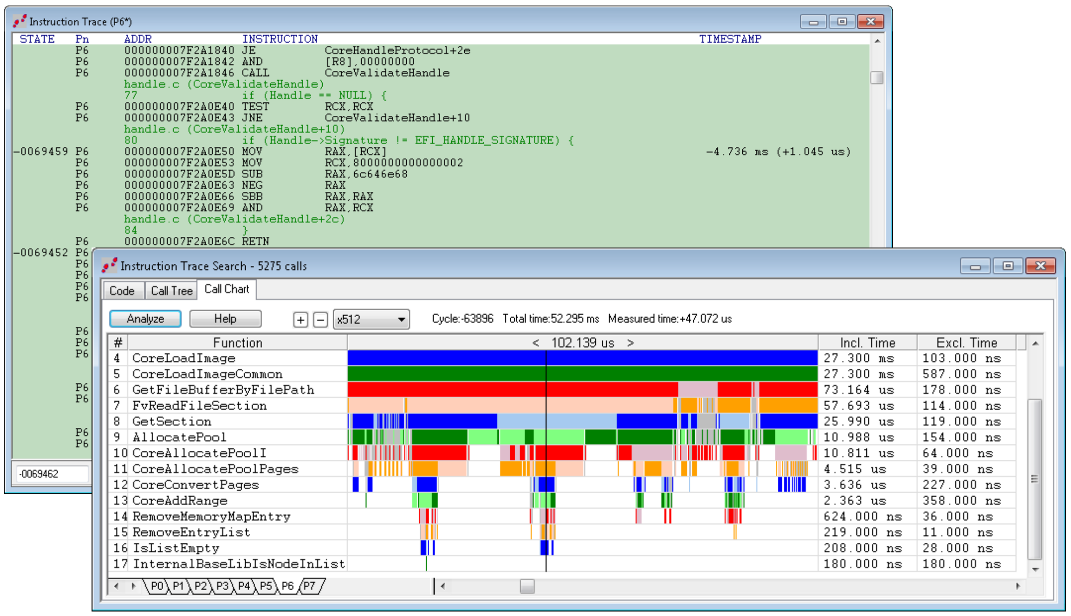
Guide to Intel® Debug and Trace



```
STATE    Pn      ADDR             INSTRUCTION                                              TIMESTAMP
         P6      000000007F2A1840 JE          CoreHandleProtocol+2e
         P6      000000007F2A1842 AND         [R8],00000000
         P6      000000007F2A1846 CALL        CoreValidateHandle
                 handle.c (CoreValidateHandle)
                 77               if (Handle == NULL) {
         P6      000000007F2A0E40 TEST        RCX,RCX
         P6      000000007F2A0E43 JNE         CoreValidateHandle+10
                 handle.c (CoreValidateHandle+10)
                 80               if (Handle->Signature != EFI_HANDLE_SIGNATURE) {
-0069459 P6      000000007F2A0E50 MOV         RAX,[RCX]                                 -4.736 ms (+1.045 us)
         P6      000000007F2A0E53 MOV         RCX,8000000000000002
         P6      000000007F2A0E5D SUB         RAX,6c646e68
         P6      000000007F2A0E63 NEG         RAX
         P6      000000007F2A0E66 SBB         RAX,RAX
         P6      000000007F2A0E69 AND         RAX,RCX
                 handle.c (CoreValidateHandle+2c)
                 84               }
         P6      000000007F2A0E6C RETN
-0069452 P6
         P6
         P6
         P6
         P6

         P6
         P6
         P6

         P6
         P6

         P6
         P6
-0069462
```

Cycle:-63896 Total time:52.295 ms Measured time:+47.072 us

#	Function	< 102.139 us >	Incl. Time	Excl. Time
4	CoreLoadImage		27.300 ms	103.000 ns
5	CoreLoadImageCommon		27.300 ms	587.000 ns
6	GetFileBufferByFilePath		73.164 us	178.000 ns
7	FvReadFileSection		57.693 us	114.000 ns
8	GetSection		25.990 us	119.000 ns
9	AllocatePool		10.988 us	154.000 ns
10	CoreAllocatePoolI		10.811 us	64.000 ns
11	CoreAllocatePoolPages		4.515 us	39.000 ns
12	CoreConvertPages		3.636 us	227.000 ns
13	CoreAddRange		2.363 us	358.000 ns
14	RemoveMemoryMapEntry		624.000 ns	36.000 ns
15	RemoveEntryList		219.000 ns	11.000 ns
16	IsListEmpty		208.000 ns	28.000 ns
17	InternalBaseLibIsNodeInList		180.000 ns	180.000 ns

BY LARRY TRAYLOR

ASSET™

Larry Traylor

Larry Traylor co-founded Arium Corporation in 1977. Larry served as president, CEO, and chairman of the board of Arium. He was instrumental in driving that company's vision for product creation of hardware-based program debug and code trace tools. In 2013, Larry joined ASSET InterTech when Arium was acquired by ASSET. He has a BSEE from Cal-Poly Pomona.

Table of Contents

Table of Figures

Acronyms	
BTM	Branch Trace Messages
DDR	Double Date Rate
DMI	Direct Media Interface
DCI	Direct Connect Interface
ME	Management Engine
MIPI	Mobile Industry Processor Interface
MMIO	Memory Mapped IO
PEI	Pre-EFI
QPI	Intel Quick Path Interconnect
SoC	System-on-a-chip
SPI	Serial Peripheral Interface
STM	System Trace Messaging
STPV2	System Trace Protocol version 2
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus

Purpose

Any new software-based product will contain newly written code in which there are bugs that must be found and fixed. This is best accomplished using quality debug tools. For an Intel processor-based computer, the firmware image containing the UEFI code must be made to work (debugged). The really hard-to-find bugs, encountered toward the end of big software projects like a UEFI port, often cause major schedule slips. Trace can really help shorten the time to find these bugs. Instruction trace is too difficult to navigate given today's code base size and complexity. Trace features combined with other newer Intel debug hooks, in the silicon, are now enabling tool features that make all phases of debug much more efficient.

Hard-to-find bugs are usually caused by asynchronous events. Examining the flow of instructions as they were executed (Instruction Trace) is the only way to see these bug causes clearly. That said, the instruction trace process used on today's code bases tends to produce huge trace files composed of millions to trillions of instructions. Finding the correct place to look is nearly impossible without a way to navigate at a higher, coarser level. Trace Hub (System Trace) provides this capability by allowing the programmer to label code states in real-time trace.

Programmers debugging BIOS (now UEFI) have been without trace for the last 20 years. Intel has only recently decided to put the machinery in the silicon to provide trace. Debug of UEFI (and other firmware) on Intel Architecture (IA) based systems can now be much more efficient.

The really hard bugs have the biggest effect on program schedule slippage. These same bugs are the ones most positively affected by the availability and use of trace.

This document is intended to point out many ways to work through these issues during the bring-up, debug and validation of a new UEFI port on an Intel-based platform.

Introduction

In an environment where there is not an operating system running yet (for example, UEFI), or when debugging the kernel of that operating system, the debug tools that provide the functionality required to get the job done must run on a separate computer from the one where the bugs are being found. This is because any quality debugger will need the features of an operating system like Windows, Linux or OSX in order to provide the debug environment expected. This arrangement is called remote-hosted debugging. It also usually provides hardware-assisted debug, which consists of some sort of hardware pod that interfaces between the computer hosting the debugger and the system on which the code is running (which is being debugged). In the very latest processors from Intel, there are features that can provide for remote-hosted debug without any extra hardware other than a cable or two.

Much of today's code is written in C or C++. Therefore, there is an abundance of code reuse and many layers of calls from the code that is sequencing a process to the code that is doing a specific piece of work. This means that the actual instruction pointer to the code that may exhibit misbehavior (bug or symptom) will provide little information about where in the overall process the error occurred. For example, a string copy routine that was passed a bad pointer does not have a bug in it just because it attempts to write to a location that causes a bus hang. The bug is related to the bad value in the pointer passed to it. Likewise, the routine that called the string copy may have only passed on the bad pointer. This sort of layered software may be many tens of levels deep. The actual bug may be somewhere very different. Finding the actual root cause is the whole point in using trace tools. This level of complexity is why instruction trace must be augmented with System Trace (Trace Hub) in order to navigate the massive instruction trace data set.

The references here to trace tools and their importance in finding asynchronously-generated bugs is not meant to decrease the value of quality source-level tools for static debug. All of these topics will be touched on here.

Backing up above the forest, there are several types of silicon and tool features that may be utilized at different times in development and for different types of problems or tasks.

UEFI development

Today, the word BIOS is basically synonymous with "UEFI" (Unified Extensible Firmware Interface). This is the firmware that runs from processor reset until the OS bootloader begins. It also contains all the drivers that the bootloader will then use. Remote hosted debuggers like SourcePoint® are critical when debugging UEFI.

The major steps in a new UEFI port are:

1. Write any additions or modifications and build a firmware image
2. Early board bring-up
3. New code debug
4. Platform-specific debug
5. Runtime failure cause determination
6. Stress testing failures (hot plug)

Once power supplies and basic hardware operation have been tested on a new board, a firmware image is loaded into a flash device (usually a SPI device). After the code is installed, the most convenient way to get started is to power up the new board and stop execution at the reset vector. This is accomplished using a debug tool. From the reset state the following steps may be taken:

1. The code is stepped or moved through manually to see that it basically works (code walking).
2. When each module or system is thought to be mostly functional, the system is then run as a whole and exercised by some form of test suite or environment.
3. Misbehaviors (bugs) are observed in the running system. These bugs are then root-caused and fixed.

In today's world, step 1 above is probably a little different. If the UEFI image was built using the latest EDK, ingredients provided by Intel, and using the "Best Known Configuration" ingredients, then probably the early code will be run to some point farther into the PEI phase. This will likely be the first occurrence of code that has been added or modified by the team bringing up the board. If this is far enough into the UEFI build, the person debugging may monitor a console to verify proper progress up to that point. If the code stops earlier, an earlier breakpoint can be set and the experiment re-run. When the boot process is successfully proceeding to the beginning of newly written or modified code, the code walking may be started. Both the author and many professional programmers believe the way to produce quality code is to walk all newly generated code at least once. This can be accomplished using a combination of step, breakpoint, and automatic breakpoints called "go to cursor" (depending on what debug tool is being used). This process can be made much easier and clearer if a quality source-level debug tool, such as SourcePoint from ASSET, is available so that the programmer is seeing the exact code that they wrote, including comments, in the debugger display.

Once the code has been walked, future verifications can be quickly accomplished using a call stack graph like the one included in ASSET's SourcePoint debugger. This graphical display requires tracing of the code. Other methods of monitoring overall progress in mostly working UEFI are via console output (printfs) or system trace. These methods will be discussed in detail in later chapters.

These activities will progress until the UEFI process displays a UEFI prompt, or an operating system is booted.

At this point stress testing will be started. To begin, the system will be booted many times, using different settings and devices. When any of this fails, the root cause will need to be identified and fixed. At some point, the time between finding flaws will become large enough that multiple platform stress testing will be required to find any remaining defects. It is these latent defects that are often the hardest to root-cause. Having a powerful, feature-rich debug tool will really help here.

There are two distinct types of debugging available to programmers in most environments today. These are:

Static Debug: This type of debug involves stopping program execution at some point by means of either a breakpoint or as the result of a step. The engineer then examines the program object values in that state. These program objects, which are accessed either directly or indirectly, are the processor registers, memory values and other hardware registers within the system. This type of debug is quite adequate for code walking as mentioned above and for easy-to-find root causes, such as those found during new code debug and platform-specific debug, when the system is run as a whole and where the coding error is near the symptom in address space and execution order.

Dynamic Debug: This type of debug records the execution of a program while it is running and, after the experiment, examines the results. This is trace. In most modern application processors, for practical reasons, this is often limited to the instructions executed and not the values (data objects) that these instructions operated on. This experiment often ends in stopping the execution, but this is not required and not always possible. Dynamic debug is the only effective method of finding the root cause of bugs found when the entire system is stressed. This type of bug involves a flaw or failure in a running system.

In the most fatal types of bugs, the system actually hangs in a way that retrieving any data about the machine state when it crashed may be very difficult. On most modern designs this will result in a catastrophic error (known as CATERR, often arising from a 3-strike error where context is lost). In this case, the only method of static debug is via facilities called "crash-dump". If there was any form of real time tracing occurring at this point it may yield better hints to determining the actual root cause of the crash.

Less fatal bugs in the boot process often result in the system simply not completing the boot process. In many of these cases, the processor core can still be stopped and state examined. As mentioned before, the exact location of the executing code is usually not the problem, but instead, is a result of the root cause. Many methods of getting an overall picture of progress are available. The simplest (and crudest) is the simple POST code display. Debug messages to the console provide a much more granular and descriptive method of feedback, but often slow the boot process so much that behavior is modified. Real-time trace in the form of instruction trace or system trace (or a combination of the two) is the most elegant and productive tool for this.

One more issue for any source level debug (static or dynamic) is the modularity and location scheme for UEFI. UEFI basically has three different types of code location and linkage. The "Framework", also known as TianoCore is a statically located hard-bound piece of code much like any other firmware. PEIM and UEFI modules are located and bound in their own specific ways. Additionally, UEFI modules are dynamically located. Good source-level debuggers that support UEFI debug have a set of button-operated macros that load the debug information at various appropriate times for convenient and smooth debug experiences.

Possible Debug Use Cases and Features

As mentioned in the previous chapters, there are several points in a project that will indicate the use of different types of features offered by debug tools. These specific tool use features and configurations that are often called use cases. For an Intel processor processor-based design with major UEFI work done, the use cases in timeline order might be:

1) Port new code to an Intel supplied development platform with new silicon. Tool features:
   a) Static Source level Debug using Run-Control on the Intel CRV
   b) POST code display
   c) Console debug messaging OR Trace Hub message tracing
   d) Full suite of trace tools
2) Move new code to OEM platform and finish debug
   a) Tool features: Same as 1), but functional on OEM platform
3) Stress test a new product in Stress lab with many instances running
   a) Tool features: Same as 1) PLUS hot-plug; STM may also be helpful here

With these use cases as the target debug issues to address, the list of tool/silicon features required to expediently get the job done are:

1) **Static Source-Level Debug including**
   a) Run-control directly out of reset or power-up
   b) Source level debug of the UEFI code base including fast symbol searching across multiple programs (a UEFI code base is composed of many dynamically loaded programs)
   c) Debugger features that deal with UEFI source location
2) **Overall State Monitoring**
   a) A POST code display is the crudest form but still indicates last state achieved (this does little for showing the sequence leading to the state nor any source reference)
   b) Console tracing (printfs which slows the code so much it often changes the problem being examined)
   c) Event trace (**STM** is the general acronym for System trace). This type of trace is achieved using the Trace Hub in Intel devices. STM displays show a timestamped version of all console outputs plus other state markers
      i) There are two forms: SVEN from Intel (still a little invasive); and Tool-Hosted Printf from ASSET InterTech, which is very fast.
   d) **Instruction trace** which when time-correlated to STM provides the details of the program execution
3) **Hot-Plug**
   a) This is a hardware feature allowing a system that has hung or entered an error state to be stopped and queried even if there was no debugger attached when the error occurred.
4) **Crash Dump**: a method where the state is dumped from a target that has experience a catastrophic error (CATERR)

These four types of features will be expanded on in subsequent sections of this document.

It should be noted, that in order to facilitate all of these displays in a Source based form that the UEFI code must be built with the debug features enabled. The UEFI framework is structured such that the location of each EFI program is dynamically assigned, and its identity and location must be determined with the UEFI hooks intended for this.

Debug topologies vs chip type

Not all debug features are available in all generations of Intel chips. The following table summarizes some of this:

	Run-Control	Instruction Trace	Trace Hub	DCI-OOB	DCI-DbC
Broadwell	X	-	-	-	-
Skylake	X	X	X	X	-
Coffee Lake	X	X	X	X	X
Beyond	X	X	X	X	X

The upcoming sections will describe these different types of Debug and Trace feature support.

Event Trace

Introduction to Trace Hub and its relation to Intel PT

Intel added the Trace Hub to its processor system in order to provide several trace features that augment the newly added Intel Processor Trace (Intel PT). These features include (1) coalescing of trace streams from different sources with IDs and timestamps, (2) System Trace Module feature (STM), and (3) common multipath trace transport. In the first generation of Intel chips to offer the Trace Hub (server and client solutions) the primary advantage is the STM feature set. This allows for code instrumentation (similar to printf, but in real time) in several code bases and multiple cores. The post-processed trace stream can then be time-correlated to the Intel PT trace streams to provide the trace navigating facilities described in this eBook.

The Intel Trace Hub is a piece of hardware (IP within the Intel chips) that is a slave on the MMIO fabric of a given platform or SOC. The vagueness here is necessary because Trace Hub is used in both SOCs as well as the largest server platforms. A write to a memory space that is not routed to one of the actual DDR memory controllers or north-complex IO may be destined for the Trace Hub. In client or server sets of chips today, this means that writes to the Trace Hub are routed down the DMI fabric. In SOCs the nature of the fabric is different. In addition there can be other transports that stimulate trace message generation in the Trace Hub.

In the most basic use, programs running on any core (IA, ME, or other) can write debug messages (think printf or logging) and direct them to Trace Hub instead of to a serial port, console or memory log. This means they are timestamped and correlated with instruction trace. It also means they can be transported in a variety of ways depending on target, silicon and target state.

The address space of the slave interface to the Trace Hub is divided into masters and each master is subdivided into channels. Writes to different addresses within a channel result in the production of different message types. All messages and timestamp packets are eventually formatted in STPV2 which is a MIPI standard for system trace encoding and transport. The nature of this protocol is intended to be generated with Master/Channel message identification. In this way the message identification is implicit instead of needing to be handled by the software under debug. This mechanism provides for a more code efficient (low insertion loss) method of instrumenting code. This means instrumentation inserted in the code has less effect on the code size and speed of execution.

In client and server chip sets the important sources for software debug are:

(A) Direct program writes from the host cores
(B) Direct program writes from the ME
(C) AET (Architectural Event Trace) (generated by the core for selected architectural event types)

In SOCs, in addition to the above, Intel PT may be routed through the Trace Hub. A simple view of the Trace Hub is shown in Figure 1.

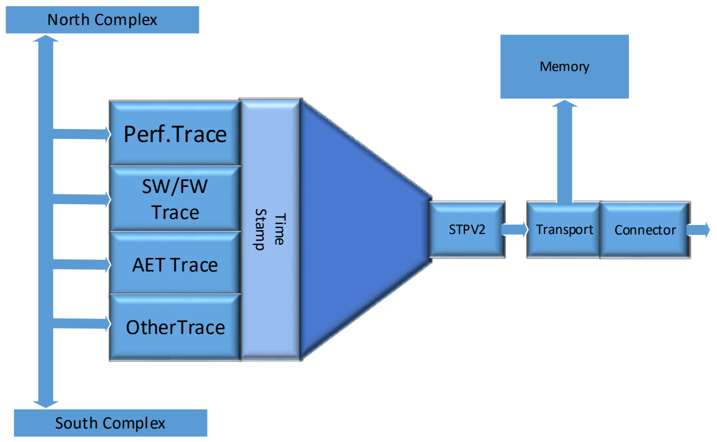


Figure 1: Simple View of Trace Hub

In Figure 1, above, there is a time stamp unit shown. This unit can place a time stamp on each individual trace message. There is a timestamp alignment mechanism that allows the Trace Hub time stamp to be directly correlated to each Intel PT (Instruction Trace) stream in the system. Therefore, all views of each of the types of trace can be time-aligned with all other views. This means locating an event in a Trace Hub sequence will directly point to a spot in the instruction trace.

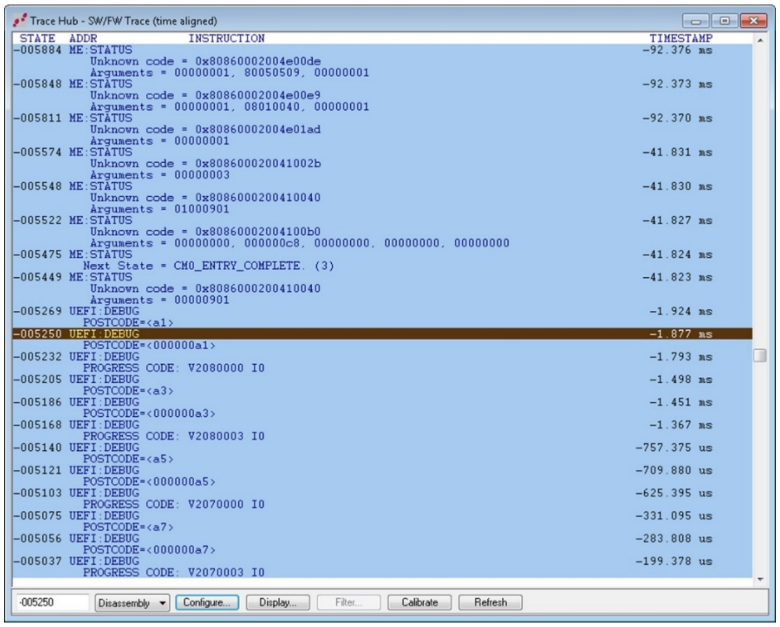
How Trace Hub can shorten the time to find the really hard bugs.

Isolating a bug usually begins by observing a symptom (blue screen?) and then attempting to set a breakpoint at the point in the code that is generating the symptom (this might be as simple as searching the code base for the string that is printed at the point of the symptom. A breakpoint can then be set on the code that outputs that string). For simple bugs, this is almost the end of the process as it is likely that statically inspecting the code will quickly reveal the bug. Harder to find bugs almost always show up at this point as a piece of code operating on bad data. The executing code is not making an error; it is just processing data that is not correct.

With execution (instruction) trace, it is then possible to see what function called the function processing the wrong data. In a large body of code like UEFI, this is still likely to be a properly operating piece of code that is, again, operating on bad data. This story, and the time it takes to wade through trace, can go on and on. A much more effective way to view the trace at this point is to look at a less fine-grained resolution view of it. A call analysis tool might be a good solution. This can show the software engineer what major functions are running and which have just run. It will be obvious what called the offending code and passed down bad data.

For really complex situations, STM trace with good instrumentation in the code base can make this quick and easy. There needs to be instrumentation in the code that outputs messages when major functions of the code base are started or important nodes within these functions are reached. It turns out that the UEFI code base already has many such lines of instrumentation which currently are passed to the serial port or to the console (depending on where in the BOOT process the message is generated). In EDK2-generated UEFI many of these messages are generated with a "DEBUG" macro which uses syntax much like printf, including a printf string. These STM messages can easily be redirected to the Trace Hub. With this small change to the UEFI code base, there is now the two-level trace system described here. In the future, these messages can be augmented to take even better advantage of STM with Processor Trace.

A source level software debugger, like SourcePoint from ASSET InterTech, should have trace list displays which clearly display the formatted STM messages, with time stamps, and with the ability to directly synchronize the cursor in Intel Processor Trace to the STM message trace listing. In this way moving from the macro view to the micro view is trivial. This is illustrated in Figure 2 below.



```
 STATE  ADDR               INSTRUCTION                                                    TIMESTAMP
-005884 ME:STATUS                                                                         -92.376 ms
          Unknown code = 0x80860002004e00de
          Arguments = 00000001, 80050509, 00000001
-005848 ME:STATUS                                                                         -92.373 ms
          Unknown code = 0x80860002004e00e9
          Arguments = 00000001, 08010040, 00000001
-005811 ME:STATUS                                                                         -92.370 ms
          Unknown code = 0x80860002004e01ad
          Arguments = 00000001
-005574 ME:STATUS                                                                         -41.831 ms
          Unknown code = 0x808600020041002b
          Arguments = 00000003
-005548 ME:STATUS                                                                         -41.830 ms
          Unknown code = 0x8086000200410040
          Arguments = 01000901
-005522 ME:STATUS                                                                         -41.827 ms
          Unknown code = 0x80860002004100b0
          Arguments = 00000000, 000000c8, 00000000, 00000000, 00000000
-005475 ME:STATUS                                                                         -41.824 ms
         Next State = CM0_ENTRY_COMPLETE. (3)
-005449 ME:STATUS                                                                         -41.823 ms
          Unknown code = 0x8086000200410040
          Arguments = 00000901
-005269 UEFI:DEBUG                                                                         -1.924 ms
         POSTCODE=<a1>
-005250 UEFI:DEBUG                                                                         -1.877 ms
         POSTCODE=<000000a1>
-005232 UEFI:DEBUG                                                                         -1.793 ms
         PROGRESS CODE: V2080000 I0
-005205 UEFI:DEBUG                                                                         -1.498 ms
         POSTCODE=<a3>
-005186 UEFI:DEBUG                                                                         -1.451 ms
         POSTCODE=<000000a3>
-005168 UEFI:DEBUG                                                                         -1.367 ms
         PROGRESS CODE: V2080003 I0
-005140 UEFI:DEBUG                                                                       -757.375 us
         POSTCODE=<a5>
-005121 UEFI:DEBUG                                                                       -709.880 us
         POSTCODE=<000000a5>
-005103 UEFI:DEBUG                                                                       -625.395 us
         PROGRESS CODE: V2070000 I0
-005075 UEFI:DEBUG                                                                       -331.095 us
         POSTCODE=<a7>
-005056 UEFI:DEBUG                                                                       -283.808 us
         POSTCODE=<000000a7>
-005037 UEFI:DEBUG                                                                       -199.378 us
         PROGRESS CODE: V2070003 I0
```

Figure 2: STM trace of UEFI and ME

In addition to using STM to provide a guide to what functions are running, it can also be used to trace the data a program is operating on. Intel PT, much like many other trace mechanisms on high performance application processors, does not trace the values of data being operated on. It only traces which instructions were executed. All decisions in code are made on data values (A==B). Many times, to truly understand why a program decision went wrong, it is necessary to know the value of a program variable at some specific point in that program. It is not practical to trace all data all the time. This would consume two much trace transport bandwidth. STM allows inserting very small code snippets that dump particular values just at the time it is desired to test the value. These data probes are inserted temporarily when working through a problem. Again, they are timestamped, so that correlating with Instruction Trace is easy. This process in general is referred to as data tracing.

In summary, the use of Trace Hub described here provides navigation, drill down, and data value probe capabilities. These features are exactly what is needed to make it easy and more efficient to work with large buffers of Instruction Trace. Again, it is Instruction Trace that will actually reveal the specific software malfunction.

What it takes to use the Trace Hub

There are several things involved in setting up an environment to use Intel's Trace Hub. Some of these are within the tool set used, some are within the target software being debugged, and some may involve hardware configuration. Exactly how each item is accomplished is dependent on the version of Intel chips you are using as well as the debug tools you are using. The various items required for 6th generation Intel chips include:

A. Target:
   (1) Provide an area in RAM to store trace buffers. In the UEFI world this is accomplished by allocating a reserved, non-cacheable area for all trace buffers. This is described in detail in the BIOS writers specification provided by Intel.
   (2) Build in STM messages as needed into the code bases that are to be debugged (instrument your code). For console/serial port messages from UEFI, this is a simple modification to a couple of files. ME messages are built into the firmware distributed by Intel.

B. Target, via debug tool:
   (1) Set up registers that enable the Trace Hub as well as point to the storage buffer. In SourcePoint this is all done via settings in the SourcePoint GUI interface.

C. Via Tool at experiment time:
   (1) Establish an end point for the experiment. This is usually a breakpoint (trigger) that stops the processors from executing.
   (2) Run the experiment
   (3) Select the display desired in the tool and the tool will automatically perform the trace buffer post processing, using the trace buffer contents, the object files associated with the source files and possibly the target static information. This last item is dependent on tool set up. The post processing of STM information will also require a metadata file that provides target/program information required for the post processing of the raw STPV2 data.

Once this is all set up, each successive experiment will cause the tools to capture and prepare all of this automatically so that the displays are available at the click of a button.

STM via Trace Hub overcomes performance issues of Printf techniques

UEFI, like other embedded code bases, is riddled with printf style debug statements. In the EDK2-based version of UEFI these statements are actually macros which call debug print routines with several possible destinations. These macros, such as “DEBUG”, use a syntax and process which includes, as an argument, a printf style text formatting string. The processing code to turn the arguments into a readable string can run thousands of instructions in the target code being debugged. This execution time added to the backpressure caused by synchronous call to drivers for slow transports such as RS232 serial IO can cause the execution speed of the boot code to increase from tens of seconds to several minutes. Not only is this console logging process time consuming for the software engineer, it often changes the nature of the code running program enough to stop the bug from occurring.

The speed of the Trace Hub as a transport, in addition to its other advantages, removes the effect of the transport induced backpressure. This improvement combined with modern tools that move string formatting to the tools and to display time, almost completely remove the timing effects of enabling all of the debug messaging.

Taking these advantages even farther, ASSET InterTech's SourcePoint offers a proprietary version of this tool-based string formatting that makes it transparent to the software engineer. Thus, the engineer adds messages just like before, yet gains all the advantages of ASSET's "Tool-Hosted Printf".

Instruction Trace

How Intel Processor Trace compresses the information

Intel Processor Trace, like many trace algorithms today, limits the data it carries to time-stamped instruction-flow information. In the most compressed form, a portion of a trace stream of bytes will simply represent taken/not-taken branches in the execution stream. In this mode, each byte represents up to 6 branches, and this will usually represent 30 or more executed instructions. Along with these taken/not-taken packets there are several other packet types that include new address packets (when needed), time-stamp information, and other auxiliary packet types. Some of these packets are periodic while others are as-needed.

This format is more completely described in the referenced Intel manual. It provides a very compressed trace stream for collecting and post processing by tools like SourcePoint.

Trace features used by ASSET InterTech's SourcePoint tools

One of the most important features of "High Speed Processor Trace" in Intel's newer ICs is that it is nearly full speed. It has no significant impact on the execution speed of the program being executed. In contrast, when using BTMs with BTS (storage to memory) there was a minimum of a 60% slow down. For some code this could be much greater. This change in execution speed could often impact whether a bug does or does not occur. High Speed Processor Trace (HSPT) has no measurable impact on the experiment.

In addition, HSPT has several types of time stamp available in most of its instantiations. Using cycle-accurate timestamp, time can be measured with a resolution of the processor clock. In later instantiations, global timestamp will allow alignment with all threads and all other trace sources, including AET and other new sources.

This highly-compressed, full-speed trace, when enabled, provides instrumentation of an operating program to allow for examination of the exact sequence of execution of instructions, including asynchronous sequences like exceptions and external interrupts.

These trace features are on par with trace methods found in other architectures and will produce the results that firmware engineers have come to expect. These features can also be used at the application level and for diagnosing system faults.

The older Intel Trace methods:

In the ten years prior to Intel's introduction of Processor Trace, Intel had only two methods of instruction trace. These were LBR, and BTM to BTS.

LBR trace was based on a relatively small number of pairs of last-branch-record registers. A typical number was 8 or 16 pair. Each pair of registers would record the "from" and "to" addresses of the last 8 or so changes of execution flow. This could typically record 40 or 50 instructions. This would rarely capture all of the last interrupt. Due to the small depth of this trace, it often did not contain the fault-producing event.

The other method available was BTM to BTS. This was cumbersome to setup and use, and it slowed the execution of the processor greatly. This often altered the problem being diagnosed. Because of the difficulty in use and the speed issue, most programmers did not use this feature.

Neither of these types of trace had any notion of a timestamp, so, many post-processing features could not be implemented, such as time-based execution call graphs and statistics views.

Use Models and Advantages for High Speed Trace

There are many powerful uses for instruction trace. These include several types of defect root-cause determination, understanding of performance issues, and gaining a quick overview of the execution of a program or process.

The classical and still most compelling use of instruction trace is in finding the interference in a particular program sequence by an asynchronous event. There is nothing more frustrating than finding the place that a program is making the wrong decision, only to discover that you have no way of telling what altered that data object upon which the errant decision is based. In a simple case, it could be a matter of determining what code sequence called this code; call stacks can often show this. In a difficult case, the data may have been changed by code running in an interrupt that was not supposed to modify the data in question (maybe from an errant pointer?). In this case, even though the software engineer may be able to reliably trigger the debug tool on the exact errant decision, he has no way of knowing what piece of code modified the bad data or why. Using trace to see what code preceded the bad decision will usually yield the culprit in a very short amount of time. This can literally save weeks in diagnosing a blue screen or Linux "oops".

Another very common use of trace is to actually measure which parts of code are contributing to the execution time of a function. Nothing shows this more clearly than a statistics view of the code operating at full speed. This can often directly show the engineer exactly which pieces of code can be optimized for the maximum improvement. Measuring these times using real-time trace allows making the measurement without altering the experiment.

These are just two examples of how using instruction trace can take weeks out of a development schedule. Many developers of embedded programs have been using trace for years and the number of ways it can be used to diagnose problems is almost limitless. Firmware (UEFI) development on Intel-based computers can now take advantage of these silicon/tool features.

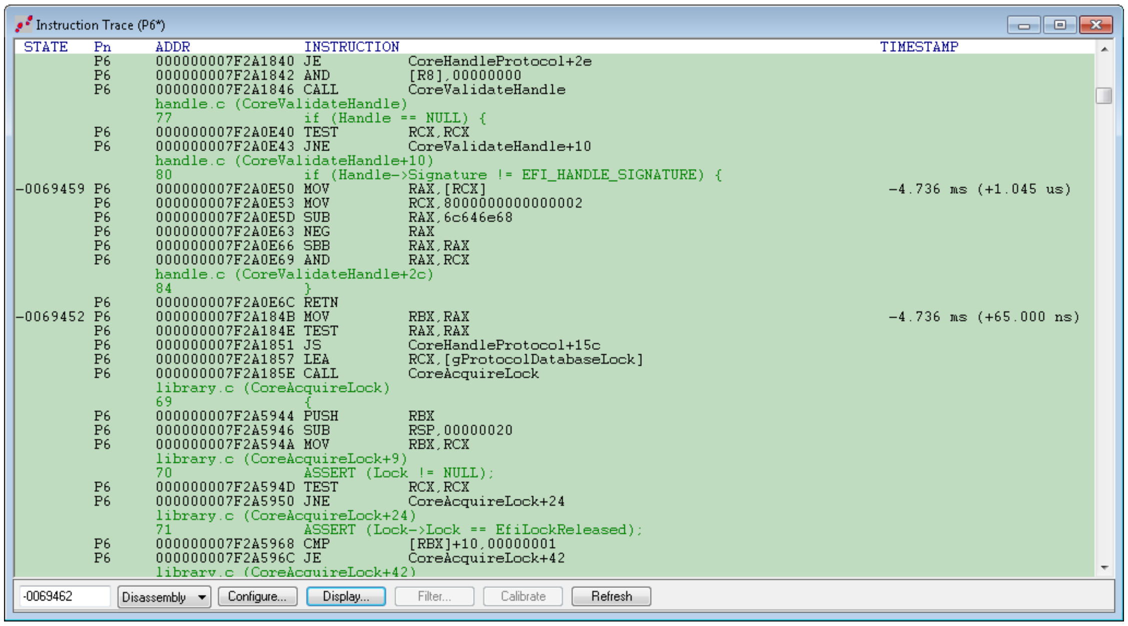
How trace can be displayed in modern tools like SourcePoint

Now that High Speed Processor Trace is available in Intel chips, development engineers can take advantage of the powerful features in tools like SourcePoint. SourcePoint has many different ways it can display data collected in a trace buffer. Conventional displays that are list-based are available with many format options ranging from simple disassembly to full source display. These displays are enhanced by many features such as flyover symbol. In addition to the classical list-based displays, SourcePoint offers several trace post-processing features and displays that make it very easy to visualize code execution at a high level and then drill down to the line-by-line views. Modern large trace buffers, in the gigabyte range, make examining trace detail without good browsing tools impractical. SourcePoint offers several types of post-processing tools which include:

- Structured Search
- Call Charts
- Call Graphs
- Statistical Summary Displays.

Several of these will be described in detail in later sections.

The most basic trace display is the list display. In SourcePoint, the software engineer can select the items to be displayed and control the color coding to differentiate multiple trace sources. The lines are time-stamped and can be used to index into other displays such as source windows, chart windows, or other trace list displays. The lines can be assembly, source, or mixed. An example of a list display is shown in Figure 3. This display shows both assembly and source-level depiction of the executed code. This list display is very configurable by the user in SourcePoint.



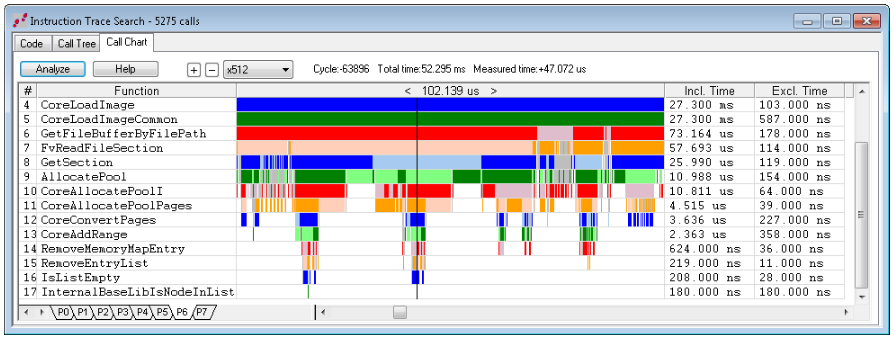
```
STATE    Pn      ADDR              INSTRUCTION                                                  TIMESTAMP
         P6      000000007F2A1840 JE         CoreHandleProtocol+2e
         P6      000000007F2A1842 AND        [R8],00000000
         P6      000000007F2A1846 CALL       CoreValidateHandle
                 handle.c (CoreValidateHandle)
                 77               if (Handle == NULL) {
         P6      000000007F2A0E40 TEST       RCX,RCX
         P6      000000007F2A0E43 JNE        CoreValidateHandle+10
                 handle.c (CoreValidateHandle+10)
                 80               if (Handle->Signature != EFI_HANDLE_SIGNATURE) {
-0069459 P6      000000007F2A0E50 MOV        RAX,[RCX]                                           -4.736 ms (+1.045 us)
         P6      000000007F2A0E53 MOV        RCX,8000000000000002
         P6      000000007F2A0E5D SUB        RAX,6c646e68
         P6      000000007F2A0E63 NEG        RAX
         P6      000000007F2A0E66 SBB        RAX,RAX
         P6      000000007F2A0E69 AND        RAX,RCX
                 handle.c (CoreValidateHandle+2c)
                 84               }
         P6      000000007F2A0E6C RETN
-0069452 P6      000000007F2A184B MOV        RBX,RAX                                             -4.736 ms (+65.000 ns)
         P6      000000007F2A184E TEST       RAX,RAX
         P6      000000007F2A1851 JS         CoreHandleProtocol+15c
         P6      000000007F2A1857 LEA        RCX,[gProtocolDatabaseLock]
         P6      000000007F2A185E CALL       CoreAcquireLock
                 library.c (CoreAcquireLock)
                 69               {
         P6      000000007F2A5944 PUSH       RBX
         P6      000000007F2A5946 SUB        RSP,00000020
         P6      000000007F2A594A MOV        RBX,RCX
                 library.c (CoreAcquireLock+9)
                 70               ASSERT (Lock != NULL);
         P6      000000007F2A594D TEST       RCX,RCX
         P6      000000007F2A5950 JNE        CoreAcquireLock+24
                 library.c (CoreAcquireLock+24)
                 71               ASSERT (Lock->Lock == EfiLockReleased);
         P6      000000007F2A5968 CMP        [RBX]+10,00000001
         P6      000000007F2A596C JE         CoreAcquireLock+42
                 library.c (CoreAcquireLock+42)
```

Figure 3: A SourcePoint List Display

Call Graph Display

When a large amount of trace has been captured, and the code base is extensive, looking at the detail of the trace is very tedious. The Call Graph display allows the SourcePoint user to look at large portions (or even all of the trace buffer) and view it in a graph showing call depth. Each line in this graph can represent a different function at different points in time. Changes in color represent changes in a function. Each line moving downwards represents another level of call depth. A moveable cursor points to specific points on the timeline (x-axis of graph). The left-hand column displays the names of the functions, at each level, at the point indicated by the cursor.

The controls above the graph allow the user to expand the graph (zoom in) at the point indicated by the cursor. Figure 4 shows an actual trace of a range of UEFI in the boot-up of an Intel based computer. This is a good illustration of the power of this viewer:

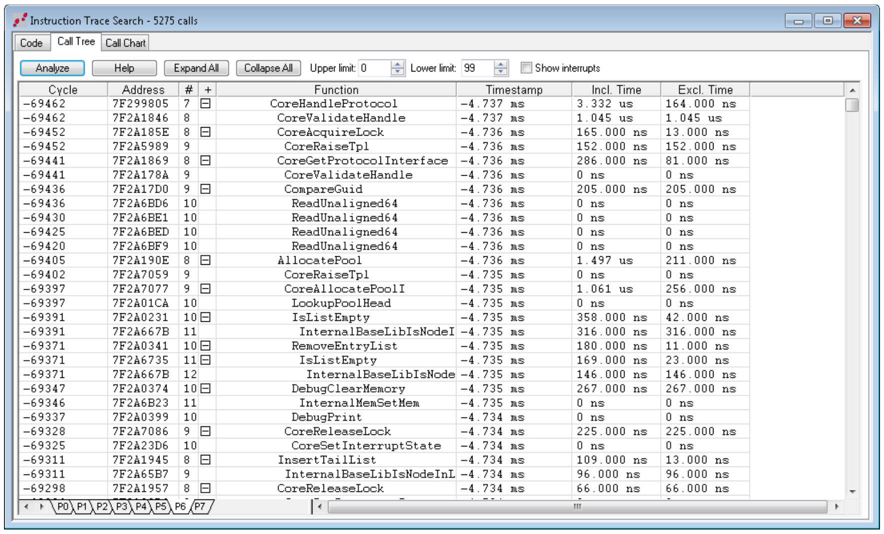


Cycle:-63896 Total time:52.295 ms Measured time:+47.072 us

#	Function	< 102.139 us >	Incl. Time	Excl. Time
4	CoreLoadImage		27.300 ms	103.000 ns
5	CoreLoadImageCommon		27.300 ms	587.000 ns
6	GetFileBufferByFilePath		73.164 us	178.000 ns
7	FvReadFileSection		57.693 us	114.000 ns
8	GetSection		25.990 us	119.000 ns
9	AllocatePool		10.988 us	154.000 ns
10	CoreAllocatePoolI		10.811 us	64.000 ns
11	CoreAllocatePoolPages		4.515 us	39.000 ns
12	CoreConvertPages		3.636 us	227.000 ns
13	CoreAddRange		2.363 us	358.000 ns
14	RemoveMemoryMapEntry		624.000 ns	36.000 ns
15	RemoveEntryList		219.000 ns	11.000 ns
16	IsListEmpty		208.000 ns	28.000 ns
17	InternalBaseLibIsNodeInList		180.000 ns	180.000 ns

Figure 4: Sample Instruction Trace of UEFI Code

Another way of looking at the same information is with the Call Chart. In this view, specific areas can be drilled into by function name, expanding or collapsing as desired. Both of the call views can be ***synchronized*** to a list view so that the specific code can be examined at the point of interest. Figure 5 is a Call Chart display.

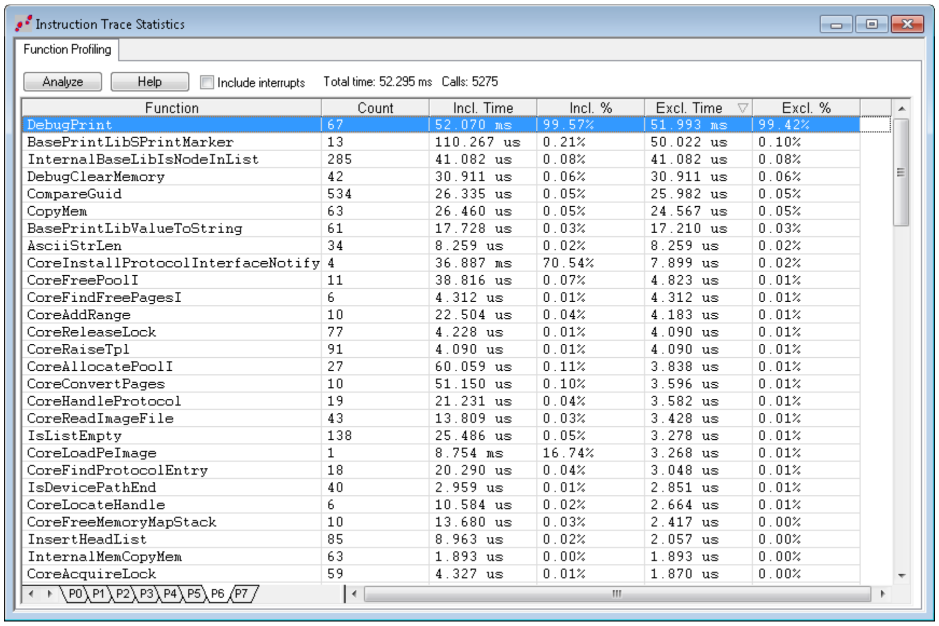


Cycle	Address	#	+	Function	Timestamp	Incl. Time	Excl. Time
-69462	7F299805	7	⊟	CoreHandleProtocol	-4.737 ms	3.332 us	164.000 ns
-69462	7F2A1846	8		CoreValidateHandle	-4.737 ms	1.045 us	1.045 us
-69452	7F2A185E	8	⊟	CoreAcquireLock	-4.736 ms	165.000 ns	13.000 ns
-69452	7F2A5989	9		CoreRaiseTpl	-4.736 ms	152.000 ns	152.000 ns
-69441	7F2A1869	8	⊟	CoreGetProtocolInterface	-4.736 ms	286.000 ns	81.000 ns
-69441	7F2A178A	9		CoreValidateHandle	-4.736 ms	0 ns	0 ns
-69436	7F2A17D0	9	⊟	CompareGuid	-4.736 ms	205.000 ns	205.000 ns
-69436	7F2A6BD6	10		ReadUnaligned64	-4.736 ms	0 ns	0 ns
-69430	7F2A6BE1	10		ReadUnaligned64	-4.736 ms	0 ns	0 ns
-69425	7F2A6BED	10		ReadUnaligned64	-4.736 ms	0 ns	0 ns
-69420	7F2A6BF9	10		ReadUnaligned64	-4.736 ms	0 ns	0 ns
-69405	7F2A190E	8	⊟	AllocatePool	-4.736 ms	1.497 us	211.000 ns
-69402	7F2A7059	9		CoreRaiseTpl	-4.735 ms	0 ns	0 ns
-69397	7F2A7077	9	⊟	CoreAllocatePoolI	-4.735 ms	1.061 us	256.000 ns
-69397	7F2A01CA	10		LookupPoolHead	-4.735 ms	0 ns	0 ns
-69391	7F2A0231	10	⊟	IsListEmpty	-4.735 ms	358.000 ns	42.000 ns
-69391	7F2A667B	11		InternalBaseLibIsNodeI	-4.735 ms	316.000 ns	316.000 ns
-69371	7F2A0341	10	⊟	RemoveEntryList	-4.735 ms	180.000 ns	11.000 ns
-69371	7F2A6735	11	⊟	IsListEmpty	-4.735 ms	169.000 ns	23.000 ns
-69371	7F2A667B	12		InternalBaseLibIsNode	-4.735 ms	146.000 ns	146.000 ns
-69347	7F2A0374	10	⊟	DebugClearMemory	-4.735 ms	267.000 ns	267.000 ns
-69346	7F2A6B23	11		InternalMemSetMem	-4.735 ms	0 ns	0 ns
-69337	7F2A0399	10		DebugPrint	-4.734 ms	0 ns	0 ns
-69328	7F2A7086	9	⊟	CoreReleaseLock	-4.734 ms	225.000 ns	225.000 ns
-69325	7F2A23D6	10		CoreSetInterruptState	-4.734 ms	0 ns	0 ns
-69311	7F2A1945	8	⊟	InsertTailList	-4.734 ms	109.000 ns	13.000 ns
-69311	7F2A65B7	9		InternalBaseLibIsNodeInL	-4.734 ms	96.000 ns	96.000 ns
-69298	7F2A1957	8	⊟	CoreReleaseLock	-4.734 ms	66.000 ns	66.000 ns

Figure 5: A SourcePoint Call Chart Display

Using the Statistics View to Tune Execution Times

Without trace, it can be very difficult to determine the execution time of the various areas in the program. Very often some programmed operation, such as booting up a computer with UEFI, takes longer than desired. It may not be obvious what portions of the code are the real culprits. SourcePoint's statistics view can be used to quickly find out exactly where the time is being spent. Figure 6 shows the statistics view in SourcePoint.



Total time: 52.295 ms Calls: 5275

Function	Count	Incl. Time	Incl. %	Excl. Time	Excl. %
DebugPrint	67	52.070 ms	99.57%	51.993 ms	99.42%
BasePrintLibSPrintMarker	13	110.267 us	0.21%	50.022 us	0.10%
InternalBaseLibIsNodeInList	285	41.082 us	0.08%	41.082 us	0.08%
DebugClearMemory	42	30.911 us	0.06%	30.911 us	0.06%
CompareGuid	534	26.335 us	0.05%	25.982 us	0.05%
CopyMem	63	26.460 us	0.05%	24.567 us	0.05%
BasePrintLibValueToString	61	17.728 us	0.03%	17.210 us	0.03%
AsciiStrLen	34	8.259 us	0.02%	8.259 us	0.02%
CoreInstallProtocolInterfaceNotify	4	36.887 ms	70.54%	7.899 us	0.02%
CoreFreePoolI	11	38.816 us	0.07%	4.823 us	0.01%
CoreFindFreePagesI	6	4.312 us	0.01%	4.312 us	0.01%
CoreAddRange	10	22.504 us	0.04%	4.183 us	0.01%
CoreReleaseLock	77	4.228 us	0.01%	4.090 us	0.01%
CoreRaiseTpl	91	4.090 us	0.01%	4.090 us	0.01%
CoreAllocatePoolI	27	60.059 us	0.11%	3.838 us	0.01%
CoreConvertPages	10	51.150 us	0.10%	3.596 us	0.01%
CoreHandleProtocol	19	21.231 us	0.04%	3.582 us	0.01%
CoreReadImageFile	43	13.809 us	0.03%	3.428 us	0.01%
IsListEmpty	138	25.486 us	0.05%	3.278 us	0.01%
CoreLoadPeImage	1	8.754 ms	16.74%	3.268 us	0.01%
CoreFindProtocolEntry	18	20.290 us	0.04%	3.048 us	0.01%
IsDevicePathEnd	40	2.959 us	0.01%	2.851 us	0.01%
CoreLocateHandle	6	10.584 us	0.02%	2.664 us	0.01%
CoreFreeMemoryMapStack	10	13.680 us	0.03%	2.417 us	0.00%
InsertHeadList	85	8.963 us	0.02%	2.057 us	0.00%
InternalMemCopyMem	63	1.893 us	0.00%	1.893 us	0.00%
CoreAcquireLock	59	4.327 us	0.01%	1.870 us	0.00%

Figure 6: SourcePoint's Statistics View

Note that in all of the trace post-processing displays there is a tab per processor. The list display, which can be time-aligned to all these other displays, can be one display per processor or multiple color-coded trace displays.

The exact time-stamp features, and therefore the exact alignment features available, differ from one Intel processor to another. For exact features for a specific processor please contact your local ASSET representative.

Other Features of SourcePoint that Make Use of Trace

In addition to the post-processing screens shown, SourcePoint has several other popular trace-processing features. Trace views may be searched in either simple textual algorithms or in structured algorithms that evaluate addresses and data and search for those. Also, all trace views contain flyover examination of data objects. Code windows and symbol windows can be opened, referencing cursor-selected objects in the trace window.

SourcePoint also contains some of the most powerful and quick symbol-finding/evaluating dialogs available in any tool. These symbol-search tools work across program modules making them extremely convenient when used in a UEFI environment.

Consent Considerations

For several generations of Intel processors (going back to around 2012), there has been a mechanism called "Consent" that enables (or disables) the ability of a particular system to support remote-hosted debug. The major effect of Consent is enabling the capability of the cores in a processor to enter probe mode. Forms of consent have varied over the past five or so generations, and the exact details in more current silicon cannot be covered here due to confidentiality requirements.

That said, consent may involve some or all of the following:

1) "switches" that may be set by Intel at chip manufacturing time
2) "switches" that should be set late in the OEM manufacturing process
3) Softer "switches" that are set when the firmware image is created
4) On some generations there are hardware straps that might be controlled by a debug tool. These straps affect consent but are often not the complete solution.

It is important to note that these mechanisms have changed significantly in very recent generations. Due to Intel confidentiality issues, the reader should either engage with Intel directly or their tool vendor of choice for assistance. In either case the correct agreement with Intel must be in place to enable discussion.

Different classes of targets will treat Consent differently. Reference boards from Intel containing early silicon will usually have some form of Consent enabled so that firmware images can be debugged. For OEM early boards the OEM will need to work through some of these issues in order to use a debugger.

Closed chassis debug may require a different set of enablement to facilitate debug.

The transition to DCI

Beginning with the processor family that was code named Skylake a new mechanism of connectivity for debug was introduced. This technology is called “Direct Connect Interface” (DCI). This technology provides for remote-hosted debugging using a standard connector already found on products to be debugged. The first versions of this use a USB connector. In some cases the protocols are also industry standard while in other cases they are not.

There are two basic configurations available today for DCI debug use. They are:

1) BSSB bridge -- This is an adaptor that bridges between a standard USB port on a debugger host (DTS – Debug & Test System) and a target to be debugged. It may plug into a USB3 port on the target which is equipped to support “DCI-OOB”.
    a. These adaptors (Closed Chassis Adapter - CCA from Intel or Closed Chassis Controller - CCC from ASSET InterTech) provide a robust debug connection that on some generations of silicon covers almost all power management flows.
    b. The target connections for popular forms of DCI-OOB often require special signal requirements on the debug target (TS). This might be a problem for some product configurations.
2) Debug Class (DbC) -- This is a direct cable connection from the host (DTS) to the target (TS). The cable is actually specialized and depends on the exact situation. A very important consideration is to ensure that there is not a conflict between devices that could drive VBUS on the cable. Intel sells several cables designed for this purpose. Examples include type A to type A and type A to type C. For the type C versions the configuration pins must be strapped for the specific, non-standard use (this often will consist of

3) configuring at least one port to be upward facing in a basically downward facing connector).
    a. DbC has the advantage of not needing any special “pod”. It is usually just a cable from the DTS to the TS.
    b. DbC has even more consent issues than tool topologies using an ITP connector.
    c. DbC in all generations to date has some power state debug flows that it does not support. For UEFI debug these will usually be covered more robustly using a BSSB bridge (CCA or CCC) or an XDP-based tool.

DCI will continue to become more and more prevalent as a debug tool for Intel-based designs. It will be covered in much more depth in a future edition of this document.

Conclusion

The use of appropriate debug tools can make a huge impact on both quality and schedule effectiveness in the development of and the porting of UEFI-based firmware for use on Intel processor-based products. Important features of these debug tools include robust source handling, trace features, and modern debug transport compatibility. Source level debug for UEFI requires not only strong handling of high-level language data elements and multiple program segments, but also requires the ability to find and associate the source using UEFI-specific debug hooks and procedures. There are many phases to a project to deploy UEFI on new design. Different phases require different tool features. Debug tool choices should be an important consideration in any UEFI project.