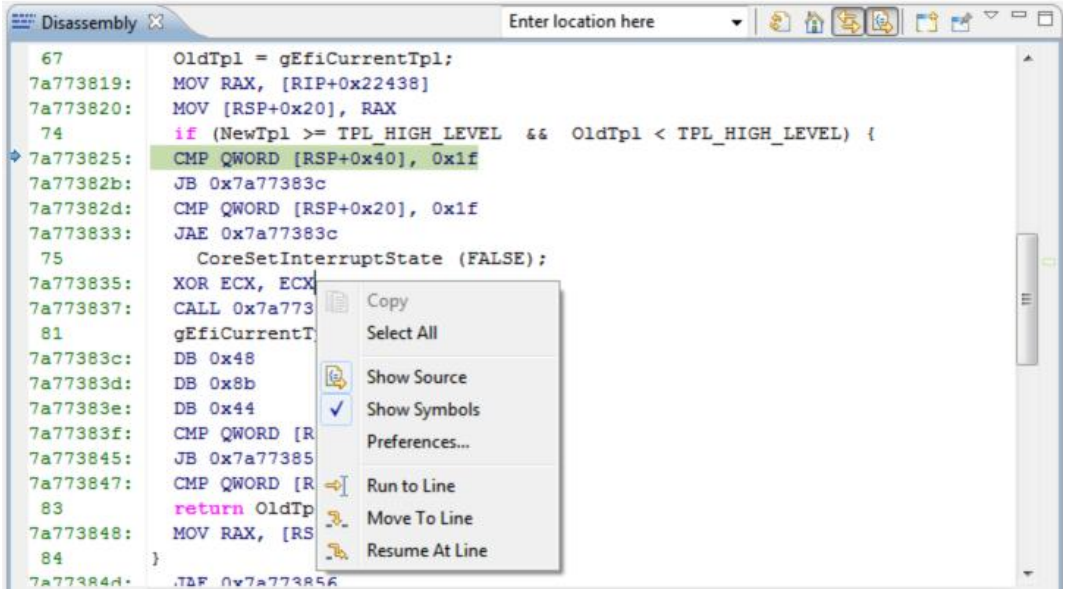
Step Return

Click Step Return on the Debug menu or select debugger tool bar icon [] or use the hot key **F7** to execute Step Return. It is used to return from a method, which has been stepped into. Even though we return from the method, the remainder of the code inside the method will be executed normally.

Run To Line

In the Source View, the user can select on a valid line in the execution control and select the Run to Line option from the Debug Menu or use the hot key **CTRL+R** or using tool bar icon []. The Disassembly view provides the option to 'Run to Line' using its Right click menu option Like in below image. Selecting this option will runs code until the selected line encounters and halt at the selected line.

Run to Line is very useful when it comes to skipping over parts of a function that you do not want to step through. For example, if you have a loop in your method and you do not want to step through the loop, select a line just after the loop and select Run to Line (CTRL+R).



Set Next Statement

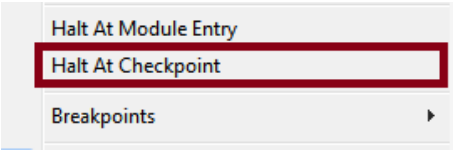
Set Next Statement is frequently used in conjunction with edit and continue. That is, after changing some code during the debug session, you can set the next statement to the location before the change and re-execute the code to see the effect of your changes. Set Next Statement allows you to jump around in a method without executing any code in between. You can also use set next statement to jump over code that you believe is buggy, or to jump past a conditional test on an if statement to execute a path of code that you want to debug without having to make the condition true.

In the Source View, the user can select on a valid line in the execution control and select the Set Next Statement option from the Debug Menu or select debugger icon [ ]or use the hot key **CTRL+SHIFT+N**. The Disassembly view provides the option to 'Move To Line' using its Right click menu option, like in above image. Move to Line works similar to Set Next Statement option.

Halt Checkpoint

Users can access the Halt At Checkpoint by -

- Select : Select: *Debug →Halt At Checkpoint*



- Or using the Debug Toolbar Icon [ ]

  In the Halt Checkpoint View, user can specify to halt the target on the occurrence of a Port 80 checkpoint update.

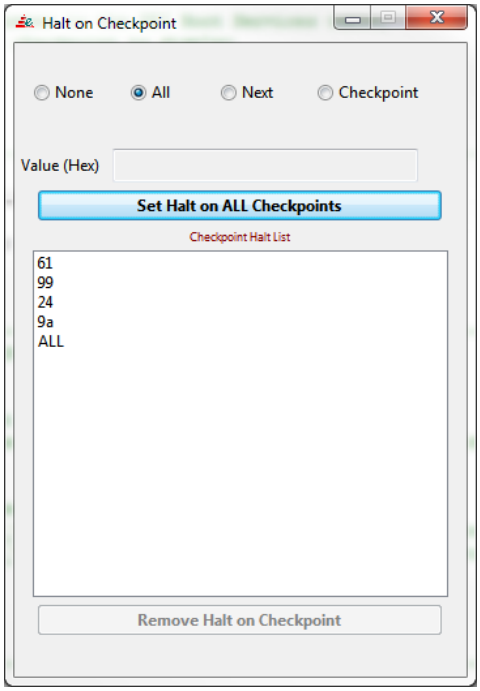


Figure: Halt Checkpoint View

Options

None - Does not halt on any Checkpoint, or Clears existing checkpoint halts.

All - Halts on the occurrence of all checkpoints

Next - Halts at next Checkpoint only, then Checkpoint Halt 'Next' is cleared.

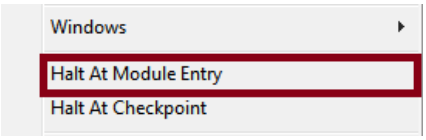
Checkpoint - Halts at user specified Checkpoints only.

NOTE: Halt Checkpoint names/state are persisted across Target reboots, therefore the user need to disable the checkpoint halt if not required during Target reboots.

Halt Module

Users can access the Haltdriver window by -

- Select : Select: *Debug →Halt At Module Entry*



- Or using the Debug Toolbar Icon [ ]

In the module selection window, provide the module name and select “ok”. Users can select the type of Halt driver option they want (like All, Next) or select ‘Driver Name’ and type in the drivername to halt on.

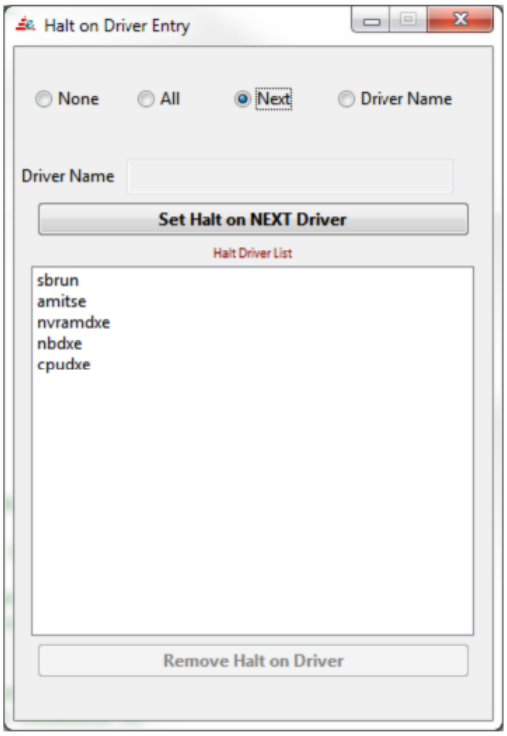


Figure: Halt Module

Options

None – Does not halt on any Module Entry, previous halts are all cleared, acts like Halt Driver OFF.

All – Halt at all loaded drivers entry points.

Next - Halts only at next loaded driver's entry point.

Driver Name - Halts at user specified driver's Entry Point only, displayed in the Halt Driver List.

For instance, user provides Halt all and in the subsequent entry point user Halt All removed and is modified as Halt Next. Halt module will no longer halt at all modules but will only halt at the next loaded halt module. The behavior is same as Halt Driver->Next.

> ☛ *NOTE: Halt module names/state are persisted across Target reboots, therefore the user need to disable the Halt if not required during target reboots.*
>
> ☛ *Note: The driver name is the final efi image name such as "CpuPei" (without the quote).*

Stop Debugging

Click Stop Debugging on the Debug menu or select tool bar icon [] or use the hotkey **CTRL+F12** to stop the target's execution, clear the debugger. It ends the target process and all its threads.

Ability to boot if AMIDebugRx is not connected

Target boots normally if AMIDebugRX device is not connected

Toolbar Options

Command	Menu Option	Shortcut Key	Icon	Description
Start Debug	Debug->Start Debug			Starts a debugging session
Resume	Debug→Resume	F8		To Proceed Execution
Step Into	Debug→Step Into	F5		To jump one instruction step from current step
Step Over	Debug → Step Over	F6		Moves to next source line

Step Return	Debug→Step Return	F7		Select the Step Return command to return from a method, which has been stepped into.
Run To Line	Debug→ Run To Line	CTRL+R		Select the Run To Line command to have program execution run to the line number
Set Next Statement	Debug→Set Next Statement	CTRL+SHIFT+N		Sets the execution point to the line of code you choose.
Show Disassembly	Debug→Disassembly	CTRL+D		Selecting the menu opens up the disassembly view window.
Checkpoint window	Debug→Checkpoint	CTRL+SHIFT+C		Show/Hide checkpoint window
Break on SMM Exit	Debug→SMM->Break on SMM Entry			Enables/Disables break on SMM entry
Break on SMM Exit	Debug→SMM->Break on SMM Exit			Enables/Disables break on SMM exit
SMM out context	Debug->SMM->SMM Out Context			Window displays the SMM Outside Context.
Registers Window	Debug → Windows-> Registers	CTRL+SHIFT+R		Show/Hide register window
Memory Window	Debug→Windows→Memory	CTRL+SHIFT+M		Show/Hide memory window
Variables Window	Debug→Windows→Variables	CTRL+SHIFT+V		Show/Hide local Variable window
Expression Window	Debug→Windows→Expression	CTRL+SHIFT+E		Show/Hide Expression window
NVRAM Variable View	Debug→Windows-> NVRAM Variable			Show/Hide NVRAM Variable view
PCI View	Debug→Windows→PCI	CTRL+SHIFT+P		Show/Hide PCI view.
IO View	Debug→Windows→IO	CTRL+SHIFT+I		Show/Hide IO view.
CPUID View	Debug→Windows→CPUID			Show\Hide CPUID View
MSR View	Debug→Windows→MSR			Show\Hide MSR View
IIO View	Debug→Windows→IIO			Show\Hide IIO View

Console redirection	Debug→Windows→Console redirection	CTRL+SHIFT+T		Display console redirection view
Performance measurement	Debug→Windows→Performance measurement			Display to performance measurement window
Halt Module window	Debug->Halt at module entry			Display window to configure the halt at module entry
Halt Checkpoint window	Debug->Halt at checkpoint			Display window to configure the halt at checkpoint
Breakpoints View	Debug→Windows→breakpoints	CTRL+SHIFT+B		Show/Hide breakpoints view
Add Custom Breakpoints	Debug->Add custom breakpoints	F9		Add a custom HW/SW breakpoints
Add Inspection point	Debug-> Breakpoints-> Add Inspection point			Display window to add the inspection point
List loaded driver	Debug->List Loaded Drivers			Displays the Loaded driver Explorer view
LoadFV	Debug->LoadFV			Opens Windows explorer to load FV_MAIN image file or a ROM Image
List Handle view	Debug->List handles			Show/Hide EFI window
Log Debug Messages	Debug->Log Debug Messages			Logs Debug message to DbgmsgLog file
Enable PEI breakpoint	Debug->Initial breakpoints->PEI Breakpoints			Enable/Disable initial breakpoint in PEI phase
Enable DXE breakpoint	Debug->Initial breakpoints->DXE Breakpoints			Enable/Disable initial breakpoint in DXE phase
Reset Target	Debug→Reset	F12		To Reset the target
Stop Debugging	Debug→Stop Debugging	CTRL+F12		To stop the Debug Session.
Debug Option	Debug->Option			Show the Debug option window

DCI Debugger

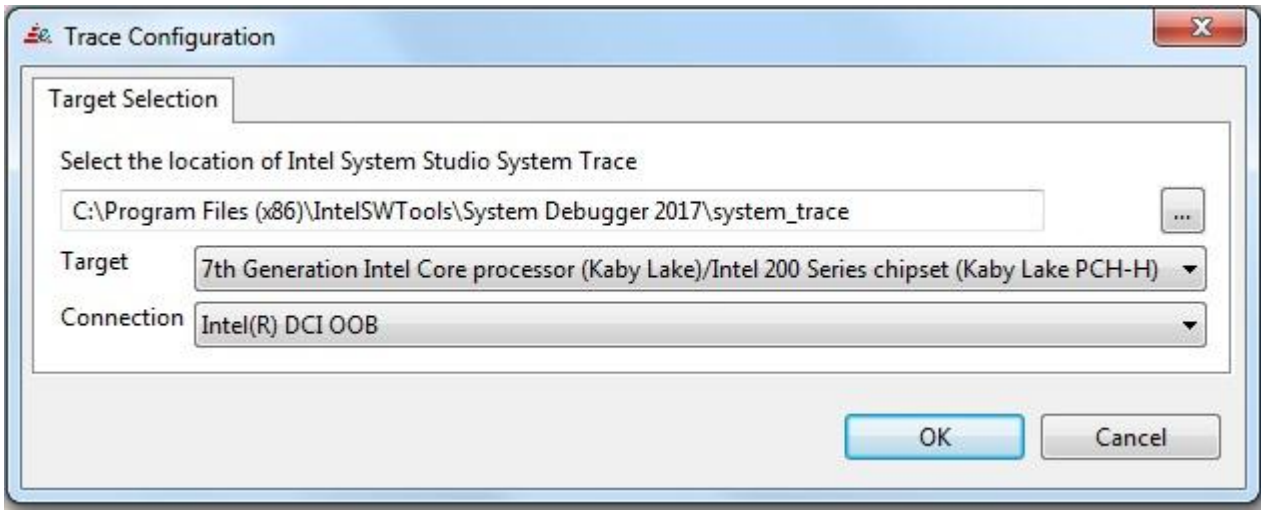
Host software requirements

- PC Host with one of the following version of Windows OS 7\8\8.1(32 bit OS is not supported)
- Intel System Studio 2017 (Trial version is enough) - [Link]
- Microsoft Visual C++ 2015 Redistributable x86 - [Link]

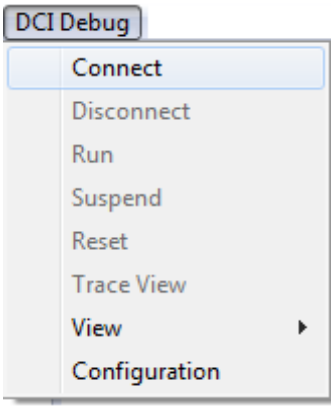
DCI Debugger Usage

In order to use DCI debugger, follow the below steps:

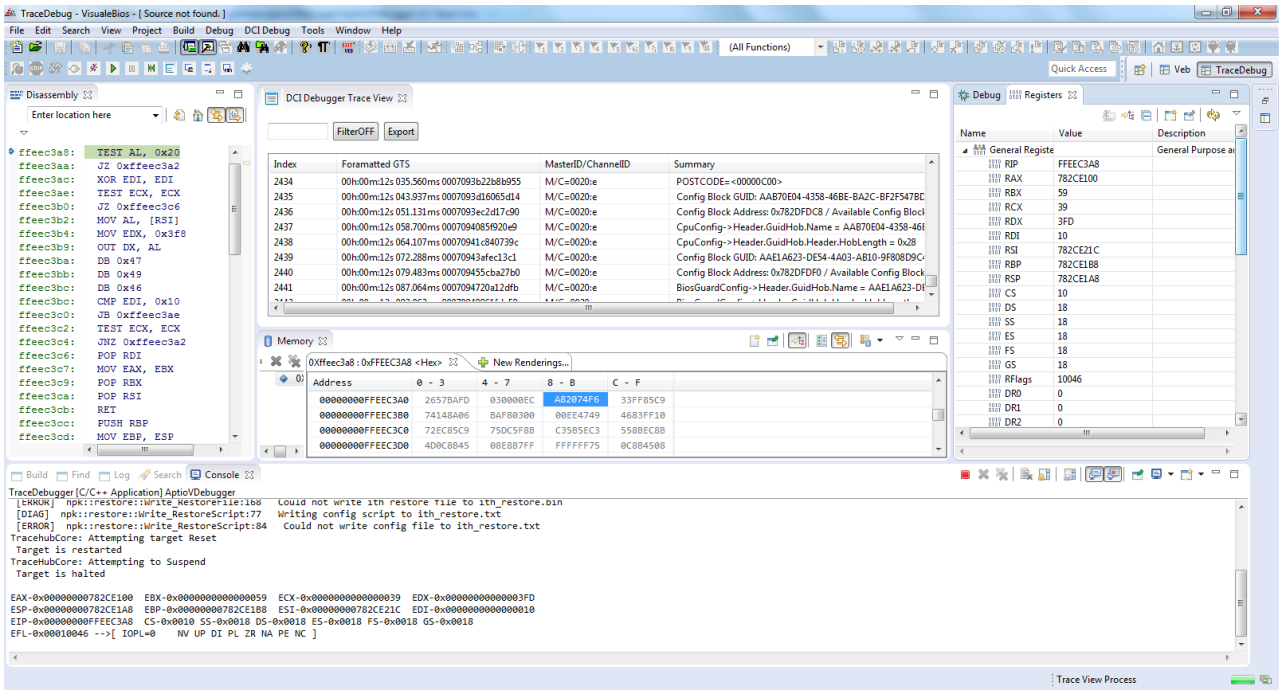
1. Flash the debug mode enabled image (Debugger Modules are not required) in SUT and power on the target and enable the DCI debugger from bios setup (see the setting up Bios section).
2. Connect the host machine and target platform using Trace Hub device (CCA device).
3. Click on configuration from toolbar/DCI Debug Menu, and select the target configuration and connection type.



4. Click on the connect option and wait for connection. In AptioV console user can see the connection status whether connection made success full or not. For success case, connected [OK].



5. Once connected, VeB layout will be as shown below.



6. User can see the traces in trace view.
7. In order to suspend the target, click on suspend option
8. After suspending target, user can see the CPU register content and Traces will stop coming.
9. User can see the memory content at specified location, by selecting the memory view.
10. Also, user can see the disassembly code in disassembly view.
11. User can resume the target, by clicking on run option, then again traces will continue updating.

12. In order to reset the target, click on reset option.

13. To disconnect the connection between host and target, click on disconnect  .

Setting up BIOS for DCI Debugger

Boot the board till Bios Setup page then configure the below things:

1. ChipSet Menu → PCH-IO Configuration → Trace Hub Configuration Menu → Trace Hub Enable Mode →Set to value <Host Debugger>
2. ChipSet Menu → PCH-IO Configuration → DCI Enable (HDCIEN) → Set to value <Enabled>
3. Advanced Menu → CPU Configuration → Debug Interface → Set to value <Enabled>
4. Advanced Menu → CPU Configuration → Direct Connect Interface → Set to value<Enabled>

Bios Module Optimization

For source level debugging, always set OPTIMIZATION Token as OFF. If you are unable to disable the OPTIMIZATION token then add the below preprocessor directives in source code for debug.

#pragma optimize ("", off)

.....

.....

#pragma optimize ("", on)

If user may face problem while setting breakpoint and get error message as “No Address associated with source line”. Reason of this error is optimization of code. To fix this issue, either disable the optimization token or use the above-mentioned preprocessor directive.

If user needs to set breakpoint in code at compile time itself then add the __debugbreak() breakpoint macro. This is required for break in SMM mode.

For source level debugging, when Debug Mode token is disabled then use the following configuration in BoardCommonFeatureToken.sdl

```
ELINK
    Name = "*_*_*_DLINK_FLAGS"
    Type = "BuildOptions"
    Arch = "COMMON"
    InvokeOrder = ReplaceParent
    OutDSC = Yes
End
ELINK
    Name = "/DEBUG /PDB:$(DEBUG_DIR)/$(BASE_NAME).pdb"
    Parent = "*_*_*_DLINK_FLAGS"
    Type = "BuildOptions"
    InvokeOrder = AfterParent
End
ELINK
    Name = "/Zi"
    Parent = "*_*_*_CC_FLAGS"
    Type = "BuildOptions"
```

```
    InvokeOrder = AfterParent
End
```

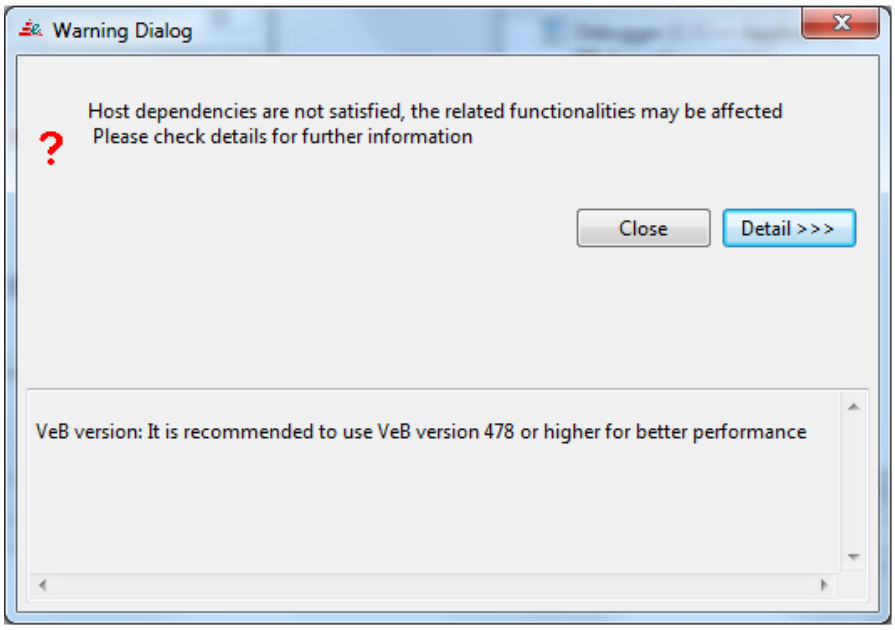
Troubleshooting with AptioVDebugger

Installation Related Issues

Troubleshooting for Host Side Prerequisite Check:

Host side support has been added to validate the basic prerequisite needed for debugger host to work. When the user presses start debugging, the requirements are evaluated and user is shown with a popup message if some requirements are not met. Following are the cases in which the popup would appear:

1. USB 2 Driver not found : Please install AMIDebugRxDriver. Driver is available with Debugger eModule
2. USB 3 Driver not found: Please install AMI USB3 Driver. Driver is available with Debugger eModule
3. Microsoft VC++ redistributable 2010 not found : Please install Microsoft Visual C++ 2010 Redistributable Package (x86)
4. Microsoft VC++ redistributable 2013 not found : Please install Microsoft Visual C++ 2013 Redistributable Package (x86)
5. VEB version need update : Minimal version recommended is 7.19. Stepping Into or above for optimal performance. User may upgrade the VEB version used to a newer version for best performance.
6. OS Not supported : Support of debugger validated with Windows OS. Any other OS may show up this message.



Troubleshooting for debugger installation:

While installation of debugger, sometime user may face some trouble. In such case, please check the following point:

- **Registering Components failed:** This error occurs in Windows 7 when the command prompt is not launched in admin mode. Therefore please ensure to **open the command prompt in Admin mode**. Change directory where AptioVDebugger Component was downloaded. Run install.bat – **"update.bat /f <vebpath>"** (Referring installation instruction step #3). When the batch prompts for Overwrite confirmation, Press A indicating All.
- **Duplication of toolbar:** Sometimes, after installation the debugger plugin user may face duplication of toolbar issue. In order to resolve this issue **use the latest VeB** or **increase the Veb version**.
- **Debug menu is not visible:** Sometime after installation of AptioVDebugger,user may face trouble that debug menu is not visible in veb.
    - Ensure that **VisualeBios version >= 7.15.0166**
    - For debugger version AptioVDebugger 3.02.0017 onwards with Loader plugin support, ensure that VisualeBIOS 7.17.03xx or higher is used.

Bios Integration build error/warning:

The target project with debugger modules added would show up build time warning/error message with error code if any conflicting configuration is present. Following details on the error/warning code may be helpful to understand more details:

#0x100001: OPTIMIZATION OFF preferable, OPTIMIZATION ON will provide limited functionality related to variables and source level debugging. While Optimization is turned ON, some variables/code will be removed off by the compiler and may not be visible during debugging. User may optionally turn OFF optimization only for the needed files using #pragma optimize("", off) if turning optimization OFF for entire project is not desirable.

#0x100002: DEBUG_MODE ON is necessary to enable the source level debugging. This is the basic token needed for enabling source level debugging on a target platform.

#0x100003: DEBUG_CODE ON is necessary to enable the target message redirection. This token is optional if user wish to see the redirected messages printed from target. User can continue source level debug even if token not set.

#0x100004: Library Mapping Not Found - DebugPortingLib. Refer Debugger Porting Guide Document for more information on porting. Refer AMI_Debug_for_UEFI_AptioV_Porting_Guide_NDA.pdf . This is the porting requirement needed from the BIOS project end for USB/Serial debugger to have the needed initialization.

Conflicting Tokens:

#0x300001 AMI_DEBUGGER_SUPPORT requires the AMIDEBUGGERPKG_SUPPORT SDL token to be 1

#0x300002 AMI_DEBUGGER_SUPPORT requires the USB_REDIRECTION_SUPPORT SDL token to be 0

#0x300003 AMI_DEBUGGER_SUPPORT - USB_3_DEBUG_SUPPORT requires the USB_DEBUG_TRANSPORT SDL token to be 0

#0x300004 AMI_DEBUGGER_SUPPORT - USB_3_DEBUG_SUPPORT requires the PeiDebugger_SUPPORT SDL token to be 1

#0x300005 AMI_DEBUGGER_SUPPORT - USB 2 DEBUGGER requires the USB_DEBUG_TRANSPORT SDL token to be 1

#0x400002 - AMIDEBUG_RX_SUPPORT - AMIDEBUG_RX_SUPPORT - REDIRECTION ONLY requires the USB_DEBUG_TRANSPORT SDL token to be 1

#0x400003 - AMIDEBUG_RX_SUPPORT - requires the Usb3Statuscode_Support SDL token to be 0

The above errors represent that there is some conflicting token set for the current debugger enabled configuration. User may turn the value of the conflicting token mentioned in the message with the recommended value to continue.

Multiple flavor conflict:

#0x200001 SERIAL_DEBUGGER_SUPPORT and USB_3_DEBUG_SUPPORT cannot be turned ON together

The above messages indicate that two flavors of debugger cannot be activated at the same time. User may choose to turn off one of the flavor to continue as recommended in the message displayed.

#0x400001 - SERIAL_DEBUGGER_SUPPORT and AMIDEBUG_RX_SUPPORT and cannot be turned ON together

#0x400005 - AMIDEBUG_RX_SUPPORT - DEBUG_CODE ON is necessary to enable the target message redirection.

The AMIDEBUG_RX_SUPPORT alone is enabled for message and checkpoint redirection. User may choose to turn on message if the user wish to see the redirected message in VEB console.

NOTE: Provided user has set all the needed configuration in target project, the build goes to completion without errors and a summary of the debugger configuration used would be shown at end of build.

Troubleshooting for initial Connection:

- **USB 3 Device Not Detected** : In case of USB 3.0 debugging, if device is not detecting in host machine please check the following check list:
  - Ensure the USB3 cable is connected in USB 3 (Blue) Port of the Host/Target machine
  - **Check USB 3.0 extensible Host Controller driver from the Manufacturer** has been installed in host machine**.** (User can download this driver from Manufacturer Website).
- **USB 3 Device Detected With Yellow Bang**: In case of USB 3.0 debugging, sometimes user may face problem the detection of AMI USB 3.0 cable in device manager or yellow bang will show in device manager like below screenshot.

In such case, update the Windows 7 to SP1 (if not already installed) and then install the security fix KB3033929. Also ensure that the latest AMI USB 3 Debug driver is installed from the USB3DriverPkg of Debugger Module.

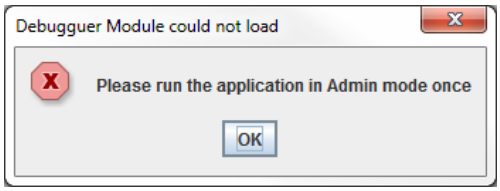
- **Serial debugging fails at Initial Handshake:** In case of**,** if after handshaking , debugger not proceeding further then make sure that serial cable/serial header connector is working properly for two way communication (i.e. reading and writing to serial port – User may use a terminal application to verify the same).
- **Connection Fail Even after following above steps:** Sometime user may face the trouble for initial connection even all debugger setup is properly. In such cases, user may try by **flashing the default ROM/BIN image** that comes with the platform and use the needed custom ROM image on top of the same. (Such cases were noticed in MSI Skylake Platform).

Debugging Related Issues

Troubleshooting for Start Debug Session:

When user start the debug session then sometime user may face the one of the below trouble:

- Received following Error Message



Launch Command prompt in Admin mode, run install /r from the root of the VeB

- **Authentication Error :** When user start debug session, user may see message box like below:

In such case, user has to **connect the AMI DebugRx device or USB debug key** to host machine.

- **Needs firmware update:** while debugging user may AMIDebugRx device shows error message "Needs firmware update". To solve this problem, user has to update AMIDebugRx device with latest firmware. To update AMIDebugRx firmware user can use AMIDebugRx user manual.
- **Console Window Missing:** Sometime user may face the trouble that the console window is not appearing after start debugging. In such cases, **delete the old VeB preferences**.
- **Communication issue with target:** When "Start Debugging" following message received "DebuggerCore: Detecting communication issues with target", even when the target is running.
    - Press "Stop Debugging" in the host
    - Turn off the target
    - Ensure that AMIDebugRx is properly connected between the host and the target system.
    - Ensure that driver for AMIDebugRx is installed.
    - Press "Start Debugging" in the host
    - Turn on target.

Troubleshooting for displaying source code:

While debugging, sometime user may face that source of code is not displaying. In such case, user can check the following checklist:

- Sometimes user may face problem for seeing source code even project built with appropriate token. In such cases user has to **install plugin again** means registering the Component in VeB.
- Host and target connects successfully but unable to see sources.
    - Ensure **Microsoft Visual C++ 2010/2013 Redistributable** is installed. (Refer installation instructions)
    - Ensure the respective BIOS module is **built with sources and in debug mode**
    - The **Source path** for the BIOS during **building and debugging must be same**. For example if source was built from C:\BIOS, while debugging the path must be same in the host computer.

Troubleshooting for hitting the breakpoint:

While debugging, sometime user may face the trouble, that breakpoint is not hitting. In such case, check the following cases:

- **Encountered in SEC/PEI phase:**

- In SEC\PEI phase there **cannot be set more than three breakpoints**, because the debugger uses Debug Registers to set Hardware breakpoint, since there are only 4 Debug registers, Debugger can set 3 user breakpoints and 1 breakpoint is used for Debugger stepping purposes.
- To avoid this scenario **make sure a maximum of 3 breakpoints are set** in SEC\PEI phase

- **No address associated with Source line:** When **OPTIMIZATION = ON**, Debugger may not be able to get the Address of some areas of the code accurately since it is optimized code, in this case there will be a message displayed in the AptioVDebugger Console which says "**DebuggerCore: WARNING: Unable to Set breakpoint - No address associated with the selected Source Line**". Disabling the optimization can fix this behavior if the line is a valid source.
- **Encountered across reboot:** When the source line belongs to a module that is not yet loaded in the current boot, in this case, when the debugger sets a Software breakpoint by writing to the memory address of the source line, this value will be overwritten when the module is loaded. To avoid this scenario it is better to first halt on the required module's entrypoint and then set the breakpoint
- **Encountered at breakpoint set on last loaded module:**
  - Check in the disassembly view that corresponding line is available and is not removed during preprocessing or optimization
  - A suggested way of using breakpoint is to halt at the module entry point and then set debug breakpoint

Troubleshooting for displaying views:

Views not refreshing: Views does not refresh and appears empty.

- Debug -> Stop Debugging.
- Debug -> Start Debugging (Debug session can be continued as the debugger automatically loads to the point where instruction pointer is currently pointing to.

Variables not listing: Sometime user may face the problem in listing variable in variable window. In such case, ensure that project build with optimization off. If not then rebuild the project with optimization off and try it.

General Feature Limitations and known Issues:

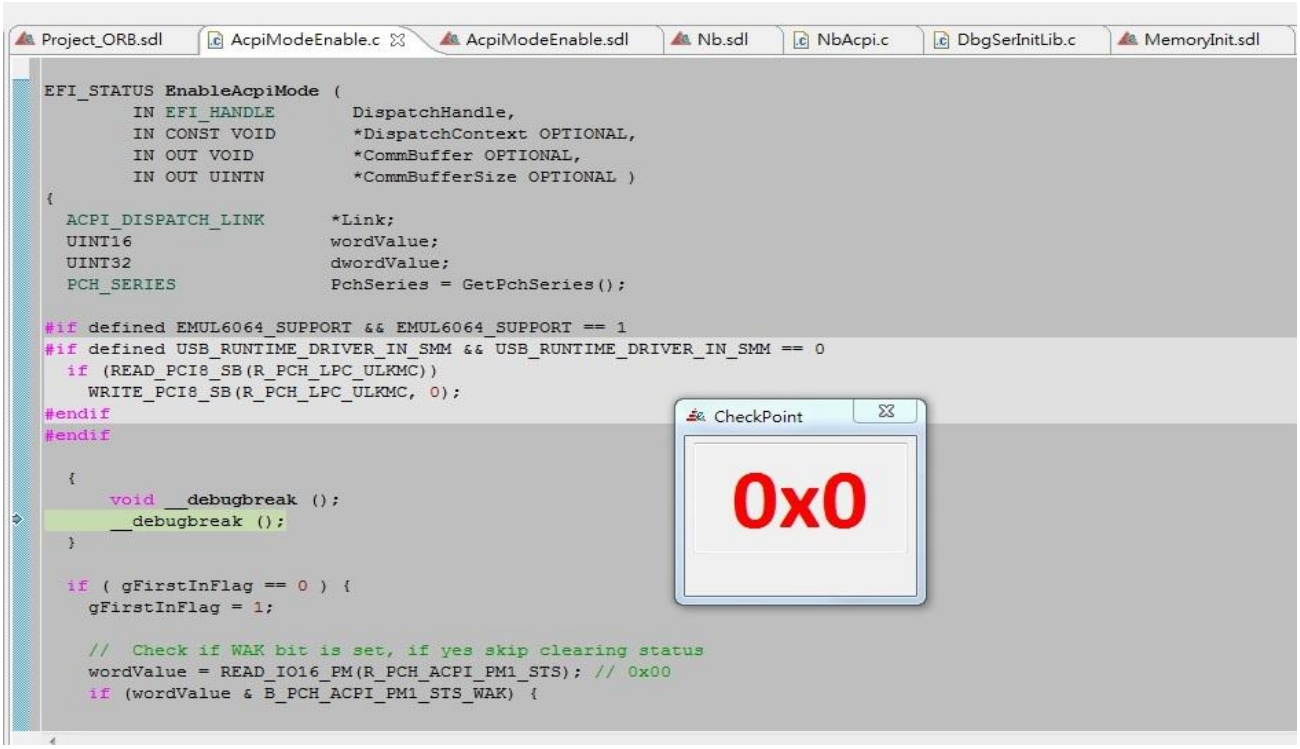
1) Debugger operations like setting breakpoint or performing stepping operations cannot be performed on the functions present within or which are linked to debugger module (Eg: LocatePpi).
2) Closing the project midway will lead to debugger unable to locate the needed source files.
3) Maximum of 4 Hardware Breakpoints allowed. Since Debugger uses 1 Hardware breakpoint for internal operations, user can use maximum 3 hardware breakpoints.
4) Variable window can hold maximum 1000 array variables.
5) JRE Environment- Debugger cannot launch if user installs Java 8 on top of Java 7. It is recommended to uninstall Java 7 completely and the install Java 8.
6) Code occurring before the debug agent is initialized cannot not be debugged: for example, early SEC, PEI to DXE, early SMM, etc.

AMI Debug for UEFI - Case Study 1

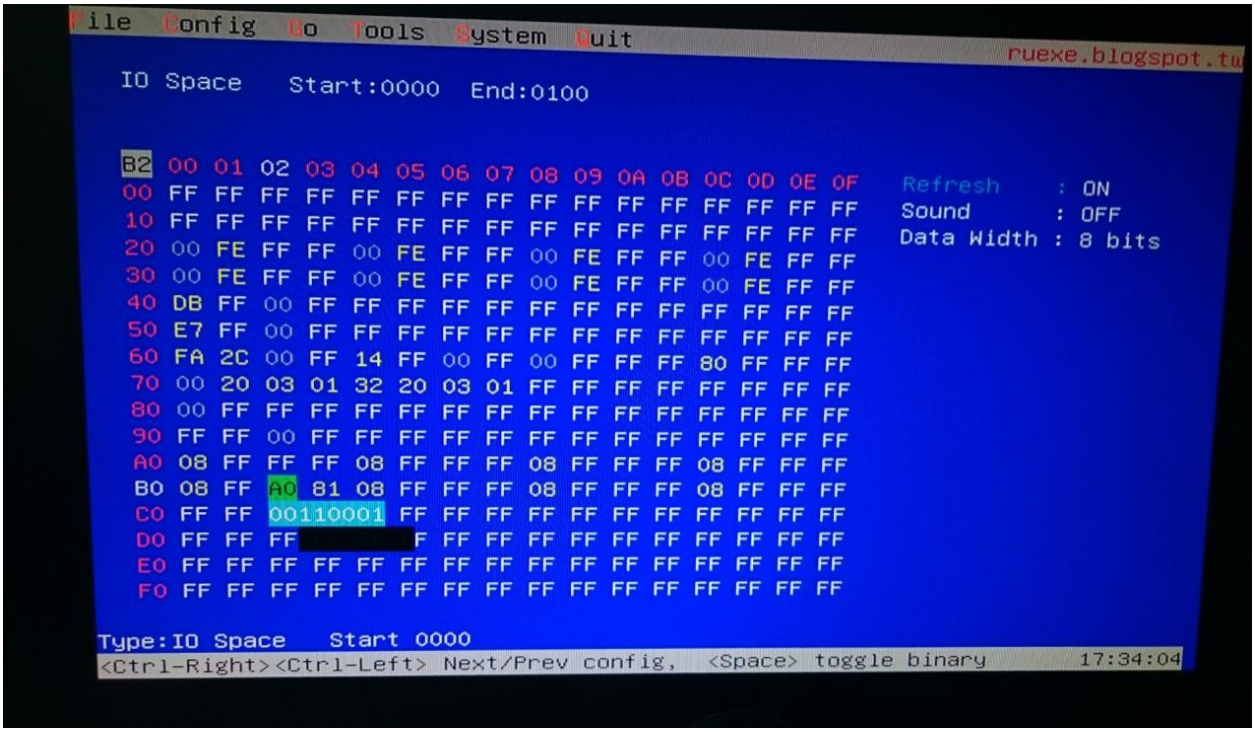
Category – Elink and CallBack function, EX: ACPI Enable SMI

Scenario – How to review **ACPI enable SMI code for ODM code change**

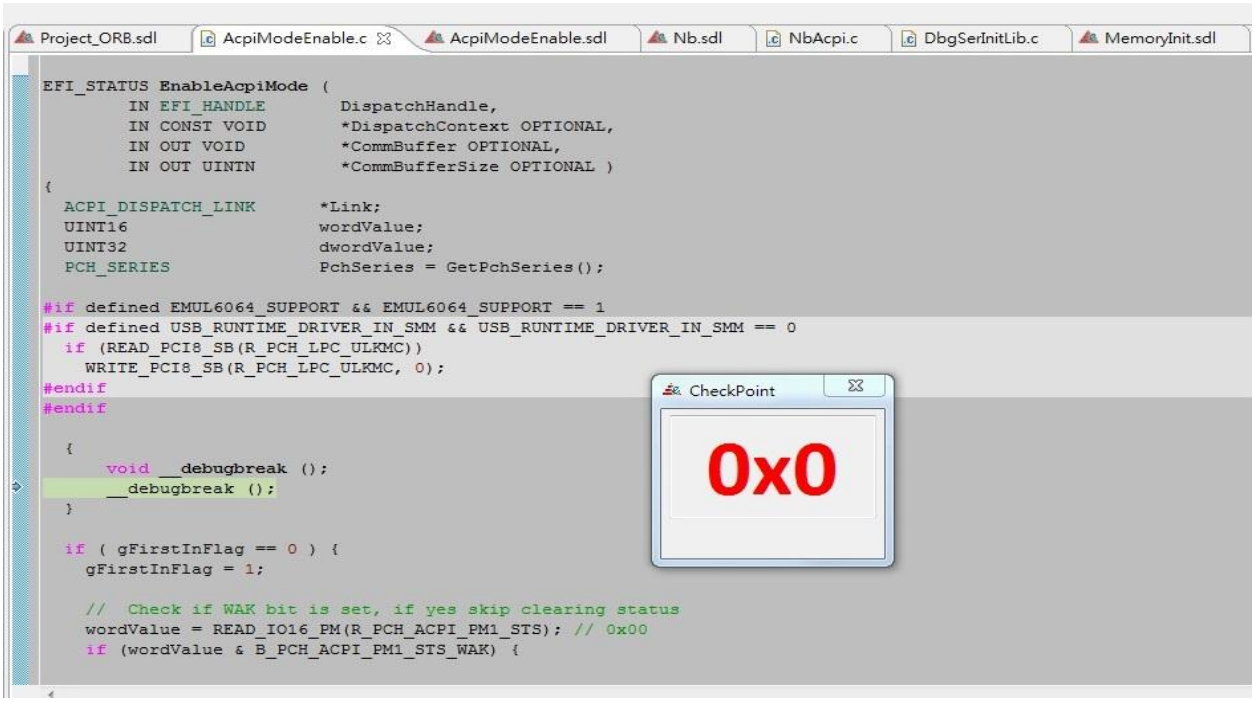
- Step 1 – Setting up source level debug
  - Setting breakpoint at EnableAcpiMode in AcpiModeEnable.c
  - Example of how to add breakpoint in code



- Step 2 – How to run SMI?
  - Option 1 – Connecting HDD with Windows installed → boot to OS → OS to call SMI
  - Option 2 – Sending SMI under Shell – sending SMI (0xa0) with ACPI enabled to IO space B2



- Step 3 – How to check the breakpoint?
  - VEB will show the breakpoint of EnableAcpiMode in AcpiModeEnable.c



➢ Step 3 – How to check the breakpoint?

- By using source level debug function to execute callback function
- F6 – (Step Over)
- CTRL+R – Run To Line
- F5 – (Step Into) run all the registered functions in loop

F6(Step Over)

➢ Step 4 – By using Elink to check callback function

- By using F5 (Step Into) to check callback function supported by Elink
- F7 – End the function and go back to original place

F5(Step Into)

```
    // Always enable B_PCH_ACPI_PM1_EN_PWRBTN
    SET_IO16_PM(R_PCH_ACPI_PM1_EN, BIT08);

    for (Link = gAcpiEnDispatchHead; Link; Link = Link->Link)
    {
        Link->Function(Link);
    }

    EnableAcpiModeCallbacks (DispatchHandle, DispatchContext);

    WRITE_IO8(0x80, SW_SMI_ACPI_ENABLE);

    return SMM_CHILD_DISPATCH_SUCCESS;
}
```

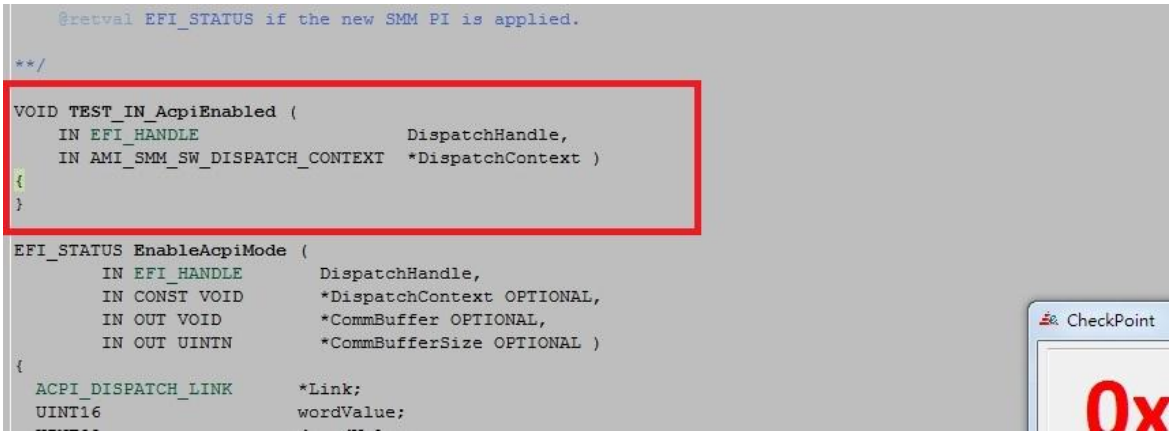
➢ Step 4 – Callback function linked by Elink

- By using F5 (Step Into) to check callback function supported by Elink
- F7 – End the function and go back to original place

```
VOID EnableAcpiModeCallbacks (
    IN EFI_HANDLE                   DispatchHandle,
    IN AMI_SMM_SW_DISPATCH_CONTEXT  *DispatchContext )
{
  UINT32                  Index;

  for (Index = 0; AcpiEnableCallbackList[Index] != NULL; Index++)
  {
        AcpiEnableCallbackList[Index](DispatchHandle, DispatchContext);
  }
}

/**
    This function enable ACPI mode by clearing all SMI and
    enabling SCI generation
    This routine is also called on a S3 resume for special ACPI
    programming.
    Status should not be cleared on S3 resume. Enables are
    already taken care of.
```



AMI Debug for UEFI - Case Study 2

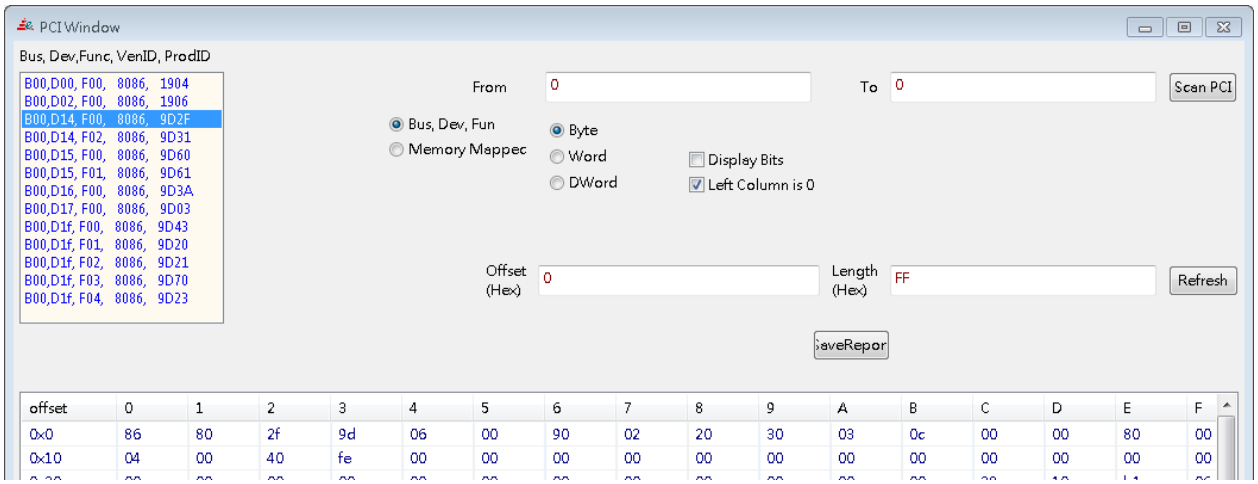
Category – Using PCI view to check PCI device resource

Scenario – WLAN card 8260 driver show yellow bang

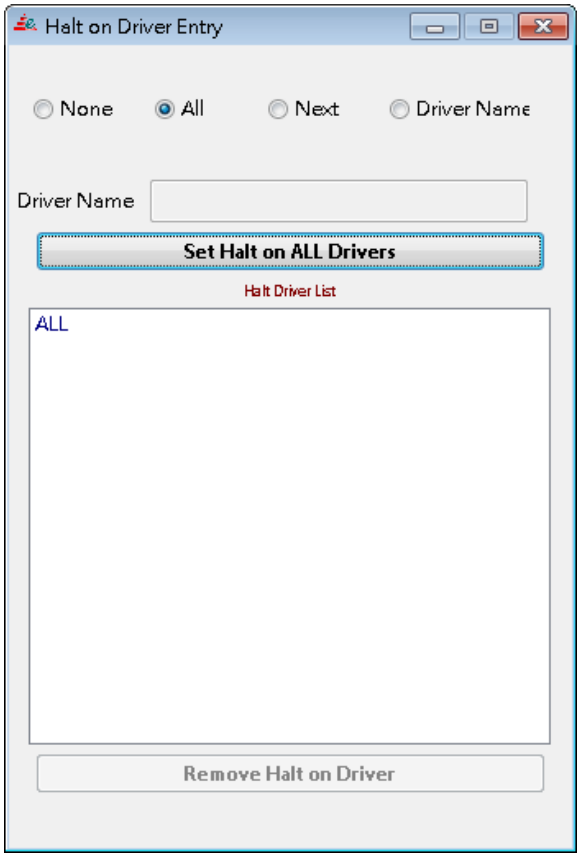
- Step 1 – Adding breakpoint
  - PCI resource is determined in function **"EnumerateBus" at "PciBus.c"** so adding breakpoint "__debugbreak ()" in this functi
  - on.

```
EFI_STATUS EnumerateBus(PCI_DEV_INFO *ParentBrg)
{
        EFI_STATUS              Status, Status1=EFI_NOT_FOUND;
        PCI_CFG_ADDR    pcidev;
        BOOLEAN                 mf,ari; //multi-function flag; ARI Flag
        UINT32                  id, cc; //Device ID Vendor ID reg
        UINT8                   ht; //Header type reg
        PCI_BRG_EXT             *ext, *parext;
        PCI_DEV_INFO    *dev, *func0, *func0ari;
        UINT8                   omb=0;
#if PCI_DEV_REVERSE_SCAN_ORDER
        UINT8            DevBuff;
#endif
        __debugbreak();
//---------------------------------
    PROGRESS_CODE(DXE_PCI_BUS_ENUM);
```

- Step 2 – PCI View
  - By using "PCI View" with "Step into" to check WLAN card resource
  - Found before WLAN root bridge (B0:D1C:F4) assign resource, there is another driver assign bus number to B0:D1C:F5 root bridge → caused function "EnumerateBus" can't find WLAN card.



- Step 3 – Halt at Module Entry
  - By using "PCI View" and "Halt at Module Entry" to chick which module assign bus number to B0:D1C:F5
  - Found NVME module assigned resource to the device for pre initialization but doesn't free temp BUS number.



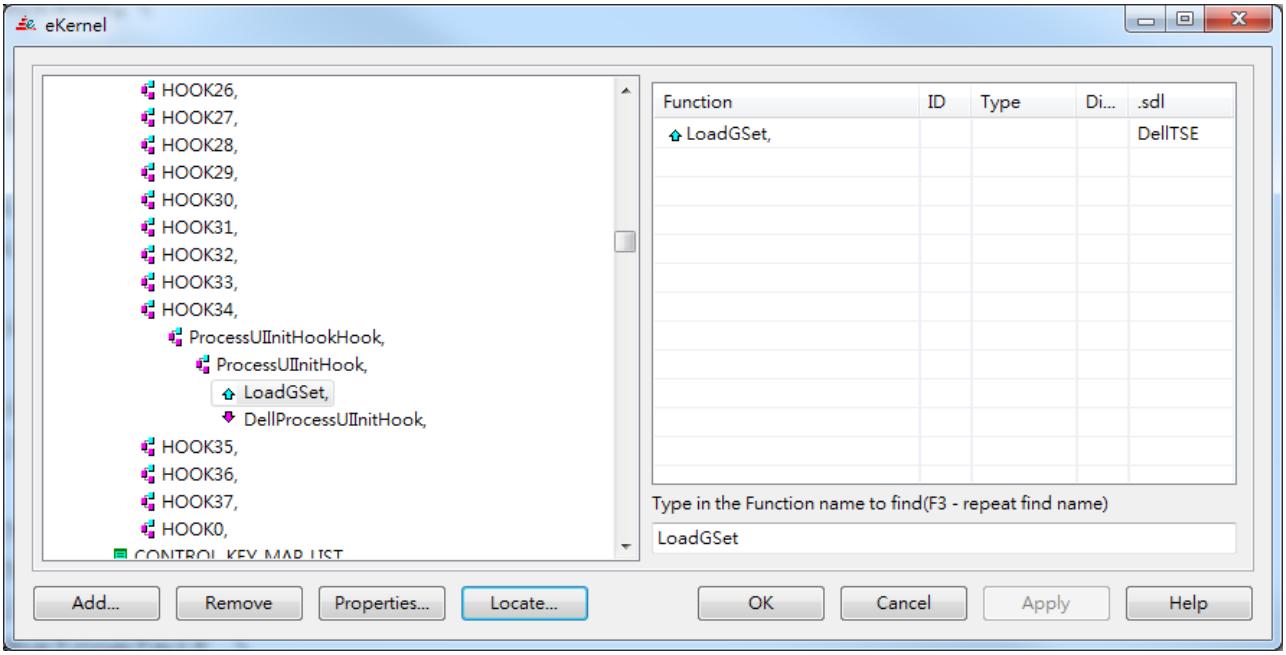
- **Root Cause-** NvmePci.c doesn't free temp BUS number to make Bus number conflict in Dxe BUS driver. It make Wlan card lose in BUS driver.
- **Solution -** Since project doesn't need to support NVME function, just disable this function.

AMI Debug for UEFI - Case Study 3

Category – Looking for Hook/Protocol trigger point

Scenario – Load BIOS default will hang in BIOS when flash BIOS and close Mfgmode(70%)

- In general case, the BIOS setup(GSET) will invoke by our ELINK, and the issue never happened by this way, so we guess it should be some private code locate the GSET and jump in directly cause it.



- Problem – Since GSET is private code, so it's almost impossible to know the behavior inside.
- Method – Redirect GSET entry point to our code then return back to check caller.

```
875 EFI_STATUS
876 NewDellSetupEntry (
877     UINT8 EntryType
878 )
879 {
880         TRACE ((-1,"[YCHsu]NewDellSetupEntry\n"));
881         __debugbreak();
882         return 0;
883 }
884
885 VOID YCHsuDebugCallback (
886     IN EFI_EVENT        Event,
887     IN VOID             *Context )
888 {
889         EFI_STATUS Status;
890         EFI_SETUP_PROTOCOL    *DellSetup = NULL;
891
892         Status = pBS->LocateProtocol (&gDellSetupProtocolGuid, NULL, &DellSetup);
893             if (EFI_ERROR(Status)) return;
894
895         OldDellSetupEntry = DellSetup->Entry;
896         DellSetup->Entry = NewDellSetupEntry;
897 }
```

- After return to caller, the Veb debugger screen shows
  "e13b5a8fa5d769e019f526ffa5b0e1afa5137f6c.pdb",
  since it's GUID of BIOS, so just search it in our code, then we know it is "SpecialBootStubDxe.efi".

AMI Debug for UEFI - Case Study 4

Category – Scripting support feature

Scenario – Loading/Executing Sharkbay ITP scripts

- Extract available Sharkbay scripts in script folder of VeB.

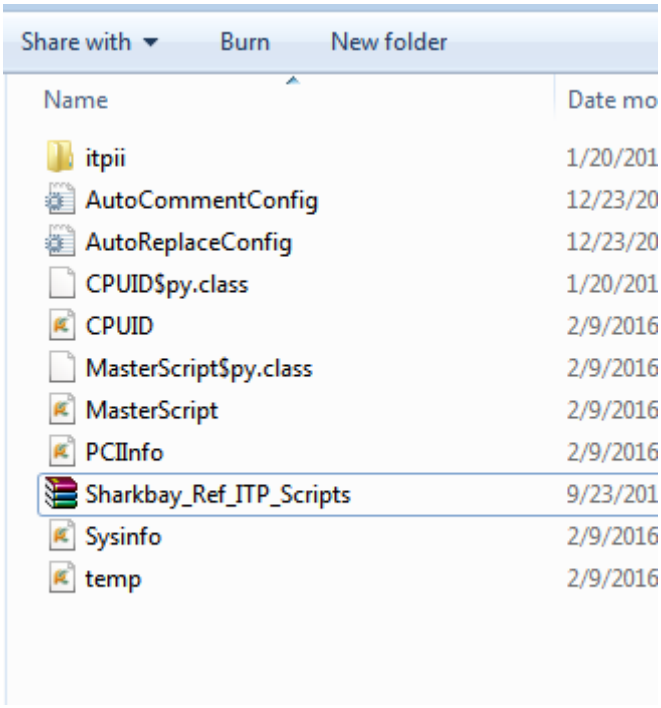


Fig: Sharkbay ITP reference script

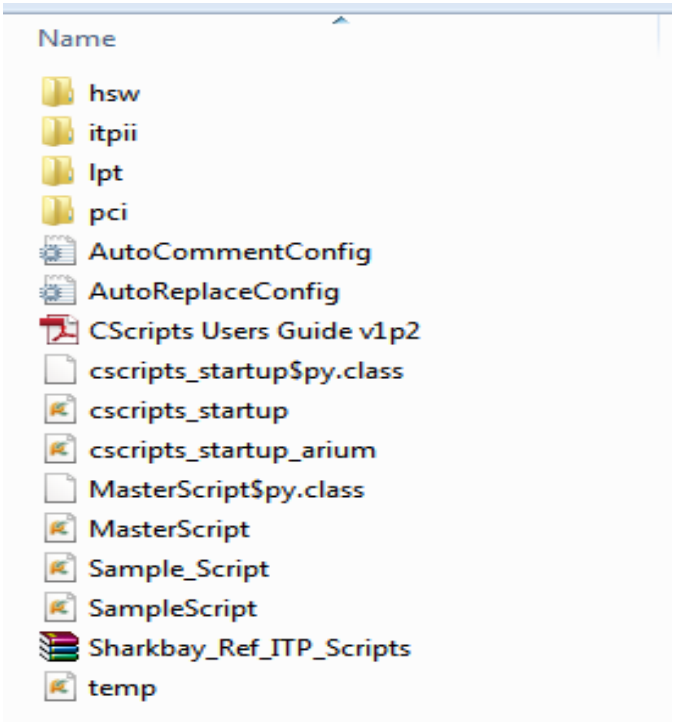
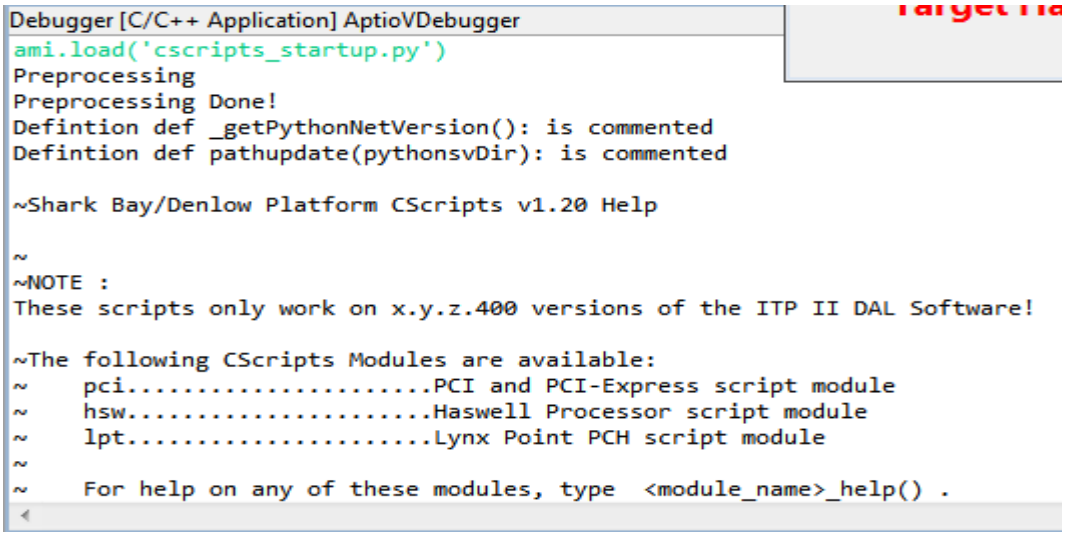
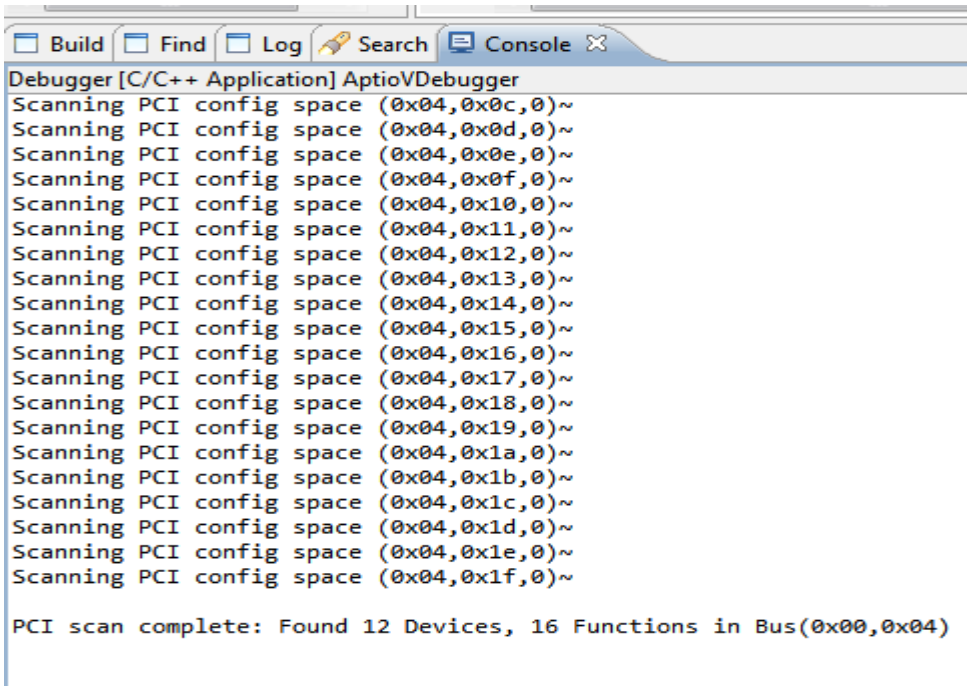


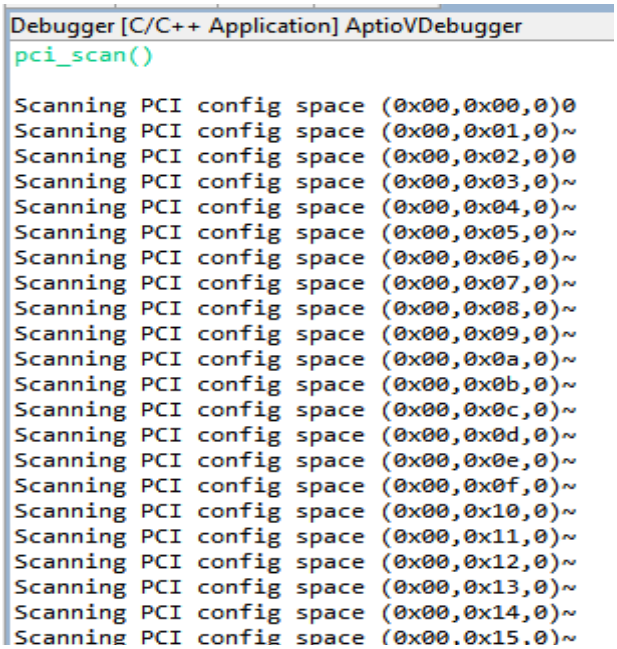
Fig: Sharkbay ITP reference script extracted to VEB Scripts folder

- Load any user defined script present in scripts of VeB folder using ami.load() command
- Use AptioVDebugger console to execute commands. The following slides show a sample execution of the 2 functions/commands defined in the Sharkbay ITP scripts.



- After loading the file, user can execute the commands present in that file
- Execute pci_scan() command after loading scan.py file to get available pci devices count.





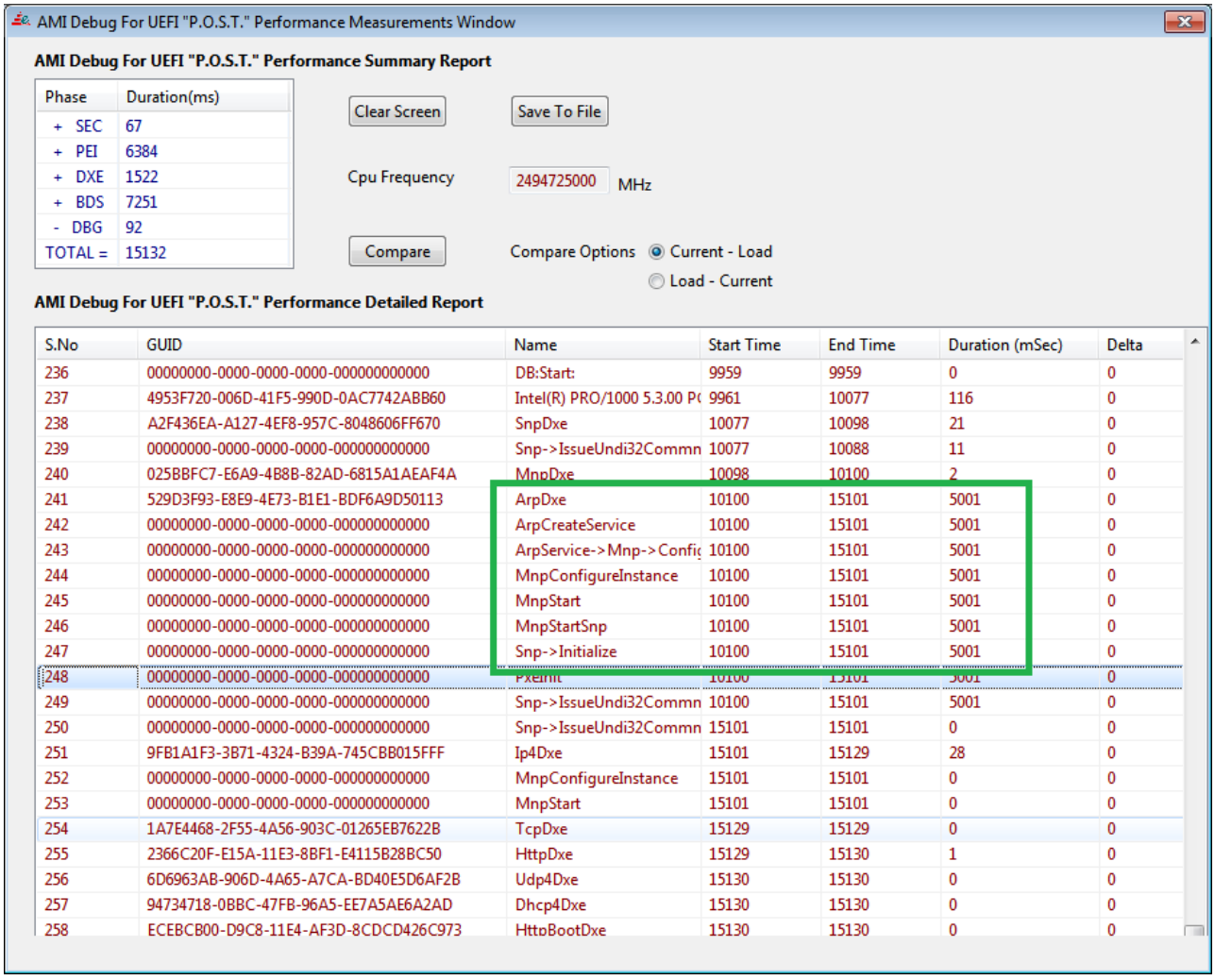
- To get MCH PCI devices enable/disable status, load pcie file present hsw folder
- Execute mch_dev check(), observe its output in AptioVDebugger Console

```
Debugger [C/C++ Application] AptioVDebugger
ami.load('hsw\pcie.py')
User Script loaded succesffuly
mch_dev_check()
|
0x00000011
~+--+--+-+--------------------------------+-----------+
~|B |D |F|                                |           |
~|u |e |n|                                |           |
~|s |v |c|Description                     |Status     |
~+--+--+-+--------------------------------+-----------+
~|00|00|0|Host Bridge/DRAM Controller     |Enabled    |
~|00|01|0|1st Host-PCI Express Bridge     |Disabled   |
~|00|01|1|2nd Host-PCI Express Bridge     |Disabled   |
~|00|01|2|3rd Host-PCI Express Bridge     |Disabled   |
~|00|02|0|Internal Graphics Device        |Enabled    |
~|00|03|0|Audio Controller                |Disabled   |
~|00|06|0|4th Host-PCI Express Bridge     |Disabled   |
~+--+--+-+--------------------------------+-----------+
```

AMI Debug for UEFI - Case Study 5

- **Category** – Performance Measurement
- **Scenario** – ArpDxe driver of UefiNetworkStack Module took 5 seconds to load when network cable unplugged. Performance Measurement feature is used to time profile and track down the function causing the delay in execution which was part of the NIC driver supplied by vendor

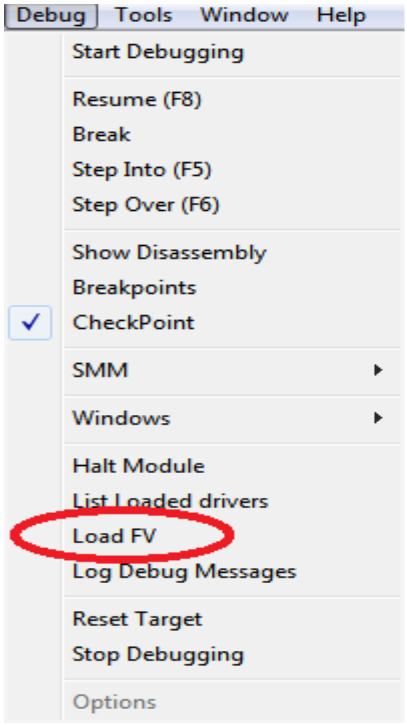
- ArpDxe Driver start took 5 seconds to execute when network cable was unplugged.
- Performance measurement was taken initially to understand how much load time each module took and ArpDxe consumes 5 seconds time
- PERF_START and PERF_END macros were added in various functions under the ArpDriverBindingStart() to track down which function caused the delay
- After re-exection it was tracked down that *ArpDriverBindingStart() -> ArpCreateService() -> [ArpService->Mnp->Configure()] -> MnpConfigureInstance() -> MnpStart() -> MnpStartSnp() -> [Snp->Initialize()] -> PxeInit() ->(*Snp->IssueUndi32Command)* *()* caused that delay which was part of UNDI driver supplied by NIC card vendor
- This saves the time in time profiling operations



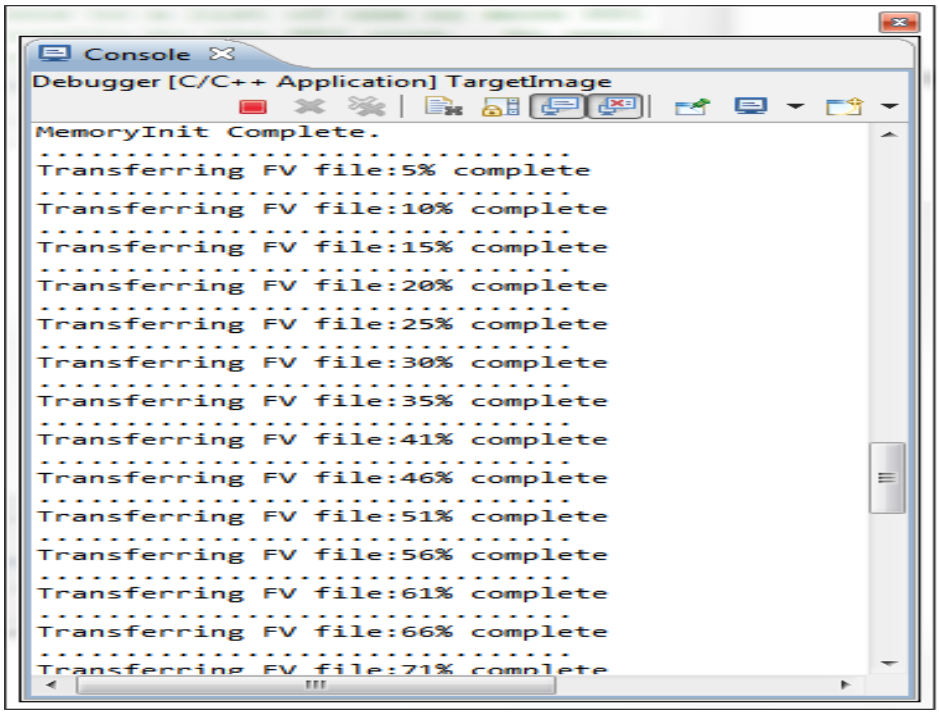
AMI Debug for UEFI - Case Study 6

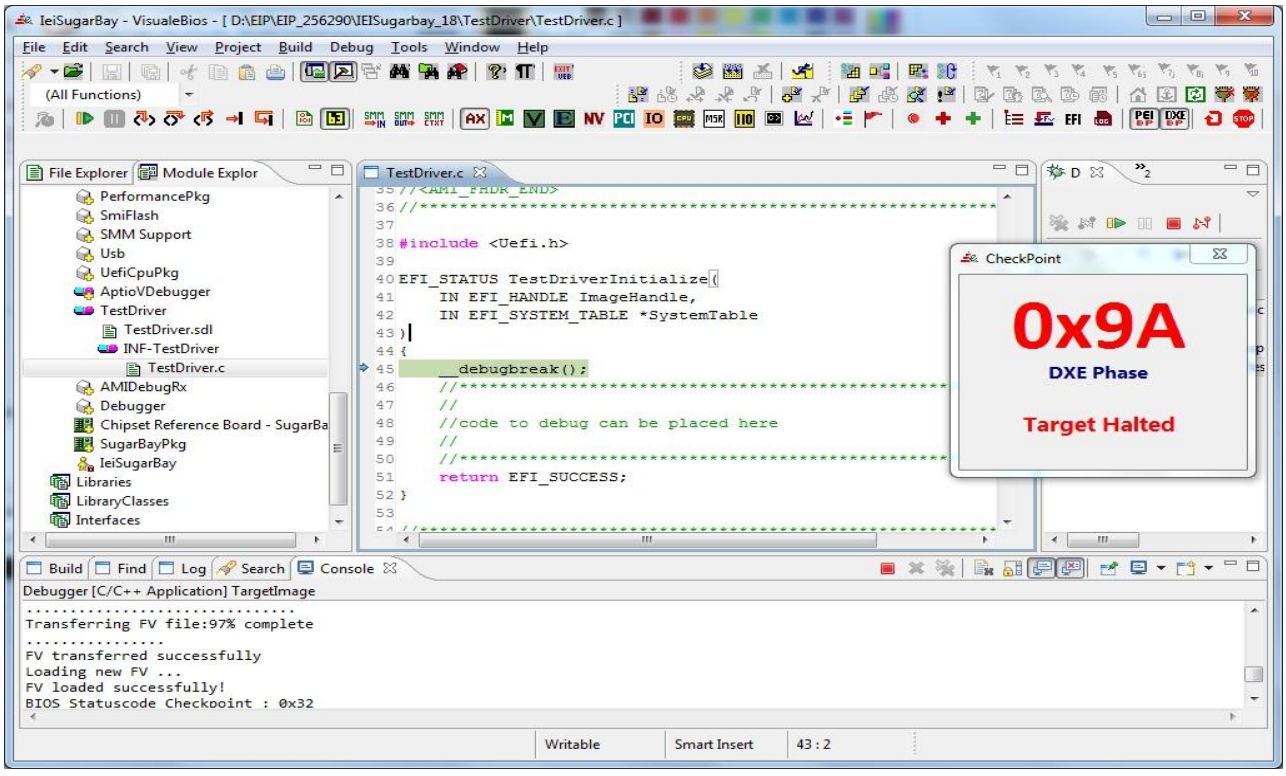
- **Category** – LoadFV
- **Scenario** – Debugging a test module by Loading Firmware volume without actually flashing the BIOS image
- There was a TestDriver that did not fit into the Flash size limits in debug mode. So the LoadFV feature was used to debug the module without actually flashing the module into the BIOS chip. This is applicable in scenario when a new BIOS module needs to be debugged without having the BIOS image flashed. This can be used for debugging the large modules that may not fit into current flash size limits in debug mode
- Have the BIOS project with the new module added. Place an __debugbreak() in the module entry point (*TestDriver.c -> TestDriverInitialize()*). built in the host side.

- With debugger connected in before memory init stage use the LoadFV option in debugger menu to choose the newly built ROM with the module to be debugged.



- Resume the debugger connection using F8.
- After memory initialization completes, the Load FV operation begins and the image gets temporarily loaded into Target without actual flash
- Wait till the module to be debugged gets loaded. When the an __debugbreak() gets hit, the debugger opens up the file corresponding to the newly loaded module and debugging can be performed on the same.


Console
Debugger [C/C++ Application] TargetImage
MemoryInit Complete.
Transferring FV file:5% complete
Transferring FV file:10% complete
Transferring FV file:15% complete
Transferring FV file:20% complete
Transferring FV file:25% complete
Transferring FV file:30% complete
Transferring FV file:35% complete
Transferring FV file:41% complete
Transferring FV file:46% complete
Transferring FV file:51% complete
Transferring FV file:56% complete
Transferring FV file:61% complete
Transferring FV file:66% complete

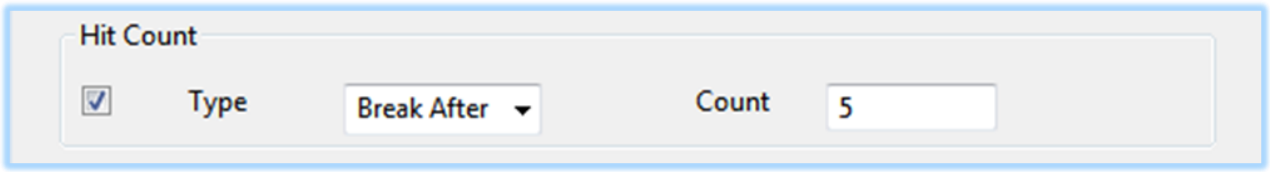

IeiSugarBay - VisualeBios - [D:\EIP\EIP_256290\IEISugarbay_18\TestDriver\TestDriver.c]
File Edit Search View Project Build Debug Tools Window Help
(All Functions)
File Explorer
Module Explor
PerformancePkg
SmiFlash
SMM Support
Usb
UefiCpuPkg
AptioVDebugger
TestDriver
TestDriver.sdl
INF-TestDriver
TestDriver.c
AMIDebugRx
Debugger
Chipset Reference Board - SugarBa
SugarBayPkg
IeiSugarBay
Libraries
LibraryClasses
Interfaces
TestDriver.c
#include <Uefi.h>
EFI_STATUS TestDriverInitialize(
IN EFI_HANDLE ImageHandle,
IN EFI_SYSTEM_TABLE *SystemTable
)
{
__debugbreak();
//code to debug can be placed here
return EFI_SUCCESS;
}
CheckPoint
0x9A
DXE Phase
Target Halted
Build
Find
Log
Search
Console
Debugger [C/C++ Application] TargetImage
Transferring FV file:97% complete
FV transferred successfully
Loading new FV ...
FV loaded successfully!
BIOS Statuscode Checkpoint : 0x32
Writable
Smart Insert
43 : 2

AMI Debug for UEFI - Case Study 7

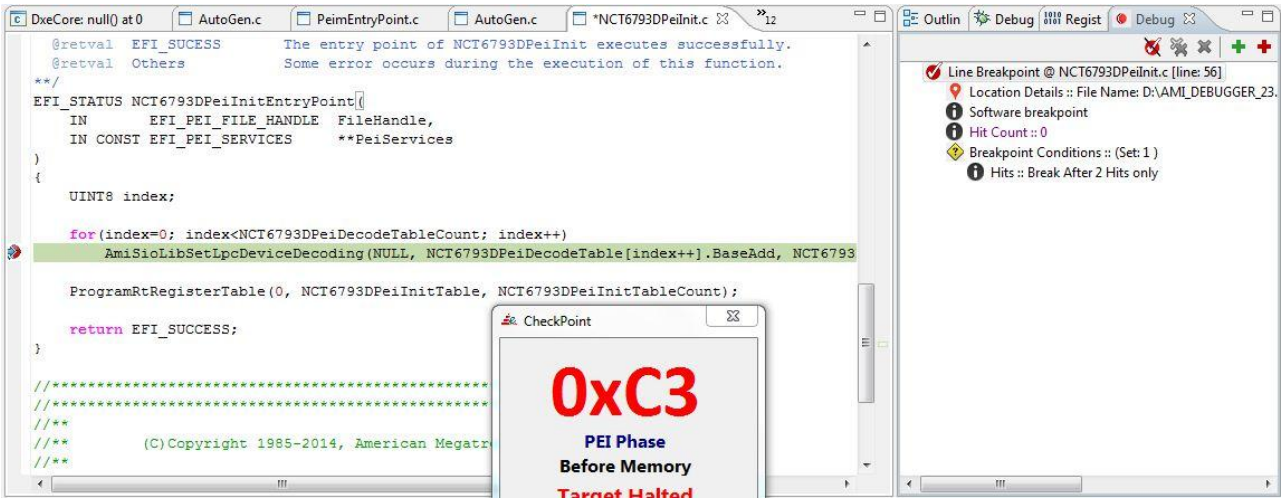
Category – Using Hit count in conditional breakpoint.

Scenario – Case of buffer overflow in modules after executing N number of loops

- There was a case in NCT6793DPeiInit.c sample code was crashing due to buffer overflow. But this happened only after executing the function for 5 times.
- When the problem happen in a function/loop that happens after multiple execution, if breakpoints are set, the breakpoints gets invoked in each loop making the debugging operation tedious.
- After it came to the notice that only after the 5th execution the crash happens, a Hit Count condition was added to the breakpoint set in NCT6793DPeiInitEntryPoint() so that first 4 hits would get skipped and directly the debugger halts in the 5th execution. This helped in making the debugging easier to identify that in 5th execution the incorrect usage of index++ overflowed the available buffer and accessed out of bounds.
- This feature is specially useful in functions which shows incorrect behavior after dozens of execution, so that debugger can continue execution till the specified time and halt after that so that debugging operation is smoother and less time consuming.
- **Method** – Put a Breakpoint in that piece of code which gets iterated while setting breakpoint, add Hit Count condition option with count given and type should be 'Break After' and add it to the breakpoint while setting.



Note: Here as for example the array **NCT6793DPeiDecodeTable** has its accessed index always one higher thus eventually it'll get out of its boundary after certain count **NCT6793DPeiDecodeTableCount** and will take garbage value in consideration hence getting execution corrupted.



AMI Debug for UEFI - Case Study 8

Category – Debug a UEFI application while launched from shell.

Scenario – when any efi application is needed to be debugged after launching it from shell application.

Problem – any EFI application is not working properly.

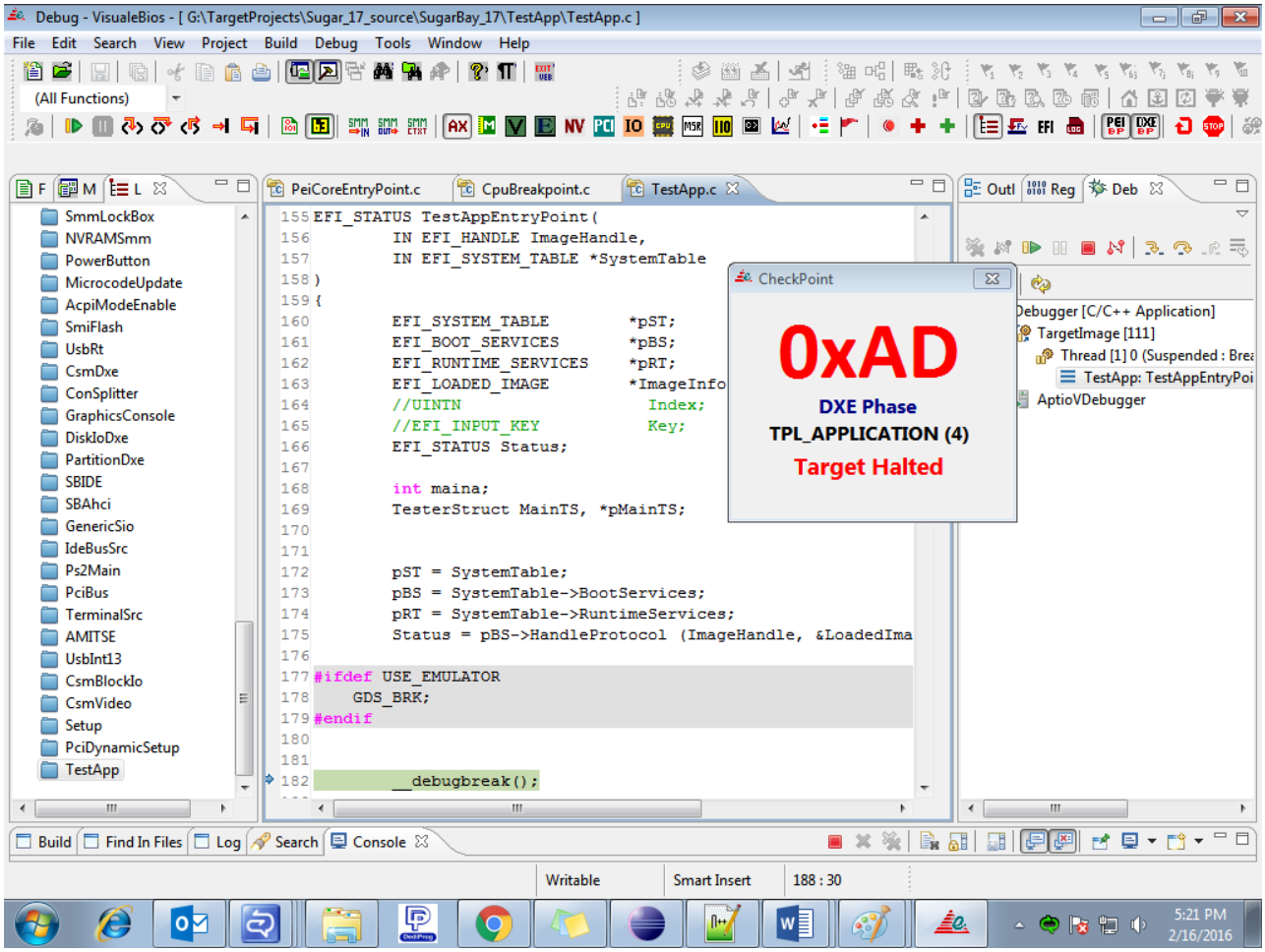
Method –

- Step1: generate EFI executable after including __debugbreak() statement in that EFI application's source entry point (*TestApp.c -> TestAppEntryPoint()*)
- Step2: With debugger connected, launch the test application from the Shell

  Shell > TestApp.efi
- Step3: When EFI gets launched then debugger itself gets break on the breakpoint statement __debugbreak() . The loaded source is opened in the editor pane.
- Step 4: All debugging windows like Variable, Memory and register window and and operations like step in/over can be used to trace the problem in the test application.

Note: Here as for example a sample app TestApp.efi is needed to be debugged for root causing an application error

When application is launched from shell application it breaks on __debugbreak(). Further stepping operations can be used to trace down the application.



AMI Debug for UEFI - Case Study 9

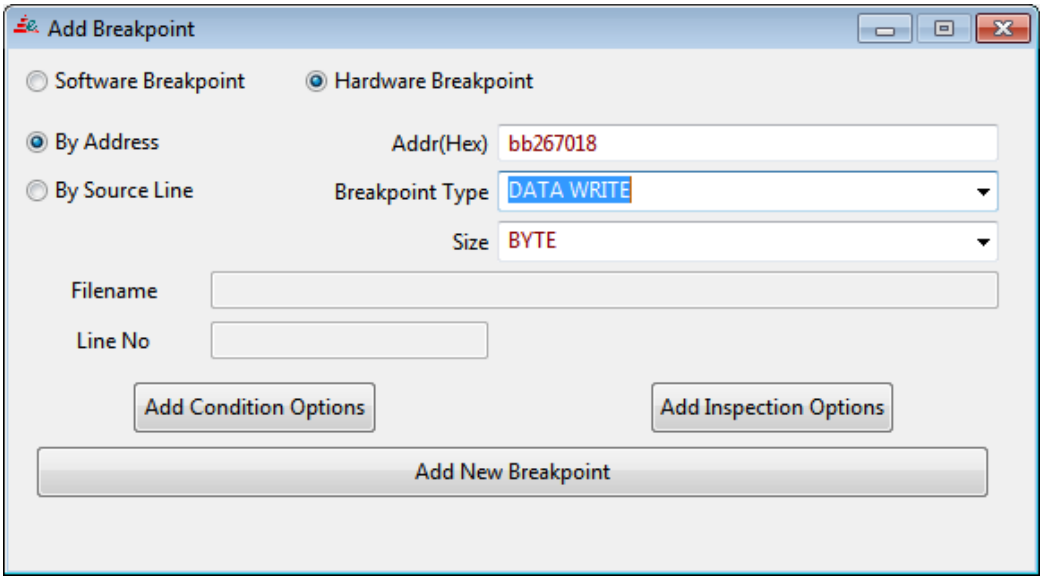
Category – Data Breakpoint

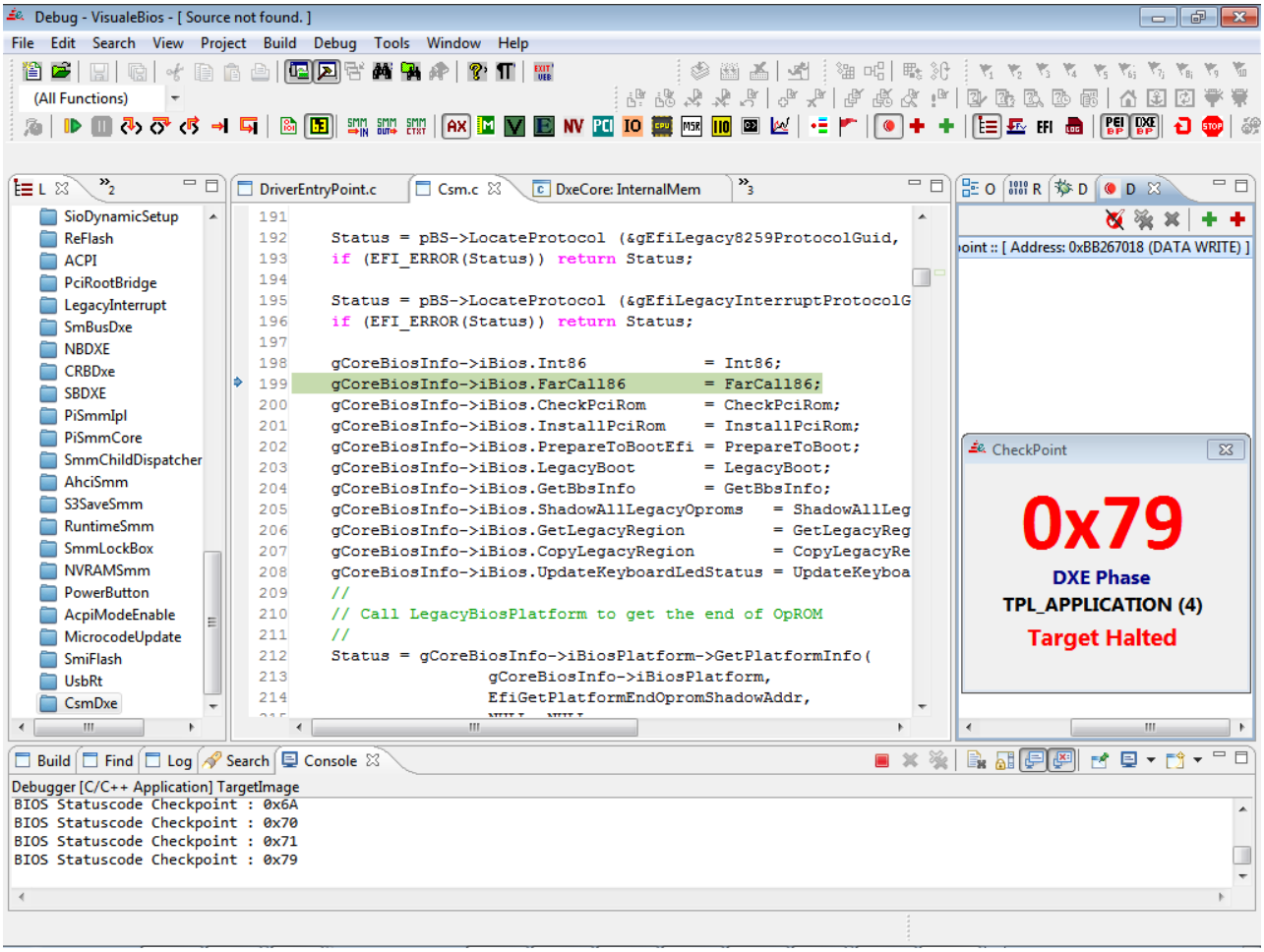
Scenario – Tracking the function which modifies a global data/memory location

Problem – Structure variable pointed by gCoreBIOSInfo was modified from other parts of the module

Method –

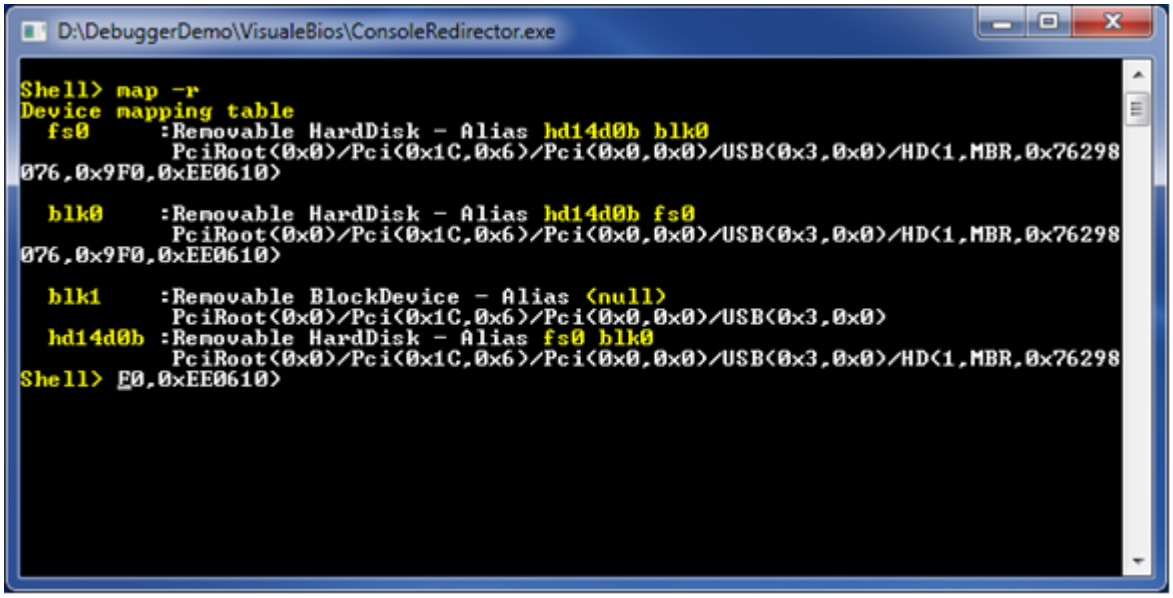
- Step1: Identify the pointer value of gCoreBIOSInfo from the expression window
- Step2: Create a DATA WRITE breakpoint using custom breakpoint window
- Step3: When the code modifying the data content is executed, the debugger halts and the corresponding source view get loaded.
- Step 4:This can be used to track the functions that modify the commonly shared global data/ Dynamically allocated address.


Add Breakpoint
Software Breakpoint
Hardware Breakpoint
By Address
Addr(Hex)
bb267018
By Source Line
Breakpoint Type
DATA WRITE
Size
BYTE
Filename
Line No
Add Condition Options
Add Inspection Options
Add New Breakpoint


Debug - VisualeBios - [Source not found.]
File Edit Search View Project Build Debug Tools Window Help
(All Functions)
DriverEntryPoint.c
Csm.c
DxeCore: InternalMem
SioDynamicSetup
ReFlash
ACPI
PciRootBridge
LegacyInterrupt
SmBusDxe
NBDXE
CRBDxe
SBDXE
PiSmmIpl
PiSmmCore
SmmChildDispatcher
AhciSmm
S3SaveSmm
RuntimeSmm
SmmLockBox
NVRAMSmm
PowerButton
AcpiModeEnable
MicrocodeUpdate
SmiFlash
UsbRt
CsmDxe
191
192 Status = pBS->LocateProtocol (&gEfiLegacy8259ProtocolGuid,
193 if (EFI_ERROR(Status)) return Status;
194
195 Status = pBS->LocateProtocol (&gEfiLegacyInterruptProtocolG
196 if (EFI_ERROR(Status)) return Status;
197
198 gCoreBiosInfo->iBios.Int86 = Int86;
199 gCoreBiosInfo->iBios.FarCall86 = FarCall86;
200 gCoreBiosInfo->iBios.CheckPciRom = CheckPciRom;
201 gCoreBiosInfo->iBios.InstallPciRom = InstallPciRom;
202 gCoreBiosInfo->iBios.PrepareToBootEfi = PrepareToBoot;
203 gCoreBiosInfo->iBios.LegacyBoot = LegacyBoot;
204 gCoreBiosInfo->iBios.GetBbsInfo = GetBbsInfo;
205 gCoreBiosInfo->iBios.ShadowAllLegacyOproms = ShadowAllLeg
206 gCoreBiosInfo->iBios.GetLegacyRegion = GetLegacyReg
207 gCoreBiosInfo->iBios.CopyLegacyRegion = CopyLegacyRe
208 gCoreBiosInfo->iBios.UpdateKeyboardLedStatus = UpdateKeyboa
209 //
210 // Call LegacyBiosPlatform to get the end of OpROM
211 //
212 Status = gCoreBiosInfo->iBiosPlatform->GetPlatformInfo(
213 gCoreBiosInfo->iBiosPlatform,
214 EfiGetPlatformEndOpromShadowAddr,
oint :: [Address: 0xBB267018 (DATA WRITE)]
CheckPoint
0x79
DXE Phase
TPL_APPLICATION (4)
Target Halted
Build Find Log Search Console
Debugger [C/C++ Application] TargetImage
BIOS Statuscode Checkpoint : 0x6A
BIOS Statuscode Checkpoint : 0x70
BIOS Statuscode Checkpoint : 0x71
BIOS Statuscode Checkpoint : 0x79

AMI Debug for UEFI - Case Study 10

- **Category** – Console Redirection
- **Scenario** – During the debugging operation, client and target may need additional KVM resources for input/output. Console redirection helps in avoiding the need of extra hardware needed to control the target as long as the operation is in text mode.
- It is possible to control the target without any extra hardware while in text mode/BIOS setup
- This helps in testing built in BIOS modules/BIOS Setup operation and Shell based test applications with minimal hardwares like mouse/Keyboard/Monitor.



Appendix

Working with IO.ini file

RULES

1) Every Section must have an EndSection

   Eg: [CPU] and [/CPU]

2) Every SubSection must have an EndSubSection

   Eg: [IO] and [/IO]

   Eg: [IOInvalid] and [/IOInvalid]

3) Valid AddressRange of IO must be in the following syntax under [IO] section;

   Eg: StartRange<Space>-<Space>EndRange

   Eg: 80 - A1

4) if no Range is specified DEFAULT range will be 00 – FFFF

5) Invalid AddressRange of IO must be in the following syntax under [IOInvalid] Section

   Eg: StartRange<Space>-<Space>EndRange

   Eg: 1800 - 1900

6) For READONLY AddressRange, it must be in the following syntax

   Eg: 'R'<Space>StartRange<Space>-<Space>EndRange

   Eg: R 70 - 77

NOTE:

1. IF IO range clashes with Valid and Invalid, Invalid will get precedence

2. Comments must be preceded with ';'

3. Please do not enter RESERVED Ranges

Example INI format

```
[CPU]
  ;Invalid Range
    [IOInvalid]
```

```
                1800 - 1900
    [/IOInvalid]
  ;Valid Range

        [IO]

          1901 - FFFF
          R 7F - 8F

        [/IO]

  [/CPU]
```

Reference

- AMI Visual eBIOS Eclipse User Manual
- UEFI specifications & related documents can be downloaded at http://www.TianoCore.org
- Additional documentation may be available at AMI's website (http://www.ami.com) or at the AMI Customer Portal (https://cp.ami.com).
- AMI Debug Rx
    - AMI_Debug_Rx_Quick_Start_PUB
    - AMI_Debug_Rx_User_Manual_PUB
    - AMI_Debug_for_UEFI_AptioV_Porting_Guide_NDA.pdf