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:

🎇 LoadCurrent 🖓 PEIMs 🦓 DXEs 🖓 GoToNextDriverEntry 🖏 LoadSmramSymbols

🏶 GoToShadowedPeiCore 🎕 HOBs 🆓 SysConfigTable 🏶 DumpMemMap 🖏 DumpCallStack

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.



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.
- 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:	FFE891COL 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:	FFEACBAOL Base:	FFEAC940L	"Q:\Build\Sky
CmosAccessPei	Entry:	FFEF9080L Base:	FFEF8E20L	"Q:\Build\Sky
DebugServicePei	Entry:	FFEFB580L Base:	FFEFB320L	"Q:\Build\Sky
PcdPéin	Entry:	FFEFD720L Base:	FFEFD4C0L	"Ö:\Build\Skv
ReportStatusCodeRouterPei	Entry:	FFF01AA0L Base:	FFF01840L	"Q:\Build\Sky
PlatformStatusCodeHandlerPei	Entry:	FFF03120L Base:	FFF02EC0L	"Q:\Build\Skv
TraceHubStatusCodeHandlerPei	Entry:	FFF0B1E0L Base:	FFF0AF80L	"Ö:\Build\Skv
PlatformPort80HandlerPei	Entry:	FFF0FA20L Base:	FFF0F7C0L	"Ö:\Build\Skv
PeiCore	Entry:	FFFD0380L Base:	FFFD0120L	"Ö:\Build\Skv
ReportFvRecovervPei	Entry:	FFFE7700L Base:	FFFE74A0L	"Ö:\Build\Skv
SecCore	Entry:	FFFFD3F0L Base:	FFFF3810L	"O:\Build\Skv
PO>				

Figure 3: Command Window After Running the PEIMs Macro Function



😪 Symbols - Globals P0*			
Name		Value	^
• •	MonoStatusCode.efi		
•••	PciCfg.efi		
•••	PciExpress.efi		
•••	PeiCpulo.efi		
•••	PeiMain.efi		
•••	PeiSmmRelocate.efi		
•••	PlatformStage1.efi		-
	Globals (Locals) Stack	Classes /	

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.

		LBR Trace (PO*)	
Code (Pol): (32-bit) Tracking IP: 0010:0000000 - 0010:PPPPPPP		STATE Pn	FROM TO	A
ProtectedHodeSecStart: 0010:FFFFD450 B002 MOV AL.02	<u>^</u>	PO	00000000FFFF4BD5 M0	v dx, OCF8h W DX,0cf8 t dx, eax
0010:FFFFD452 E680 OUT 80,AL 220 CALL MMX FarlyMicrocodelIndate		PO	00000000FFFF4BD9 OU	T DX.EAX
0010:FFFFD454 BE61D4FFFF MOV EST.FfFfd461 0010:FFFFD459 0F6EFE MOVD HH7.ESI		PO	00000000FFFF4BDA NO	v dx, OCFCh W DX.Ocfc w eax DWOPD PTR gPcd Fixed&tBuild Pcr
0010:FFFFD45C E9BE000000 JMP near32 ptr EarlyMicrocodeUpdate 221 STATUS_CODE (03h)		PO	00000000FFFF4BDE MO	W EAX.CS:[ffffd6b0] eax. (0 OR 1); PCIEX_LENGTH_BIT_SET
D0010:FFFFD461 B003 MOV AL.03		PO	00000000FFFF4BE4 OR	EAX.00000001
0010:FFFFD463 E680 OUT 80.AL 222 CALL_MMX SecPlatformInit		PO	00000000FFFF4BE7 OU	T DX.EAX w esi, DWORD PTR gPcd FixedAtBuild Pc
0010:FFFFD465 BE5FD4FFFFF MOV ES1, ttttd46t 0010:FFFFD46A 0F6EFE MOVD MM7.ESI		PO	0000000FFFF4BE8 M0	W ESI.CS:[ffffd6b0]
D0010:FFFFD46D EB25 JMP short ptr SecPlatformInit 223		PO	00000000FFFF4BEF AD	D ESI,00000048
224STATUS_CODE (04h)		PO	00000000FFFF4BF2 MO	W EAX.CS:[ffffd6e0]
010:FFFFD46F B004 MOV AL.04 0010:FFFFD471 E680 OUT 80.AL		PO	00000000FFFF4BF8 OR	eax, 1 EAX,00000001 Duord Ptr [esi] eax
225 CALL_MMX SecCarInit 0010:FFFFD473 BE80D4FFFF HOV ESI.ffffd480		PO	00000000FFFF4BFB MO	V [ESI], EAX
EX0110:FFFFD478 0F6EFE HOVD HH7.ESI 0010:FFFFD47B E95C78FFFF JNP near32 ptr SecCarInit		-00002 P0	00000000FFFF4BFD JM 00000000FFFF4BFD 0000000F	P EBP FFFD49E
227 STATUS_CODE (05h) 770002		PO	00000000FFFFD49E MO	<pre>v esi, DWORD PTR _gPcd_FixedAtBuild_Pr W ESI.[ffffd6b0] i</pre>
0010 FFFFD480 B005 MOV AL.05		PO	00000000FFFFD4A4 AD	D ESI, 000f9060
228 CALL_MMX EstablishStack		PO	00000000FFFFD4AA MO	V AL, PCH_HPET_AE
0010:FFFFD489 0F6EFE NOVD HN7.ESI 0010:FFFFD48C EB37 JNP short ptr fffd4c5		PO	00000000FFFFD4AC MO	V Byte Ptr [esi], ai V [ESI], AL
229 230 STATUS_CODE (06h)		PO	0000000FFFFD4AE MO	V AL. (ESI)
770003: 0010:FFFFD48E B006 MOV AL.06		PO	00000000FFFFD4B0 X0	R EAX, EAX
0010:FFFFD490 E680 OUT 80.AL 231 jmp CallPeiCoreEntryPoint		PO	00000000FFFFD4B2 MO	V ESI, fed00108
0010:FFFFD492 EB36 JMP short ptr ffffd4ca 250		PO	00000000FFFFD4B7 M0	V [ESI], EAX W eri WEFT COMP 2
251 252 Perform early platform initialization		PO	00000000FFFFD4B9 M0	V ESI.fed0010c
253	-	PO	00000000FFFFD4BE MO	V [ESI], EAX
0010.FFFFD46D • Mixed • Go Cursor Clear Break V Track IP View IP Refresh		P0 P0	00000000FFFFD4C0 MC 00000000FFFFD4C3 JM	VD ESI, MM7 IP ESI
ProtectedModeScFFFFD461L	Execute	-00001 P0	00000000FFFFD4C3 0000000F	ATUS CODE (04b)
ProtectedModeScFFFFD45DL	Execute	PO PO	00000000FFFFD46F M0 00000000FFFFD471 OU	V AL.04 T 80.AL
ProtectedModeScFFFFD4821 ProtectedModeScFFFFD4541	Execute	PO	00000000FFFFD473 M0	LL_MMX SecCarInit W ESI_ffffd480
tPeiCoreBreak PeiCore tPeiLoadImage PeiLoadImage	Execute	1		*
	Remove Al	-00012	Mixed	Filter Calibrate Betreth
Edt Add Remove	nemove All	0.000	mixed • Conligue Display.	riteresi

Figure 6: LBR Trace of Early SEC Code



Application Notes Page 4

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()**

Command		
P0>GoToShadowedPeiCore PeiCore	Entry: 965EB260L Base: 965EB000	Q:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\MdeModuleF
PU>WalkPeiDispatcher() CpuIoPei DxeIpl PeiOverClock PlatformInit SiInit	Entry: 965E92601 Base: 965E9000 Entry: 965E02601 Base: 965E0000 Entry: 965DE2601 Base: 965DE000 Entry: 950952601 Base: 95095000 Entry: 94FB22601 Base: 94FB2000	"Q:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\UefiCpuPkg "Q:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\MdeModuleF "Q:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\SkylakePla "Q:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\SkylakePla "0:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\SkylakePla "0:\Build\SkylakePlatSamplePkg\DEBUG_VS2008x86\IA32\SkylakePla
		• 4. 5414 Skyland 14052 plot is 51205_15500000 1001 Skylandsta •

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.

Code (P0*):	Tracking IP: C:\efi\\mdemodulepkg\core\pei\peimain\peimain.c	23
134 135 136 137 138 139 140 141	**/ VOID SFIAPI PeiCore (IN CONST EFI_SEC_PEI_HAND_OFF IN CONST EFI_PEI_PPI_DESCRIPTOR IN CONST EFI_PEI_PPI_DESCRIPTOR *PpiList, IN VOID *Data	^
■ 142 143 144 145 146 147 148 149 150	<pre>{ PEI_CORE_INSTANCE PrivateData; EFI_STATUS Status; PEI_CORE_TEMP_POINTERS TempPtr; PEI_CORE_INSTANCE *OldCoreData; EFI_FL_CPU_IO_PPI *CpuIo; EFI_PEI_PCI_CFG2_PPI *PciCfg; EFI_HOB_HANDOFF_INFO_TABLE *HandoffInformationTable; </pre>	
151 152 <	// // Retrieve context passed into PEI Core	-
0010:719E7354	▼ 🥖 Source 🔹 Go Cursor Set Break 🖉 Track IP View IP Refresh	

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.

> Command						×
P0>GoToShadowedPeiCore PeiCore P0>ValkPeiDicpatcher()	Entry:	719E7260L	Base:	719E7000L	"C:\efi\hsw\cr!	ь^
CpuloPei TogPei PeiSmmAccess AcpiVariableHobOnSmramReserveHobTjunk DxeIpl	Entry: Entry: Entry: Entry: Entry:	719E5260L 719E2260L 719E02D0L 719DE260L 719DE260L 719D8260L	Base: Base: Base: Base: Base:	719E5000L 719E2000L 719E0000L 719DE000L 719DE000L 719D8000L	"C:\efi\hsw\cr "C:\efi\hsw\cr "C:\efi\hsw\cr "C:\efi\hsw\cr "C:\efi\hsw\cr	ь ь ь ь
^{P0>}					,	

Figure 9: WalkPeiDispatcher



Application Notes Page 5

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.

Code	(PO*): Tracking IP: C:\efi\\mdemodulepkg\core\dxe\dxemain\dxemain.c
\$7 231	@return This function should never return.
232	
233	**/
234	VOID
236	Dremain (
237	IN VOID *HobStart
238	
239	(
240	EFI_STATUS Status;
241	EFI_PHYSICAL_ADDRESS MemoryBaseAddress;
242	UINI64 MemoryLength;
243	PE_COFF_LOADER_IMAGE_CONTEXT imageContext;
244	11
246	// Initialize Debug Agent to support source level debug in DXE phase
247	//
248	<pre>InitializeDebugAgent (DEBUG_AGENT_INIT_DXE_CORE, HobStart, NULL);</pre>
249	
250	
251	// Initialize Memory Services
252	Construction Ligo Monone Constructions (Manage Monone (Monone)
253	CoreinitializenemoryServices (@nobStart, @nemoryBaseAddress, @nemory
255	
	• • • • • • • • • • • • • • • • • • •
00000000	708656D8L 🔻 🖉 Source 👻 Go Cursor Set Break 🗸 Track IP View IP Refresh
-	

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.



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.

Command	
MOB Resource descriptor at 001DEBEC08P	^
Attributes 0x3C03	
Present	
Initialized	(1)
Uncacheable	_
Write-combinable	
Write-back cacheable	
Base address 0x00000000000000	
Length 0x0000000000000000	
HOE Resource descriptor at 001DEBEC38P	
Attributes 0x0	~
	>

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.

🌮 Command	
PO> Loading User Defined DXE Services HOB List Memory Type Table Loaded Images Table	Macro #3: C:\Program Files\American Arium\SourcePoint-IA\ at 001F45C328P GUID=05AD34BA-6F02-4214-95-2E-4D-A0-39-8E- at 001DEBE010P GUID=7739F24C-93D7-11D4-9A-3A-00-90-27-3F- at 001F45C9F0P GUID=4C19049F-4137-4DD3-9C-10-8B-97-A8-3F- at 001F45D0BCF GUID=49152E77-1ADA-4764-B7-A2-7A-FE-FE-D9-
ACPI Table ACPI 2.0/3.0 table P0>	at 001F6FE000P GUID=EB9D2D30-2D88-11D3-9A-16-00-90-27-3F at 001F6FE014P GUID=8868E871-E4F1-11D3-BC-22-00-80-C7-3C

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.

P0>DumpMe:	nMap	Т			
Type	Start	End	# Pages	Attributes	
BS_code	00000000000	00000-0000000000	0000fff 000000000000000000000000000000	1 0000000000000000	
available	000000000000	01000-00000000000	03cfff 0000000000000000	c 000000000000000000000000000000000000	
BS_data	000000000000000000000000000000000000000	58000-00000000000	058fff 000000000000000	1 000000000000000000000000000000000000	
available	000000000000	59000-0000000000	05ffff 000000000000000	7 000000000000000000000000000000000000	
BS_code	00000000000	60000-0000000000	087fff 0000000000000000	8 000000000000000000000000000000000000	
BS data	0000000000000	8f000-00000000000	086111 000000000000000000000000000000000	1 0000000000000000000000000000000000000	
BS_code	000000000000	90000-0000000000	09ffff 000000000000001	0 0000000000000000000	
available	00000000001	00000-000000000f	ffffff 000000000000ff0	100000000000000000000000000000000	
BS_code	00000000100	00000-0000000010	00afff 000000000000000	b 000000000000000000	

Figure 13: Example of UEFI System Memory Map



Application Notes Page 7

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.

> Command	
50>DumpCallStack CoreLoadImage+7H 00000010H Dump X64 call stack:	I
(0) location 0x0038:0000000708671eb: (1) location 0x0038:00000007086638b: (2) location 0x0038:000000070865474: (3) location 0x0038:0000000708652d0: (4) location 0x0038:0000000719dbf74: P0>	CoreLoadImage+/H CoreDispatcher+10BH DxeMain+69CH _ModuleEntryPoint+10H
<	•

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



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



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

