UEFI Debugging using SourcePoint® on Intel® Platforms

Overview

The Unified Extensible Firmware Interface (UEFI), commonly known as the UEFI Framework, is a well-established firmware specification standard that defines a set of software interfaces and replaces the legacy BIOS found on traditional PC computers. This framework provides the kind of modularity, flexibility, and extensibility that were formerly unavailable with traditional BIOS. With UEFI, BIOS developers can now write all their code in 'C', rather than assembly language. See the UEFI website at http://www.uefi.org/ for more information on the UEFI Framework.

Along with this firmware architecture and the 'C' code that implements it comes the need for source-level debugging. ASSET InterTech's debugger, SourcePoint® for Intel® and Arm® processors, offers native debug support for UEFI Framework platforms. Users can set breakpoints, single step, view variables, see the call stack, and access all of the feature-rich functionality SourcePoint normally provides. SourcePoint also provides several types of trace display on Intel-based systems. This includes source-level debugging during the SEC, PEI, DXE, BDS, and OS Boot phases of UEFI. Below is a set of instructions for setting up SourcePoint to debug the UEFI Framework. Throughout this document we will not only provide information about the macros that assist in UEFI debugging, but will also provide information about built-in commands within SourcePoint that assist the user in debugging.

Brief UEFI Overview

There are three major areas of code in a UEFI build. These are PEI, Framework, and EFI (DXE). One way of visualizing this topology is shown in Figure 1 below.

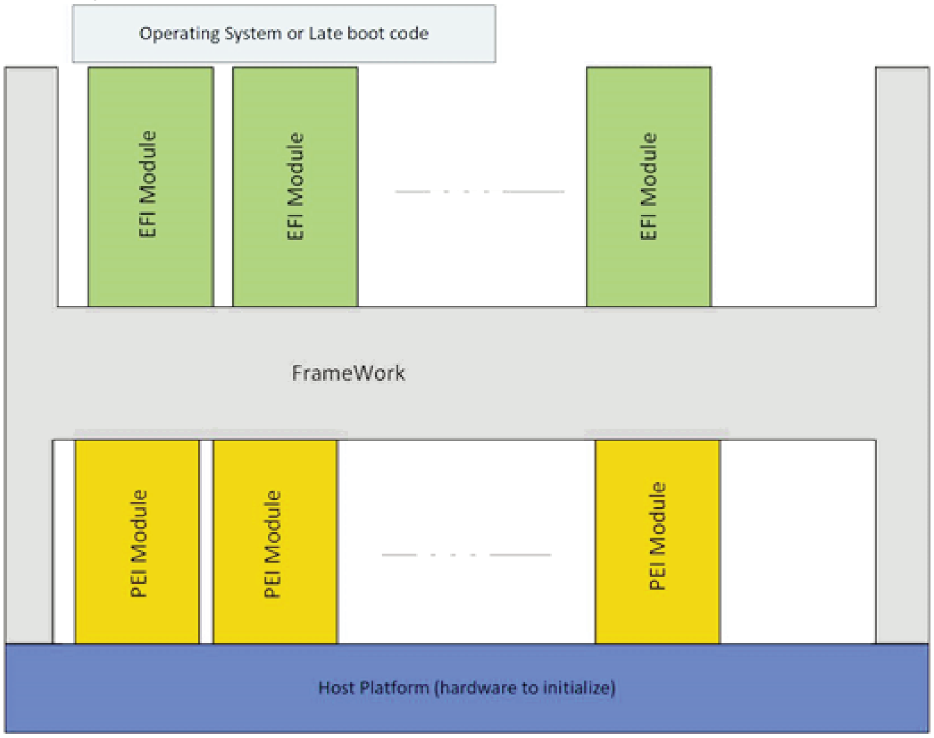

Figure 1: EFI Structure

After hardware reset, the SEC module executes. It starts with code written in assembly. This code runs in a special hardware mode where real-mode addresses are extended to address the area at the top of 4 Gbytes of memory. SourcePoint deals with this automatically, but instruction trace is not fully decoded. After only around 30 instructions and usually only three jumps, the processor is switched to protected mode and most debug features are available. Because DRAM is not available until after MRC completes, Last Branch Record (LBR) instruction trace must be used up to that point.

After early SEC code, the PEI scheduler is launched and PEI modules are executed. These modules are all written in 'C' and use special memory for the stack.

On completion of the PEI phase the DXE phase is launched, which supports all of the selected EFI modules. At some point one of these EFI modules will cause an OS boot to begin. For more detail on this architecture refer to web materials including UEFI.org and Tianocore.org.

Project Initialization

Upon starting SourcePoint, a project file must be created. For the purposes of this Application Note, we will use the default target configuration file for the Skylake Platform. (Skylake.tc)

EFI Macros

Note: The macros described below are installed in the Macro\EFI sub-folder of the SourcePoint install path. Several of the EFI macro files contain directory paths to other macro files. If you move the macro files or change the current working directory in SourcePoint (via the 'cwd' command), you will need to update the macro files with the new locations.

EFI.mac

After installing SourcePoint, run the EFI.mac macro file located in the Macro\EFI directory. This creates ten custom toolbar buttons and associates each with a corresponding EFI macro description as shown below:

Figure 2: EFI.mac Toolbar Buttons

Each macro action will be discussed below to help user can understand the action with respect to the EFI.

- The **LoadCurrent** button attempts to loads source and symbol debug information for the currently executing code.
- The **PEIMs** (Pre-UEFI Initialization Modules) button loads the symbol files for the PEI modules found in target memory.
- The **DXEs** (Driver Execution Environments) button loads the symbol files for the DXE modules found in target memory.
- The **GoToNextDriverEntry** button attempts to run to the entry point of the next loaded DXE driver/application.
- The **LoadSmramSymbols** button scans SMRAM memory space for EFI debug symbol information and loads it.
- The **GoToShadowedPeiCore** button attempts to run to the PeiCore function when executing in shadowed RAM.
- The **HOBs** (Hand-Off Blocks) button displays a list of UEFI HOBs found in target memory.
- The **SysConfigTable** button displays the contents of the UEFI system configuration table.
- The **DumpMemMap** button displays the UEFI Memory Map.

UEFI Debugging using SourcePoint on Intel Platforms

General UEFI Debugging

The LoadCurrent icon searches for symbols for the code at the current instruction pointer relative to the start of the module. So, should you stop the code execution in the middle of an unknown module, you can load the source and navigate to the beginning of the module in order to see where you are!

SEC and PEI Debugging

The SEC (Security) phase of code execution occurs just after the CPU comes out of reset. It is usually assembly language code as there is no memory available for a stack. Among other things, the SEC code creates a temporary memory store for use as a stack, allowing PEI to be written in 'C' language. The PEI (Pre-EFI) phase locates, validates, and dispatches PEI modules (PEIMs) that support platform features including full memory initialization. Since SEC and PEI code exists uncompressed in the boot ROM, SourcePoint can scan and locate SEC and PEI debug information at any time. Simply click the "PEIMs" button and SourcePoint will scan and load all SEC and PEI module debug information. PEI gets control shortly after target reset. PEI modules are dispatched and executed after cache RAM is mapped into system memory and the stack is initialized. To configure SourcePoint for source-level debugging of PEI code, follow these steps.

1. Open a Command View this will allow you to see the output from the next step.
2. The PEIMs button will load the program symbols and point the code view back to the beginning of the code block where the processor was stopped. Should there be an issue with the mapping of the symbols to the source tree, you will need to correct the mapping by changing where the symbol file points to or mirrors the source tree.

```
Command
P0>LoadPeims
AmtStatusCodePei                   Entry: FFDA04C0L Base: FFDA0260L "Q:\Build\Sky
BiosInfo                           Entry: FFDA12A0L Base: FFDA1040L "Q:\Build\Sky
CpuIoPei                           Entry: FFDA2420L Base: FFDA21C0L "Q:\Build\Sky
PcatSingleSegmentPciCfg2Pei        Entry: FFDA42A0L Base: FFDA4040L "Q:\Build\Sky
PiSmmCommunicationPei              Entry: FFDA6260L Base: FFDA6000L "Q:\Build\Sky
S3Resume2Pei                       Entry: FFDA8020L Base: FFDA7DC0L "Q:\Build\Sky
SiInitPreMem                       Entry: FFDB03A0L Base: FFDB0140L "Q:\Build\Sky
PeiVariable                        Entry: FFE891C0L Base: FFE88F60L "Q:\Build\Sky
FaultTolerantWritePei              Entry: FFE8BA20L Base: FFE8B7C0L "Q:\Build\Sky
CapsulePei                         Entry: FFE8D4C0L Base: FFE8D260L "Q:\Build\Sky
CapsuleX64                         Entry: FFE939E0L Base: FFE93720L "Q:\Build\Sky
DxeIpl                             Entry: FFE9C960L Base: FFE9C700L "Q:\Build\Sky
PhysicalPresencePei                Entry: FFEA5760L Base: FFEA5500L "Q:\Build\Sky
TcgPei                             Entry: FFEA6160L Base: FFEA5F00L "Q:\Build\Sky
PeiOverClock                       Entry: FFEAAC60L Base: FFEAAA00L "Q:\Build\Sky
PlatformInitPreMem                 Entry: FFEACBA0L Base: FFEAC940L "Q:\Build\Sky
CmosAccessPei                      Entry: FFEF9080L Base: FFEF8E20L "Q:\Build\Sky
DebugServicePei                    Entry: FFEFB580L Base: FFEFB320L "Q:\Build\Sky
PcdPeim                            Entry: FFEFD720L Base: FFEFD4C0L "Q:\Build\Sky
ReportStatusCodeRouterPei          Entry: FFF01AA0L Base: FFF01840L "Q:\Build\Sky
PlatformStatusCodeHandlerPei       Entry: FFF03120L Base: FFF02EC0L "Q:\Build\Sky
TraceHubStatusCodeHandlerPei       Entry: FFF0B1E0L Base: FFF0AF80L "Q:\Build\Sky
PlatformPort80HandlerPei           Entry: FFF0FA20L Base: FFF0F7C0L "Q:\Build\Sky
PeiCore                            Entry: FFFD0380L Base: FFFD0120L "Q:\Build\Sky
ReportFvRecoveryPei                Entry: FFFE7700L Base: FFFE74A0L "Q:\Build\Sky
SecCore                            Entry: FFFFD3F0L Base: FFFF3810L "Q:\Build\Sky
P0>
```

Figure 3: Command Window After Running the PEIMs Macro Function

ASSET

UEFI Debugging using SourcePoint on Intel Platforms

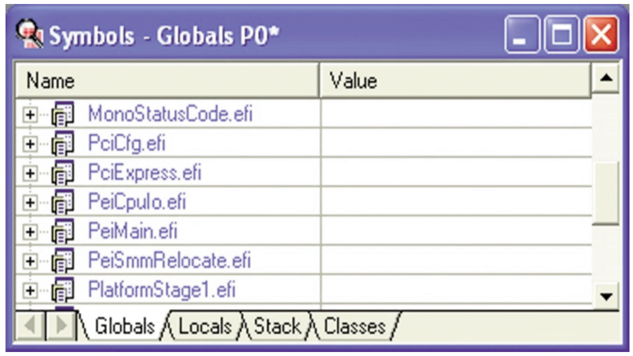

Figure 4: Symbols window after loading PEIM modules

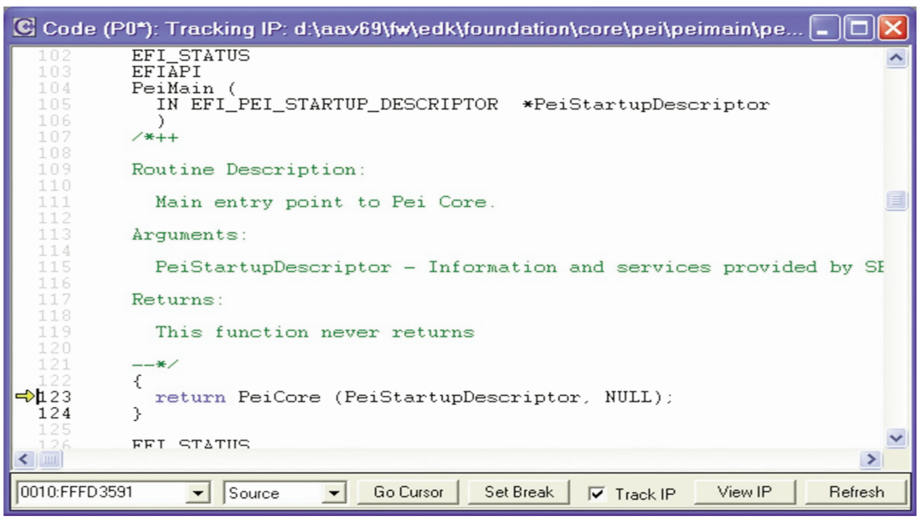

Figure 5: Code window after loading PEIM modules

Code can be traced using LBRs for pre-MRC areas and then later Intel Processor Trace (IPT) to memory when memory is available. ASSET offers several eBooks that expand on this. Figure 5 shows an example.

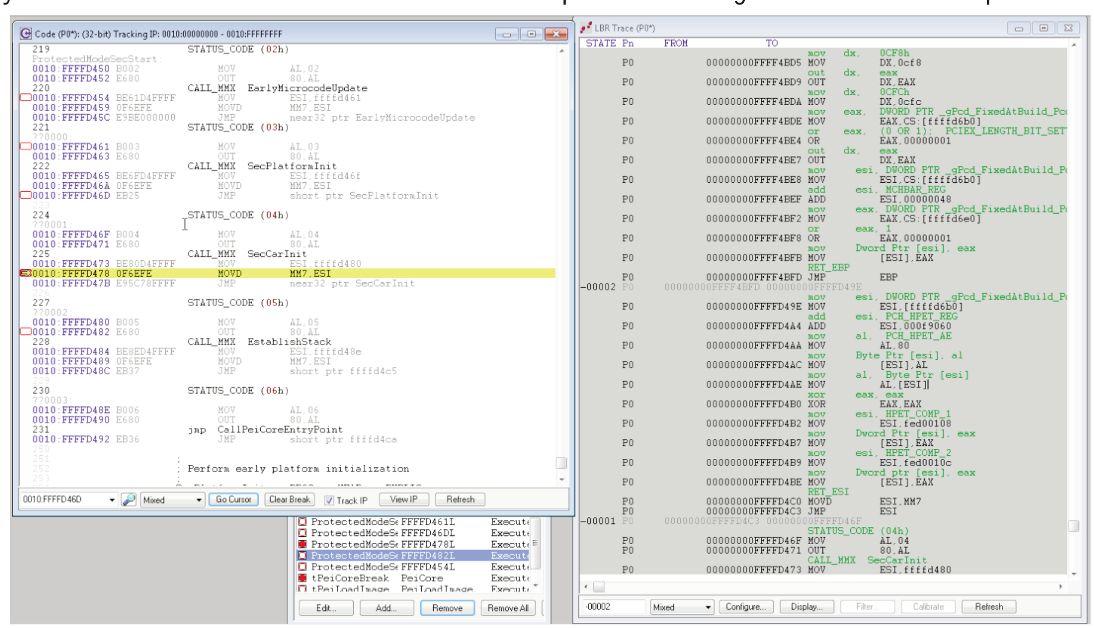

Figure 6: LBR Trace of Early SEC Code

UEFI Debugging using SourcePoint on Intel Platforms

WalkPeiDispatcher

Once PEI symbols are loaded, the "WalkPeiDispatcher" command can be entered in the SourcePoint command window. This command will attempt to break on the entry point of every dispatched PEIM and load its symbols. The result is a list of the PEIMs with the order in which they are dispatched. This command will run until PeiDispatcher returns or PeiCore is called again (usually just before shadowing to DRAM).

To execute this command in SourcePoint, follow these steps:

1. If not already opened, Open a **Command** View
2. In the **Command View** enter **WalkPeiDispatcher**()

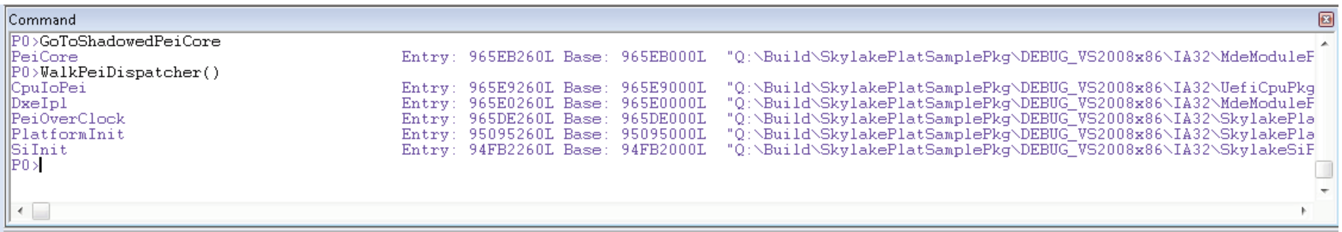

Figure 7: WalkPeiDispatcher Executed in the Command Window

Shadowed PEI Debugging

Once system RAM is initialized, some PEI code may shadow from ROM to DRAM. The PEI phase will then complete execution from DRAM before transitioning to DXE. The GoToShadowedPei button will attempt to run to the first PeiCore function call in DRAM.

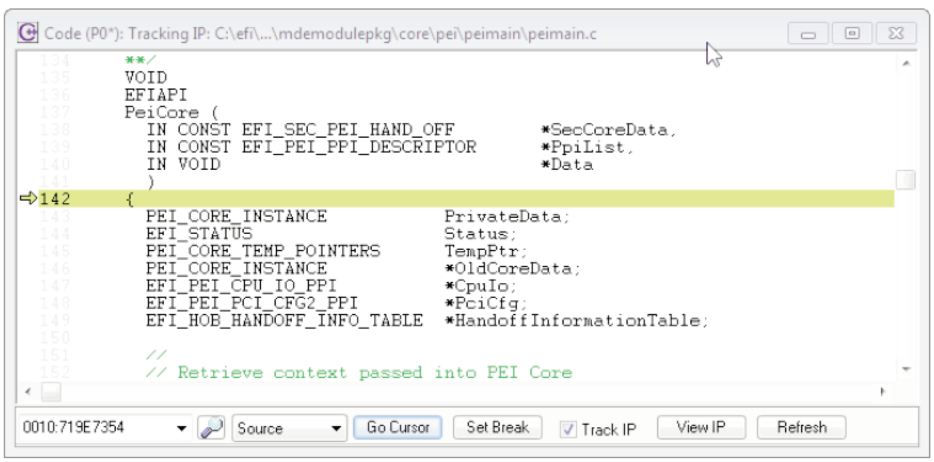

Figure 8: PEICore Shadowed in DRAM

Once there, the "WalkPeiDispatcher" command can be used to show the dispatch order of the PEIMs loaded in Shadowed PEI.

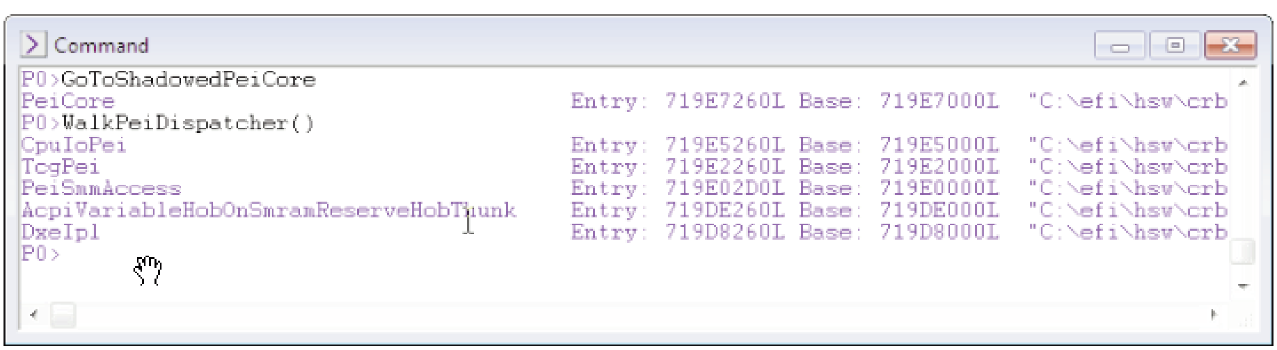

Figure 9: WalkPeiDispatcher

UEFI Debugging using SourcePoint on Intel Platforms

DXE Debugging

Once system RAM is initialized and the PEI phase completes, the DXE environment is entered. This is less specialized than PEI; nevertheless, it requires a few SourcePoint parameters to be set. The DXE drivers are compressed in the ROM, so the symbols cannot be loaded prior to the driver loading. The simplest way to load DXE driver symbols is to run the target to the UEFI shell or as far as it will go in DXE, stop the target, and then click the “DXEs” button to load all of the symbols for the DXE drivers that have been dispatched so far. At this point you should be able to browse the DXE driver symbols and set breakpoints.

To configure SourcePoint for source-level debugging of DXE code, follow these steps:

1. Run the target to the UEFI shell or as far as it will go in DXE.
2. Stop the target.
3. Click the DXEs toolbar icon to load the DXE symbols.
4. Browse the source code files using the **Symbols** window and set breakpoints in your code.
5. Reset the target and go until you hit a breakpoint.

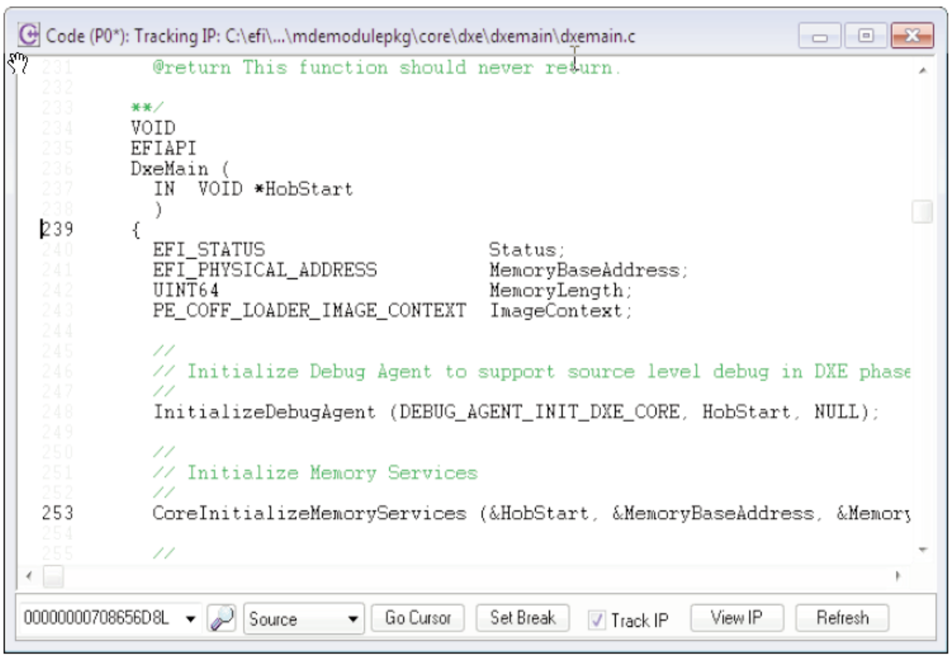

Figure 10: DXE Code Window

IMPORTANT: There are no guarantees that DXE drivers will load in the same location on subsequent boots. However, if no target hardware or software configuration changes have occurred, then in practice, the symbols should be in the same locations. If breakpoints are not working, you can reload DXE driver symbols by clicking on the DXEs button.

If your target is fatally crashing (no debug access), then the following commands can be used to try to halt before the crash occurs:

GoToDxeMain()- Attempt to locate and run to DxeMain.
GoToCoreDispatcher() - Attempt to locate and run to CoreDispatcher.
GoToNextDriverEntry() - Run to the entry point of the next loaded DXE image.
GoToNextDriverNameEntry(Name) - Run to the entry point of the DXE image that matches 'Name'. Stops at every loaded image entry point to check for a Match.
GoToDriverSymbol(DriverName, SymbolName) - Run to the code symbol 'SymbolName' contained in the Driver 'DriverName'. Uses GoToNextDriverNameEntry if needed.

UEFI Debugging using SourcePoint on Intel Platforms

HOBs

To configure SourcePoint for source-level debugging of HOB code, follow these steps:

1. If not already opened, Open a **Command** View
2. Click the HOBs toolbar icon to display the hand-off blocks on the target.

Figure 11: Example of HOB Display

System Configuration Table

To configure SourcePoint for source-level debugging of System Configuration Table, follow these steps:

1. If not already opened, Open a **Command** View
2. Click the **SysConfigTable** toolbar button to display the contents of the contents of the UEFI system configuration table on the target.

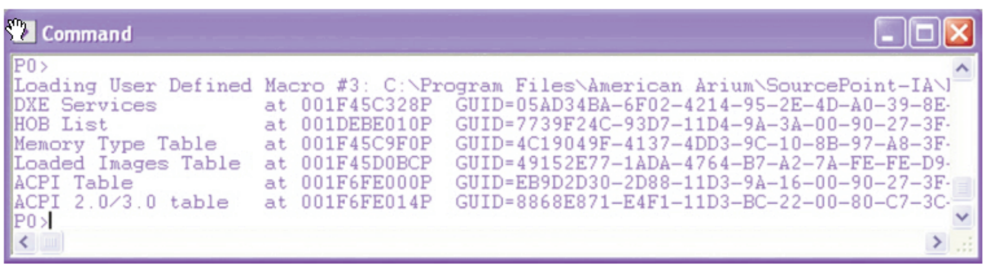

Figure 12: Example of System Configuration Table

UEFI System Memory Map

To configure SourcePoint for source-level debugging for dumping the System Memory Map, follow these steps:

1. If not already opened, Open a **Command** View
2. Click the **DumpMemMap** toolbar button to display the contents of the contents of the UEFI system memory map on the target.

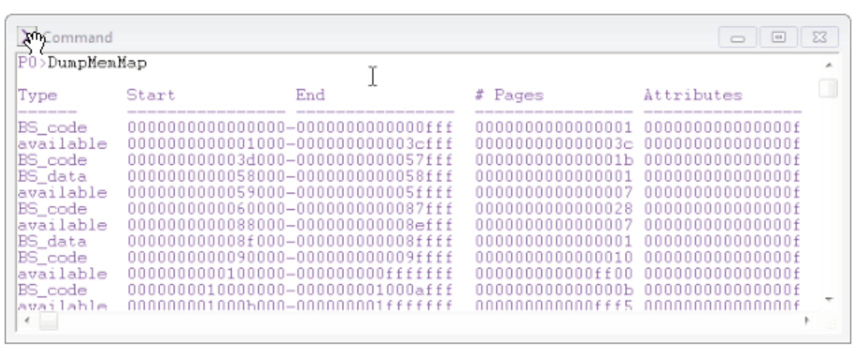

Figure 13: Example of UEFI System Memory Map

UEFI Debugging using SourcePoint on Intel Platforms

Dumping the Call Stack

To configure SourcePoint for source-level debugging for Dumping the call Call Stack, follow these steps:

1. If not already opened, Open a **Command** View
2. Click the **DumpCallStack** toolbar button to display the contents of the contents of the call stack.

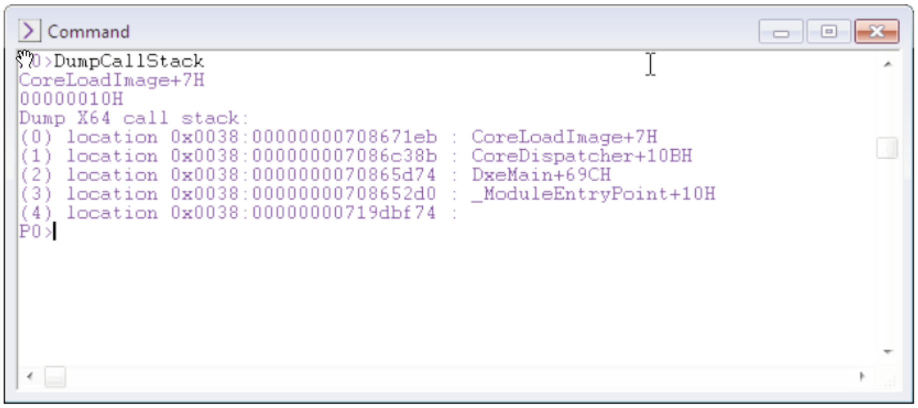

Figure 14: Example of DumpCallStack

Notes

Loading Symbols from a copied Build Tree.

When debugging an EFI firmware build on the same system where the firmware was built, the symbol file paths that are embedded in the firmware image, at build time, will match. However, if the build tree is copied to a different system in a different location, SourcePoint will prompt the user with three options:

- **Abort:** Halt all symbol loading activities
- **Retry:** Allow the user to browse to the alternate file location on this system. This will create a saved path substitution mapping used for future symbol loading. (e.g “f:=c:\efi;”)
- **Ignore:** Ignore this particular symbol file, but continue symbol loading activities. This is useful when a single module was built in a different location.

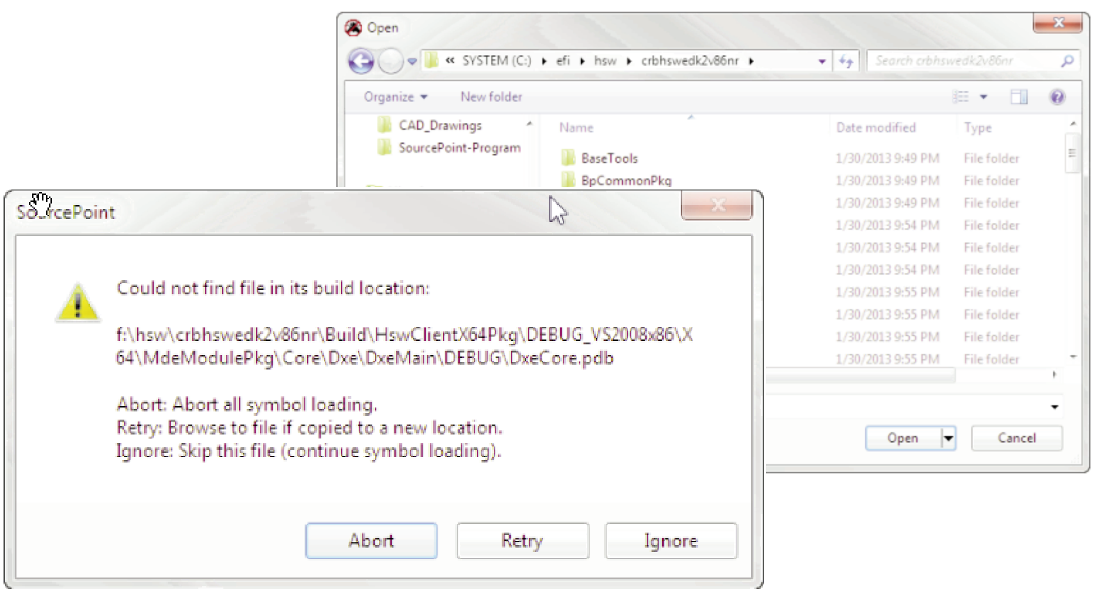

Figure 15: Repath Files

UEFI Debugging using SourcePoint on Intel Platforms

Available Commands: The following commands can be entered at the UEFI command line:

LoadSingleImage(Addr)
This function takes a code execution address and scans for relevant debug information which is loaded.

loadthis()
Scans for relevant debug information for the current IP, which is loaded.

LoadDriverName(Name)
Searches for a driver matching Name(string) and loads debug information.

LoadAllImages()
Loads symbols for all currently loaded DXE drivers

ShowDrivers()
Print out entry point address for all currently loaded DXE drivers. This function finds the EFI debug image table and walks it to show what has been loaded.

LoadDriver(Index)
Load symbols of a driver by specify the driver Index. A driver's index value is get from ShowDrivers(). This function simply calls the ShowDrivers() function with an index (passed in) to load symbols for a driver.

GoToShadowedPeiCore()
Attempt to locate and run to PeiCore in shadowed RAM.

GoToDxeMain()
Attempt to locate and run to DxeMain.

GoToCoreDispatcher()
Attempt to locate and run to CoreDispatcher.

GoToNextDriverEntry()
Run to the entry point of the next loaded DXE image.

GoToNextDriverNameEntry(Name)
Run to the entry point of the DXE image that matches 'Name'. Stops at every loaded image entry point to check for a Match.

GoToDriverSymbol(DriverName, SymbolName)
Run to the code symbol 'SymbolName' contained in the Driver 'DriverName'. Uses GoToNextDriverNameEntry if needed.

dgo()
This function tries to exit an EFI_DEADLOOP() and resume execution.

DumpAllEfiTables()

This function will dump all the EFI tables, including the EFI System Table, the Boot Service Table, the Runtime Service Table, and the Configuration Table

DumpConfigTable()
This function dumps the content in EFI Configuration Table

UEFI Debugging using SourcePoint on Intel Platforms

DumpEfiTable(Addr)
This function will dump header and content of a EFI tables at a given start address, including the EFI System Table, the Boot Service Table, and the Runtime Service Table

DumpHobs(Addr)
This function dumps the HOB list at Addr

DumpDxeHobs()
This function will find Hob list pointer in DXE Configuration Table and dump all the Hobs of this list

DumpVariable()
This function will dump content of NV variables
Usage:

DumpVariable ("VariableName") - Dump variable
DumpVariable ("*") - Dump all variables
DumpVariable ("abc*") - wildcard substitution
DumpVariable ("abc?") - wildcard substitution

DumpAcpiTable()
This function will dump ACPI tables

ShowEfiDevicePath(Addr)
This function parses content of device path in memory

DumpS3Script()
This function dumps all the entries in the S3 Boot Script Table and the Runtime Script Table

DumpCallStack()
This function dumps the call stack from the current instruction pointer

DumpExceptionContext()
This function dumps exception context preserved by UEFI code

2201 N. Central Expy., Ste 105, Richardson, TX 75080 Voice: 972-427-2800 outside the US E-mail: ai-info@asset-intertech.com. Web: www.asset-intertech.com.

A0163F-2018