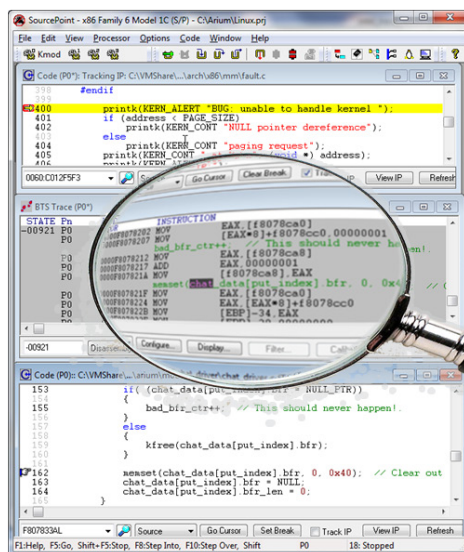


FASTER FIRMWARE DEBUG WITH INTEL® EMBEDDED TRACE TOOLS



EBOOK

BY LARRY OSBORN

By Larry Osborn



Larry Osborn, Arium Product Manager, at ASSET InterTech, has over 30 years of experience in product management, hardware/software product design and development, product delivery to the marketplace and user support. Over the years, Larry has a proven track record for identifying user needs and opportunities in the marketplace, providing innovative solutions and exceeding the expectations of users. Prior to ASSET, he has held positions with Lockheed Martin, OCD Systems, Windriver, Hewlett-Packard, Ford Aerospace and Intel® Corporation. He holds a Bachelor's Degree in Computer Science from the University of Kansas and various technical and marketing training certifications.

Table of Contents

Executive Summary	4
Trace Resources and Hardware-Assisted Debug Tools	5
How Debug Tools Apply Intel’s On-Chip Trace Resources	6
Trace as a Diagnostic Tool	7
Examples of Trace Capabilities	9
Conclusions	13
Get a Demonstration	13

Table of Figures

Figure 1: Trace uncovering a bug.	8
Figure 2: Trace discovers a corrupted stack frame.	9
Figure 3: Tracing a fault in a device driver.....	12

© 2013 ASSET InterTech, Inc.

ASSET and ScanWorks are registered trademarks while the ScanWorks logo is a trademark of ASSET InterTech, Inc. All other trade and service marks are the properties of their respective owners.

Executive Summary

Debugging software and firmware efficiently and on schedule has become increasingly more difficult as system complexity has escalated and as the sheer volume of software and firmware in multi-core, multiprocessing, high-speed systems has exploded. Advancements in computing and communications technology have had dramatic effects on the range of the typical debug investigation. Today, debugging the root causes of malfunctions and failures inevitably involves investigation and experimentation well beyond the scope of such debug exercises just a few short years ago. Methodically and quickly tracking the causes of bugs back through the interrelated web of software, firmware and hardware has become a major challenge for product manufacturers.

Fortunately, hardware-assisted debug tools are able to meet these challenges by taking advantage of the trace resources embedded by Intel® in its processors. With these debug tools developers can quickly track the suspected causes of a problem back to its real root cause. Depending on the trace features embedded in a particular Intel processor, developers can trace code execution problems within the code itself and beyond that code to the potential causes of a bug. In addition, some tools platforms can extend root cause determination considerably farther beyond code execution by applying non-intrusive debug, validation and test tools that can quickly rule out possible causes in the underlying hardware or related software modules. With a powerful set of related software and hardware debug tools at its disposal, development teams can delve more deeply and more quickly into the nature of bugs and faults that can affect the operation and performance of entire systems.

Trace Resources and Hardware-Assisted Debug Tools

All Intel processors have a hardware debug interface or port. Typically, the debug port is implemented on a customer reference board (CRB) as the eXtended Debug Port (XDP). Through the XDP, hardware-assisted debug tools are able to interface with the processor(s) to run and stop the processor, single-step through software code, and examine and deposit values (peek-and-poke) into memory and registers as part of the software/hardware debug process.

Hardware-assisted debuggers have certain advantages insofar as they are non-intrusive and they do not depend on other software resources like a functioning operating system. With hardware-assisted debug tools developers can debug code as it is executing. If the processor ‘hangs’ or stops executing code, the debug tool can still maintain control of the processor and provide visibility into system operations.

The various Intel processors offer several on-chip and platform debug resources which can be very useful in code and event trace debugging applications. Some of these Intel-specific resources are listed below. Additional explanation of the functionality and purposes of these resources is contained throughout the rest of this eBook.

- Last Branch Record (LBR) is a facility to store a relatively limited amount of trace information in on-chip resources. Technically, LBR is a flag in the DebugCtlMSR and corresponding LastBranchToIP and LastBranchFromIP MSRs as well as LastExceptionToIP and LastExceptionFromIP MSRs.
- Branch Trace Store (BTS) will store additional data to that which is gathered by LBR. BTS uses the in-system memory resources of either cache-as-RAM or system DRAM to store this additional trace data.
- Architecture Event Trace (AET) offers tracing functionality that is more selective than LBR or BTS, as well as considerably more trace data. The data gathered through AET is funneled through the processor’s XDP port and stored externally from the debug target in a connected In-Target Probe device.

How Debug Tools Apply Intel's On-Chip Trace Resources

Intel's on-chip trace resources provide the functionality to view executing code and processor events. This visibility can prove essential in a complex environment where millions of instructions are executing and many hardware events are happening every second. With Intel's trace resources, engineers can look back in time to determine exactly how the system behaved and what happened to it. This ability to capture past events and code sequences provides keen insight into system behavior. Additionally, software bugs encountered during trace can often be captured for further analysis.

Intel's embedded LBR trace instrument can capture code execution from the point of reset. Since any discontinuities while code is executing are stored in Machine Specific Register (MSR), some hardware-assisted debug tool can reconstruct the executed code by reading the 'To' and 'From' addresses, accessing memory between the specific locations and disassembling the code. The debug tool usually displays this disassembly process in a trace window in its graphical user interface (GUI). While the trace process can start at reset and does not impede any real-time performance, the size of the trace display is very shallow and typically can only contain hundreds of instructions. This may help to analyze the code that executed before a System Management Interrupt (SMI) or other exception if the debugger has set a breakpoint when the interrupt occurred. Many applications require that a greater amount of code and data are gathered for effective debug. BTS can accommodate these applications by capturing many more 'To' and 'From' events and storing the data through a cache-as-RAM (CAR) method or by storing the data in system DRAM. The debugger can derive the executed code and display it by reading either of these two memory stores and decompressing the BTS events. (Note that this is very similar to the LBR algorithm described above.) The debugger's trace window usually displays assembly and high-level languages like C/C++. Typically, BTS employs CAR as an extended data storage facility if system DRAM is not available.

Even more trace data, which amounts to additional code execution time, can be stored when system DRAM is available. The extent of time and, therefore, instructions and events that are captured is limited by the amount of memory available on the target system being debugged. With enough memory, debuggers have been able to capture and analyze an entire Linux boot. Although BTS does adversely affect system performance, CAR is much faster than storing data

in system DRAM. Both storage methods generate significant overhead. Tests have shown that BTS overhead can be anywhere from 20% to 100%, depending on the access time of RAM and the number of branches captured in the code stream.

Intel's EFI Developer Kit II (EDK II) supports hardware-assisted debuggers performing BTS trace using either the CAR or system DRAM methods. This support sets up a portion of CAR as a BTS trace storage area as soon as CAR is available. When system DRAM is available to the debugger, BTS events are recorded automatically in DRAM without user intervention. This support provides UEFI code trace visibility seamlessly shortly after the target has been reset.

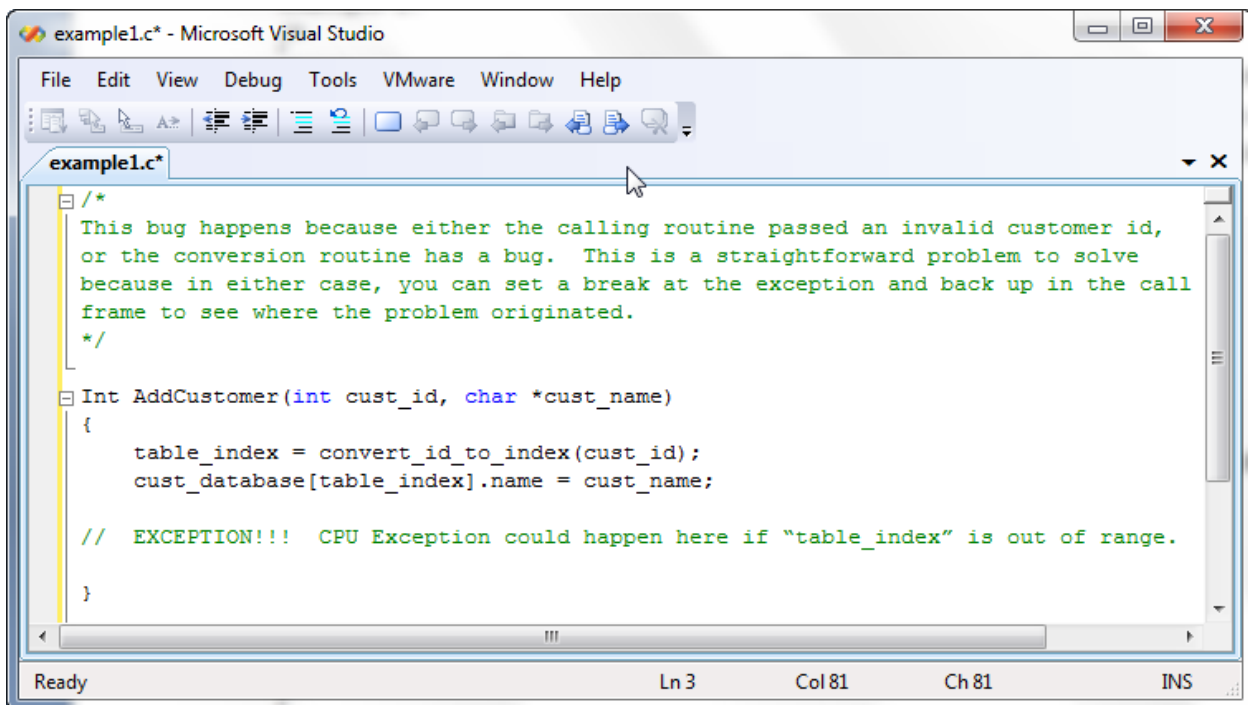
Another Intel embedded trace instrument is Architectural Event Trace (AET). This resource gives developers even greater visibility into code execution. Not only can AET capture execution history, but it can also capture events such as interrupts or I/O accesses. AET events are not stored in memory on the target, but rather in a device connected to the target by an ITP. This gives the user of the debugger the ability to analyze the trace data even when the target is not available as the result of a reset or power cycle failure, for example. If all events are captured, AET will have an effect on system performance by several orders of magnitude greater than the effect BTS produces. Fortunately, the user of the debugging tool can selectively pick and choose which events to capture through AET and thereby reduce overhead and increase the code execution time captured. In addition, debuggers can collect both BTS and AET data. BTS data can be placed in RAM while AET events are stored off-target via an ITP connection. This maximizes the advantages of both the speed of BTS and the comprehensiveness of the events captured by AET. An ITP debugger can then correlate both the BTS and AET streams using time-stamp information in the data.

AET is currently only available on client and server processors and is not available on Atom. Also, AET requires special ITP hardware which can collect and store the event data as it comes across the XDP bus.

Trace as a Diagnostic Tool

By applying the typical features of a hardware-assisted debugger most software bugs can be diagnosed in a fairly straightforward manner. These types of debuggers will set up a variety of breakpoint types, display register values and examine the entire memory space, all while

correlating full symbolic information with the underlying source code. In addition, hardware-assisted debuggers can use the stack frame to trace backwards through the code to see the execution history and calling arguments from parent functions. However, this information only pertains to the currently executing thread (Figure 1). Unfortunately, not all software bugs can be diagnosed this easily. Diagnosing bugs usually follows the 80/20 rule. That is, the person doing the debugging will spend 80% of his or her time on 20% of the bugs. The hard-to-diagnose bugs typically happen in one context, but are detected in another. Usually, there is an indeterminate amount of time between occurrence of the bug and the effects it has on system operation or performance. For example, an interrupt routine might corrupt memory that just happens to be in the stack area of an unrelated thread. The thread may do a bounds check on an array index (Figure 2) only to have the value of that index corrupted before it is referenced. Determining where the corruption occurs with traditional debug techniques would be nearly impossible because the corruption has nothing to do with the current context of the code. The actual cause of the bug could have happened millions of instructions earlier in some totally unrelated piece of code. These are the types of bugs where trace can provide the most benefit.



```
example1.c* - Microsoft Visual Studio
File Edit View Debug Tools VMware Window Help
example1.c*
/*
This bug happens because either the calling routine passed an invalid customer id,
or the conversion routine has a bug. This is a straightforward problem to solve
because in either case, you can set a break at the exception and back up in the call
frame to see where the problem originated.
*/
Int AddCustomer(int cust_id, char *cust_name)
{
    table_index = convert_id_to_index(cust_id);
    cust_database[table_index].name = cust_name;

    // EXCEPTION!!! CPU Exception could happen here if "table_index" is out of range.
}
Ready Ln 3 Col 81 Ch 81 INS
```

Figure 1: Trace uncovering a bug.

In some instances, trace debug capabilities may be the only way to debug a problem. For example, an interrupt routine might suddenly and inexplicably execute in the middle of a function (Figure 2). Regardless of the type of embedded trace instrument deployed by the tool (LBR, BTS or AET), the debugger can easily correlate the code execution history against source code.

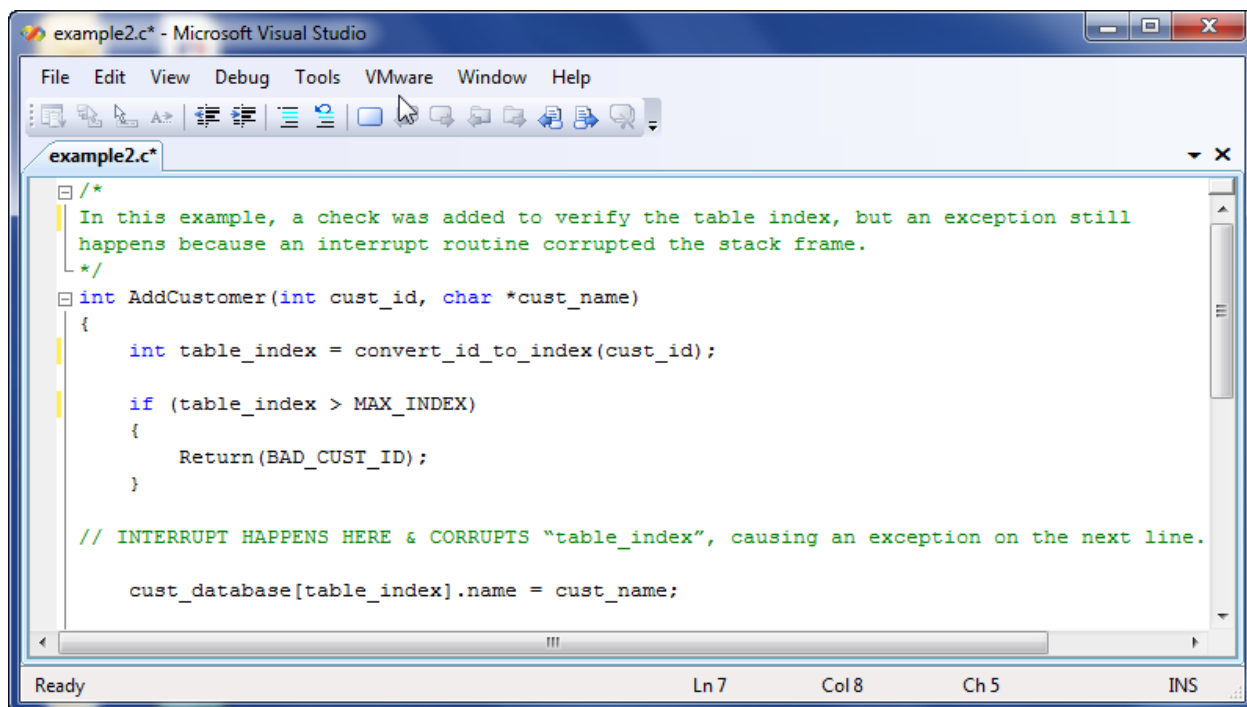


Figure 2: Trace discovers a corrupted stack frame.

Examples of Trace Capabilities

The following are several examples of specific problem areas which demonstrate how trace functions can quickly diagnose root causes.

- Interrupt Behavior

A developer is working on a driver for an Ethernet device and wants to understand the nature and timing of the interrupts. From the device's data book and based on the CPU's clock rate and the data rate of the interface, he is able to calculate the interrupt frequency on both the receive (rx) and transmit (tx). With this information he can formulate the design requirements and parameters for the device driver. Once the driver is complete, he must begin testing and fixing bugs that arise. The problem, though, is the non-deterministic nature of the interrupt patterns. The

interrupts encountered by the Ethernet device will depend on the behavior of devices on the other end of the transmission line. In addition, other interrupts will be encountered in the system, such as USB, timers, disk accesses and other sorts of interrupts. The developer may believe he has a debugged working device driver, only to find a major bug once the system has been released to the field.

Trace is a very useful tool in analyzing these types of issues, primarily because it is empirical. It proves (or disproves) any timing assumptions that were made based on the information contained in the device's data book. In addition, trace will identify any anomalies in chip behavior. A typical debug process using trace tools might happen this way. First, BTS could be used to make certain measurements. For example, the exact number of instructions in an ISR routine might be needed. In addition, the interval between the transmission of a data packet and when the transmit-complete interrupt is received could also be measured. Next, the interrupt behavior and patterns of the system in a variety of use cases such as simultaneous USB and Ethernet activity might be analyzed. In this last case, though, BTS would probably not be the tool of choice. BTS functions well for short periods of time, but the number of interrupt occurrences it captures would probably be inadequate. AET would be a better choice since it can be set to selectively record only the interrupt events. This would avoid filling up the memory storage area with unneeded and superfluous branch history. Depending on the amount of memory available in the ITP-connected device, an interrupt history from a few seconds to several minutes could be captured. This is not to imply that this type of trace capture would necessarily identify bugs in the device driver, but it would certainly provide a more comprehensive picture of the system's behavior than is available through other methods. At the very least, this type of trace analysis would either confirm or refute the expected behavior.

- Linux System Debugging

Trace capabilities can also be applied quite effectively to debugging embedded Linux. On a workstation, the debug tools would typically run on the same host that is being debugged, but this is not the case with embedded Linux. The debug agent (or server) would typically run on the target system being debugged, while the debugger's user interface (UI) would run on a host workstation which is connected to the embedded target over Ethernet, through a JTAG port or in some other manner. This configuration is often referred to as cross debugging. Historically,

debugging embedded Linux has involved a two-step approach. One set of tools would debug the user space while another would debug the Linux kernel. GDB (GNU Debugger) is the dominant user space debugging tool. It contains a server (gdbserver) that runs on the target and a host program (gdbclient) that functions as the user interface. GDB works well for process-level or thread-level debugging as long as the operating system is functioning properly and the Ethernet connection is reliable. The built-in Linux kernel debugger (KDB) and the Linux source level debugger (KGDB) are common tools for kernel debugging, but they can be problematic in a cross-connected environment. More significantly though, they rely on other software to perform correctly. For KDB and KGDB to function effectively, a stable kernel and Ethernet or serial connection are needed. This can be troublesome since a stable kernel and properly working system would not require debugging in the first place. Something must be malfunctioning or debugging would be taking place. Hardware-assisted tools that don't rely on any software running on the target can provide access to the LBR/BTS/AET embedded trace instruments and the data that is needed to identify the most difficult bugs.

At this point, a closer analysis of a bug involving a Linux device driver that corrupts memory would be helpful. In this example, the device driver is called “chat” and it exchanges data between 2 Linux processes. While testing the driver, the Linux kernel crashes and the following error message is displayed on the serial console: “BUG: unable to handle kernel NULL pointer dereference”.

After searching through the Linux source code, the error string is found at line 400 in the module “fault.c” (Figure 3). The debugging engineer sets a breakpoint at that line and runs the test again. The firmware executes to the break point, but the cause of the bug is not found because the root cause is actually in a different context. Unfortunately, because of the context switch, the debugger cannot track backwards through the call stack. This is where BTS trace capabilities can provide significant value. Since the premise for this analysis has been that the problem is related to the “chat” driver, a trace debugger will be able to search backwards through the BTS data for the text “chat” and find the last code location where the driver was invoked. Figure 3 shows the breakpoint in the fault module in the top pane of the debugger's GUI. This is where the CPU has stopped. The middle pane displays trace data with assembly instructions and interspersed “c” code. This pane shows that the last reference to “chat” was 921 branches and approximately

20,000 instructions prior to the current program instruction counter. The bottom pane contains the “c” source code correlated against the trace data. This window tracks directly with the cursor location in the trace data. The bottom pane shows the area of the device driver that last executed before the exception occurred. Note that in the buffer overrun condition at line 153 in the chat driver, the developer mistakenly used one equal sign (“=”) instead of two (“==”). This resulted in a null pointer being assigned to a location that was used as the destination address in “memset”. Effectively, this wiped out memory starting at location zero and caused the exception in the kernel.

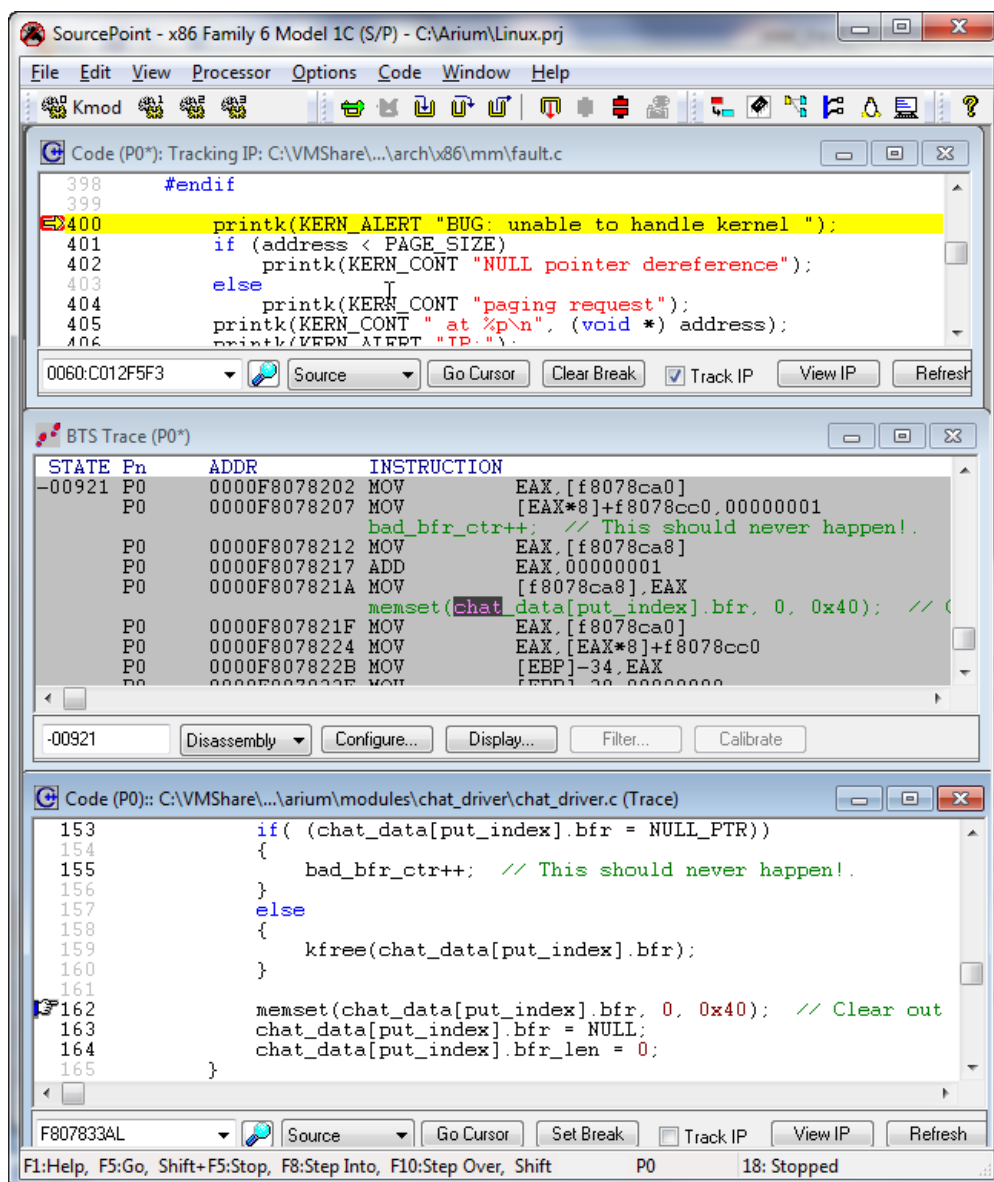


Figure 3: Tracing a fault in a device driver.

Conclusions

Using Intel's embedded trace resources, LBR, BTS, and AET, in conjunction with hardware-assisted debugging tools can be quite effective. In addition, automated deployment by hardware-assisted debuggers of CAR and system DRAM data storage methods with BTS trace resources is currently supported by Intel's EFI Developer Kit II (EDK II).

Some of the advantages and disadvantages of each embedded trace instrument are as follows:

- LBR has no overhead, but is very shallow (4 – 16 branch locations, depending on the CPU). Trace data is available immediately following reset.
- BTS is much deeper, but it has an impact on CPU performance and requires on-board RAM. Trace data is available as soon as CAR is initialized.
- AET requires special ITP hardware and is not available on all CPU architectures. It has the advantage of storing trace data off of the target board.

Engineers involved with BIOS/UEFI, device driver, boot loader and OS porting projects can benefit from hardware-assisted debug tools. Debugging intermittent failures and system hangs, as well as early firmware development are use cases that can take advantage of a debug interface like Intel's XDP.

Get a Demonstration

Advanced trace features are a powerful tool for engineers who want to find bugs fast and quickly develop high-quality code. To learn more about hardware-assisted debug tools and arrange a demonstration.



Register Here!