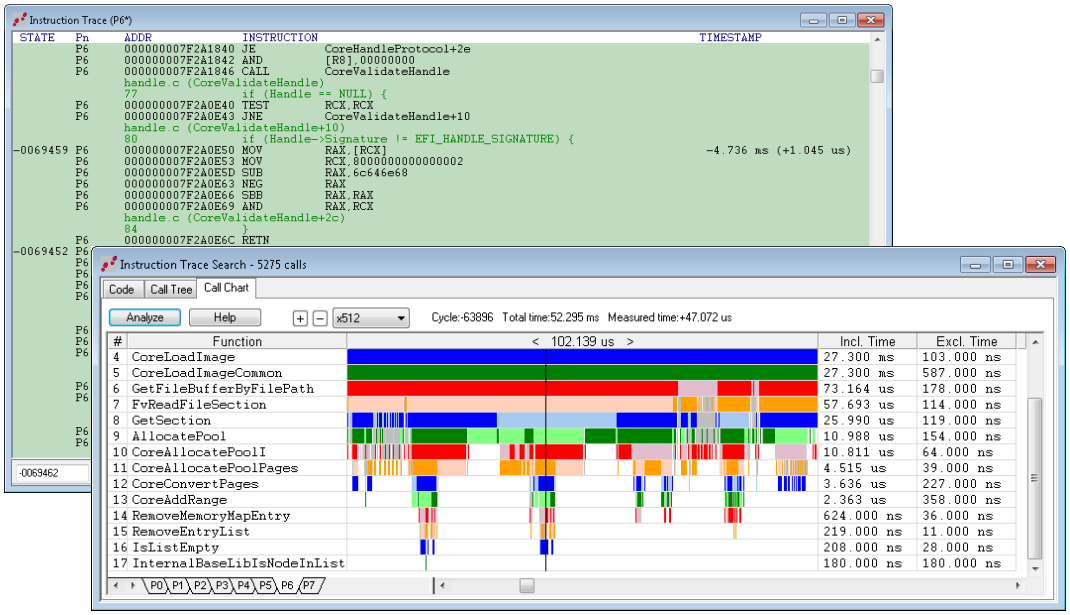
GUIDE TO INTEL® DEBUG AND TRACE



BY ALAN SGUIGNA AND 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.

Alan Sguigna

Alan Sguigna has more than 20 years of experience in senior-level general management, marketing, engineering, sales, manufacturing, finance and customer service positions. Before joining ASSET, he worked in the telecom industry. He has had profit and loss responsibility for a $150 million division of Spirent Communications, a supplier of test products and services. An avid technical writer and blogger, Alan is author of The MinnowBoard Chronicles, a journey of discovery in x86 architecture, UEFI, debug & trace, Yocto Linux and other topics.

Table of Contents

Purpose 5

Introduction 6

UEFI development 7

Possible Debug Use Cases and Features 10

Debug Topologies vs CPU 12

Event Trace 13

- Introduction to Trace Hub and its relation to Intel PT 13
- How Trace Hub can shorten the time to find the really hard bugs 14
- What it takes to use the Trace Hub 17
- STM via Trace Hub overcomes performance issues of Printf techniques 18

Architectural Event Trace (AET) 20

Instruction Trace 22

- How Intel Processor Trace compresses the information 22
- Trace features used by ASSET InterTech's SourcePoint tools 22
- The older Intel Trace methods 23
- Use Models and Advantages for High Speed Trace 23
- How trace can be displayed in modern tools like SourcePoint 24
- Call Graph Display 25
- Using the Statistics View to Tune Execution Times 26
- Other Features of SourcePoint that Make Use of Trace 27

Debug Consent Considerations 29

The Transition to DCI 30

Conclusion 32

Table of Figures

Figure 1: Simple View of Trace Hub 14
Figure 2: STM trace of UEFI and ME 16
Figure 3: At-Speed Printf Output 19
Figure 4: A SourcePoint List Display 25
Figure 5: Sample Instruction Trace of UEFI Code 26
Figure 6: A SourcePoint Call Chart Display 26
Figure 7: SourcePoint's Statistics View 27

Tables

Table 1: CPU versus Debug/Trace Support 12
Table 2: AET Events 21

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 by itself is often insufficient to navigate given today's code base's size and complexity; but Instruction Trace combined with Event Trace, enabled by newer Intel logic in the silicon, is now enabling tool JTAG-based functionality that makes 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, coreboot, and other firmware), 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 (the "host") from the one where the bugs are being found (the "target"). This is because any quality debugger will need the features of an operating system like Windows in order to provide the debug environment expected. This arrangement is called remote-hosted debugging. Remote-hosted debugging also provides for hardware-assisted debug, which consists of some sort of hardware pod that interfaces between the host and the target. In the very latest processors from Intel, there are features that can provide for JTAG-based remote-hosted debug without any extra hardware other than a specialty USB cable.

Much of today's firmware 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 serial 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 serial console output (printf’s) 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 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 can be 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 CRB
   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; Trace Hub 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 provides any source reference)
   b) Console tracing (though, printf’s slow the code so much it often changes the problem being examined)
   c) Event trace (**STM** is the general acronym for System Trace Module). 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 At-Speed 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, 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 CPU

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

Table 1: CPU versus Debug/Trace Support

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 (Intel Architecture (IA), Management Engine (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 that they are timestamped and correlated with instruction trace. It also means that 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 chips, 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 below:

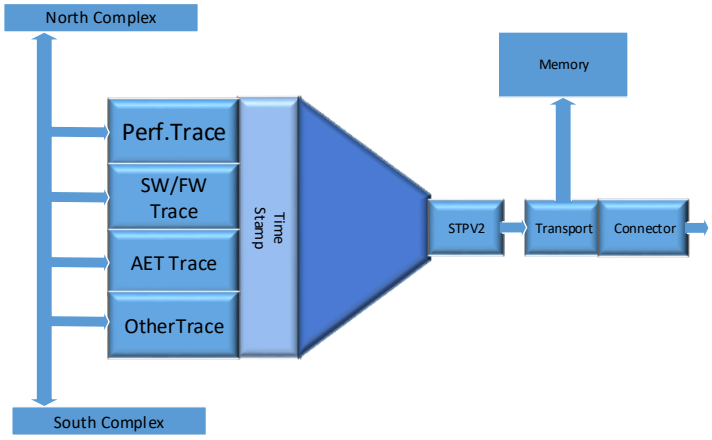


Figure 1: Simple View of Trace Hub

In the figure 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 timestamps, 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 the figure below:

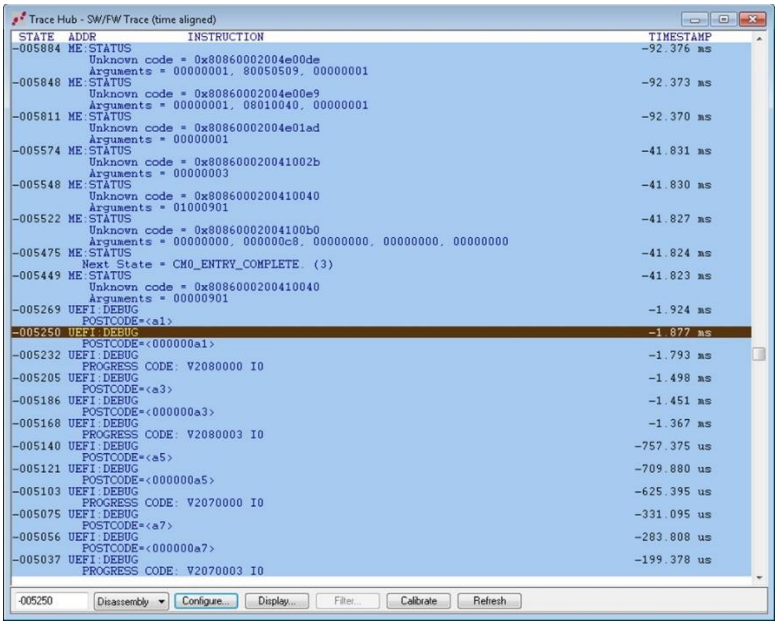


Figure 2: STM trace of UEFI and ME

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 toolset 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 and up 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 Guide 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 and macros.

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 setup. 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 execution 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 host-based debugger, 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 “At-Speed Printf” (ASPf). This utility can help in the isolation to the most timing-sensitive, intermittent bugs that are often detected in release builds, but vanish in debug builds. Race conditions, concurrency issues, and other non-deterministic behavior can thus be debugged. An example of some ASPf output, displayed within SourcePoint versus in a serial console application, is shown below.

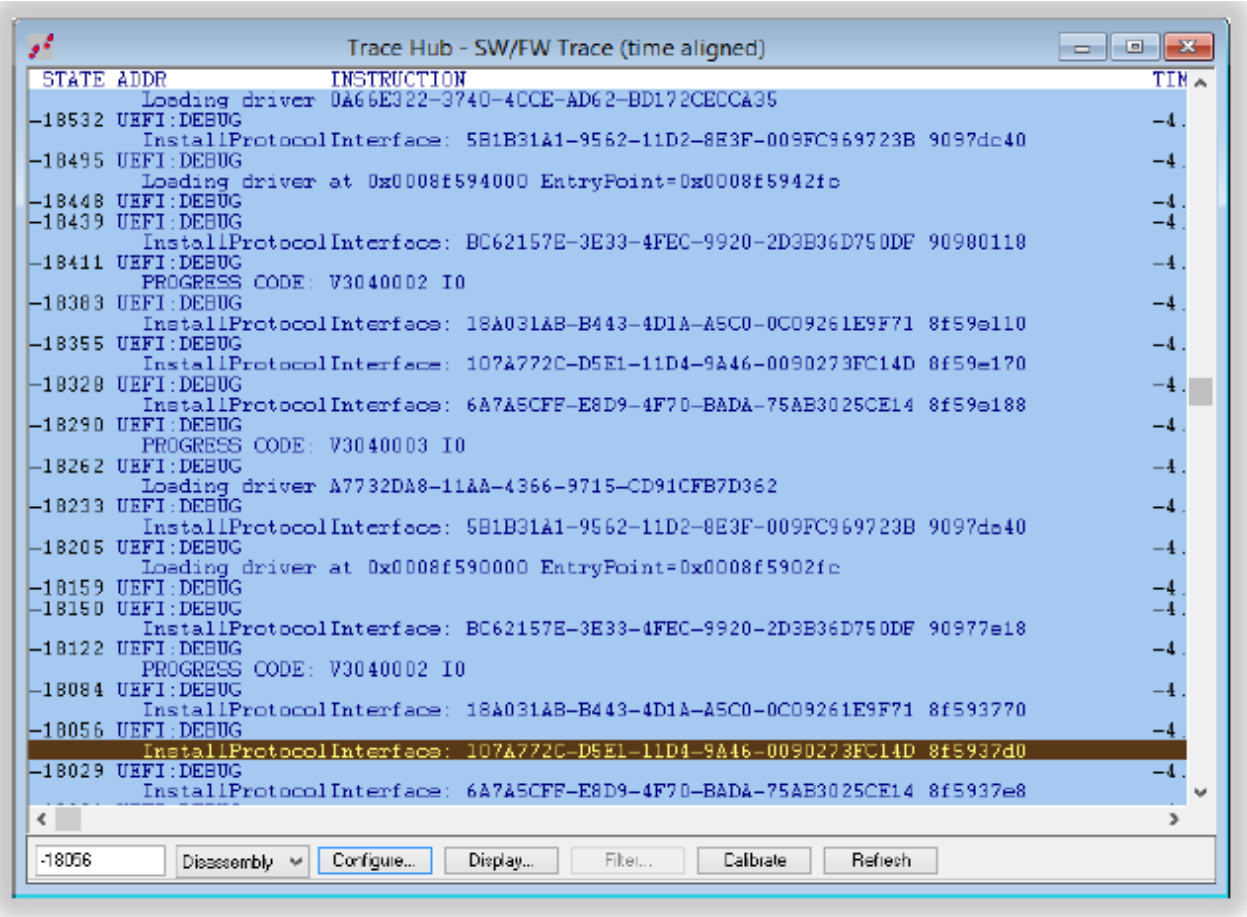


Figure 3: At-Speed Printf Output

Architectural Event Trace (AET)

AET, enabled by the Intel Trace Hub, is a technology that enables a running processor to capture event information in real-time. It differs from execution trace in that it is more well-suited to understanding the bigger picture on a platform, looking at patterns and interactions between individual processors, the BIOS and operating system, device drivers, and external peripherals.

Probe mode (i.e. JTAG, or run-control) is needed to enable AET. Use outside of probe mode will results in a General Protection Exception (#GP).

AET is actually implemented within CPU microcode, so its use does not interfere at all with the normal system operating mode. Thus, there is no need to instrument the code at all. However, low-level debug and trace logic can be changed to provide heretofore unachievable trace functionality. An example of this is with the Code Breakpoint and Data Breakpoint AET events. By setting an instruction breakpoint, and then using AET to treat it as an Event, the processor will not halt when the breakpoint is encountered; rather, the event data is captured (with timestamp and other correlated data), and the processor will run right through the code without posting a debug exception. When combined with LBR Trace or Intel Processor Trace (but not at the same time), this facility is ideal for debugging critical sections of code, concurrency issues, memory accesses, etc.

Also note that since AET reports on processor interactions rather than instruction execution, it is still extremely useful even if source code is not available. But, with source code, unparalleled insight into associations between the code and events can be obtained.

The Event Types, Subtypes, and a Description of each event supported by AET is as follows:

Event Type	Event SubTypes	Description
HW/SW Interrupt	HW_INTR	HW interrupt trace
IRET	IRET	IRET trace
Exception	Exception	Exception, fault, trap trace
MSR	RDMSR, WRMSR	MSR trace
Power Management	POWER_ENTRY, POWER_EXIT	Power management
IO	PORT_IN, PORT_OUT, PORT_IN_ADDR	IO trace
SGX	AEX, EENTER, ERESUME, EEXIT	SGX trace
CODE_BP	CODE_BP	Code breakpoint trace
DATA_BP	DATA_BP	Data breakpoint
FIXED_INT	SMI, RSM, NMI	“Fixed” interrupt trace
SW_POWER	MONITOR/MWAIT	MONITOR/MWAIT trace
WBINVD	WBINVD_BEGIN, WBINVD_END	Write-back invalidate trace

Table 2: AET Events

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), timestamp 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 Intel Processor Trace 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 Branch Trace Messages (BTMs) with Branch Trace & Store (BTS) there was a minimum of a 60% slowdown. For some code this could be much greater. This change in execution speed could often impact whether a bug does or does not occur. Intel Processor Trace has no measurable impact on the experiment.

In addition, Intel Processor Trace has several types of time stamp available in most of its instantiations. Using cycle-accurate timestamps, time can be measured with a resolution of the processor clock. In later instantiations, global timestamp now allows alignment with all threads and all other trace sources, including AET and other new sources. This provides the ultimate in diagnosability. All current Intel CPUs, as of the time of writing, support the global timestamp.

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 Last Branch Record (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, 16 or 32 pair. Each pair of registers would record the "from" and "to" addresses of the last eight or so changes of execution flow. This could typically record 100-300 instructions. This would rarely capture all of the last interrupt. Due to the small depth of this trace, it often did not, by itself, isolate 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 masked 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 classic 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 statistical 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 Intel Processor Trace is available in their CPUs and SoCs, 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 classic 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.

These will be described in detail in these upcoming 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 below. This display shows both assembly and source-level depiction of the executed code. This list display is very configurable by the user in SourcePoint.

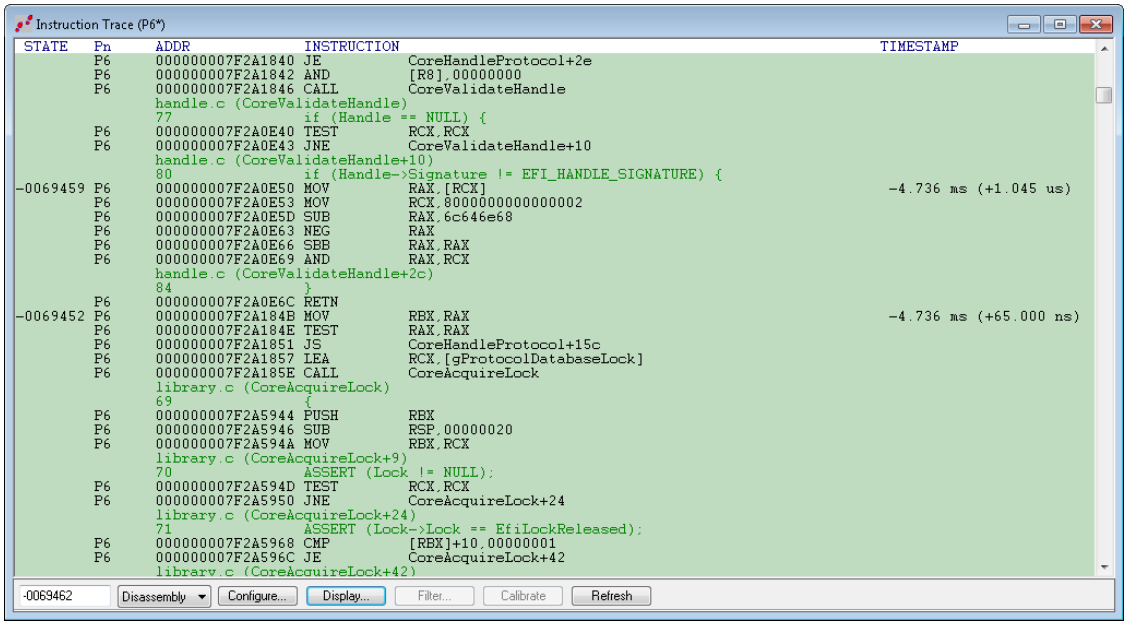


Figure 4: A SourcePoint List Display

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. The illustration below 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:

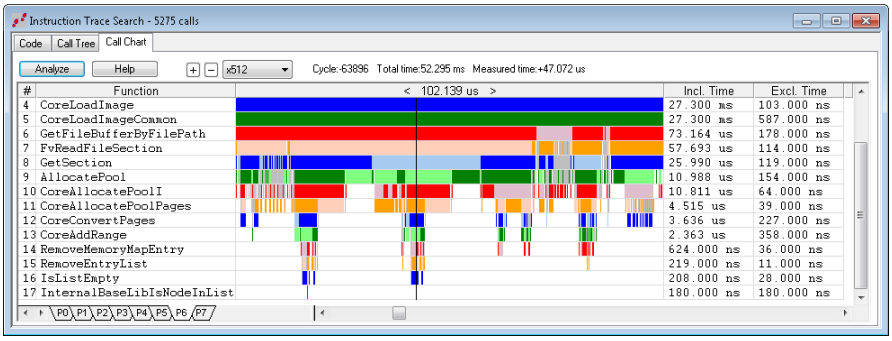


Figure 5: Sample Instruction Trace of UEFI Code

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:

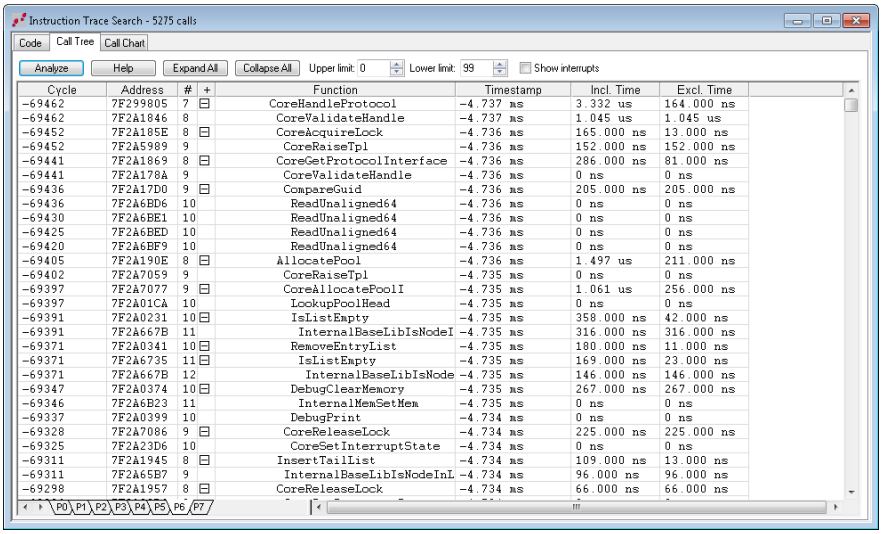


Figure 6: A SourcePoint Call Chart Display

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. The figure below shows the statistics view in SourcePoint:

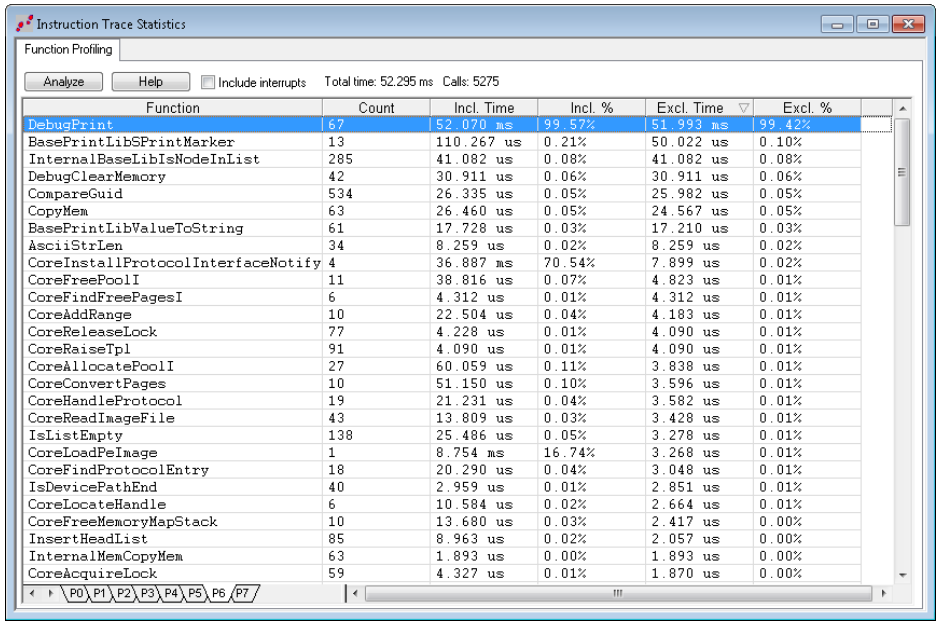


Figure 7: SourcePoint's Statistics View

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 timestamp 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.

Debug Consent Considerations

For several generations of Intel processors (going back to around 2012), there has been a mechanism called “Consent” or “Debug 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, that is, run-control. 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. This is covered in the next section.

The Transition to DCI

Beginning with the processor family that was codenamed 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. As a contrast to traditional “open-chassis debug”, which requires external access to the XDP connector on a board (typically via a hardware pod or probe), DCI epitomizes “closed-chassis debug”. 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:

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”.

   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.

   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, and some third parties, sell 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 configuring at least one port to be upward-facing in a basically downward-facing connector).

   DbC has the advantage of not needing any special “pod”. It is usually just a cable from the DTS to the TS.

   DbC has even more consent issues than tool topologies using an ITP connector.

   DbC in earlier generations (i.e. DbC3, versus the newer DbC2) has some power state

debug flows that it does not support. For UEFI debug, these will usually be covered more robustly with an ITP-based tool (that is, with a hardware pod).

One key advantage of DCI access over XDP access, is the ability to "stream" trace data out the USB port versus solely to system memory. Since the transport is initialized directly out of reset (remember, this is not USB protocol; only the wires are being used) in order to facilitate debug from the reset vector, not only can you halt from reset, you can collect trace data from reset. Normally with XDP access (via an external hardware probe), the earliest you can use JTAG to collect trace data is out of system memory once it is initialized. This can be a very useful capability.

For a practical example of using DCI for debug of off-the-shelf single board computers, see our Getting Started Guide in the SourcePoint Academy.

DCI will continue to become more and more prevalent as a debug tool for Intel-based designs.

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.