Blackhawk™ Emulation Whitepaper

Using XDS560™ Trace to Expose the Toughest Real-time Bugs

Picking up where traditional DSP debugging methods stop

Digital Signal Processor (DSP) application programs are becoming more and more complex and developers are still spending much of their time in the debug phase. The XDS560 Trace (XDS560T) system is a powerful new debug capability for Texas Instruments (TI) DSPs that gives developers visibility beyond the limits of traditional debug methods. It will save time and money by finding complex, intermittent, context-sensitive real-time bugs.

Figure 1 - XDS560 Trace System

DSP applications that depend on operating systems to handle multiple processes and interfaces are very complex. Too much valuable time is often spent tracking down hidden bugs that occur intermittently in these codes sets. The XDS560T system is a completely non-intrusive debug tool that can detect scheduling issues, intermittent glitches, false interrupts (and more) without stopping the processor.

Where does traditional debug stop?

Traditional debug methods using a JTAG emulator or simulator is still a good solution that can catch 95% (or more) of your basic bugs. However, to solve difficult problems such as runaway code, system crashes, or false interrupts that happen only when your application is running in real time, you need a different solution.

Example Scenario

The "***How did I get here?*** bug? is a common problem that has probably haunted everyone. This is where your code is running properly, then out of nowhere, your system crashes or enters a routine that should never have been called.

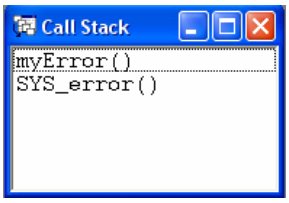

Figure 2 - Call Stack Window

In this example, we have a DSP/BIOS program with multiple tasks and are lucky enough to find out where our code stops, when this bug appears. This location, which we found using normal emulation debug, is in the default error handler for a failed DSP/BIOS system call. So we reconfigured DSP/BIOS to call our custom error handler, *myError()*, instead and set a breakpoint in our routine (*Figure 3*) to see the call stack that led up to this error. Then we rebuilt, reloaded and executed the code waiting for it to stop at this location (breakpoint).

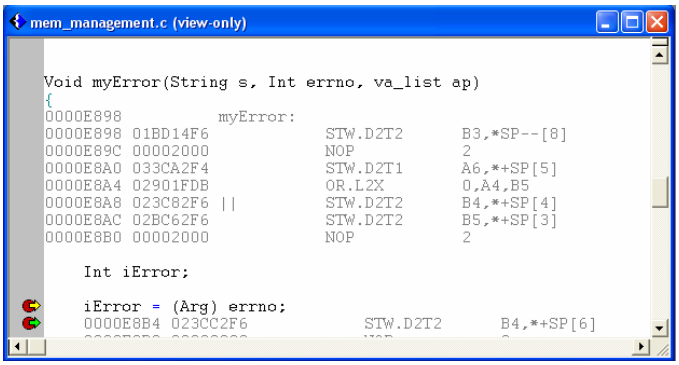

Figure 3 – Breakpoint in `myError` Handler

When the breakpoint is reached, CCS will halt the CPU and we can view the call stack (*Figure 2*) looking for the sequence leading up to the error condition. But as you can see, there is no trail or useful information because our application is part of an operating system.

All that the call stack shows is the function, *SYS_error()*, which is the default error handler for DSP/BIOS system calls, and *myError()*, the handler we set for failed DSP/BIOS system calls. We do not know which task or interrupt caused this or what DSP/BIOS system call failed to generate the call to our error handler. So how can we get past this traditional debug dead end?

Using XDS560 Trace to get you past the dead end

XDS560T can pick up where basic debugging left off and show you the sequence of events that caused a bug. Referring back to the previous example, we now have the XDS560T system connected to the same target hardware and application code using the same PC running CCStudio v3.3.

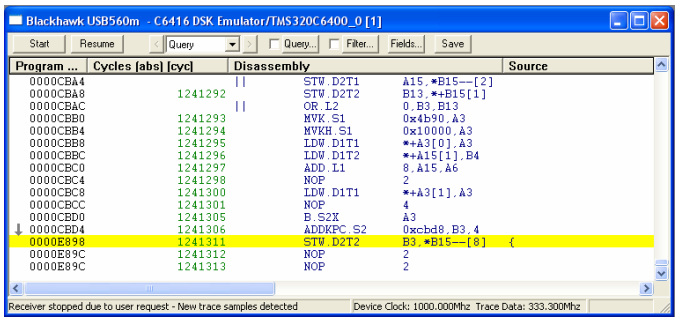

Figure 4 – Trace Display of `myError` Breakpoint

You may be wondering at this point, what is involved to get your code ready to use with the XDS560T? ***Nothing***! Just reload your OUT file.

Once your code is reloaded, you can begin to set up the trace controls to capture the trace information. The controls for XDS560T are built into CCStudio and are easily accessed from the TOOLS menu of the main menu bar (see Figure 5). This menu provides access to the trace system's controls, setup, and display.

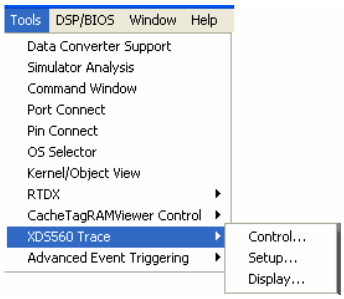

Figure 5 - Trace Menu Options

The Trace System Control dialog (Figure 6) allows you define settings on how trace information is collected, such as using a circular buffer or stopping collection on a breakpoint. This is the mode we will be using.

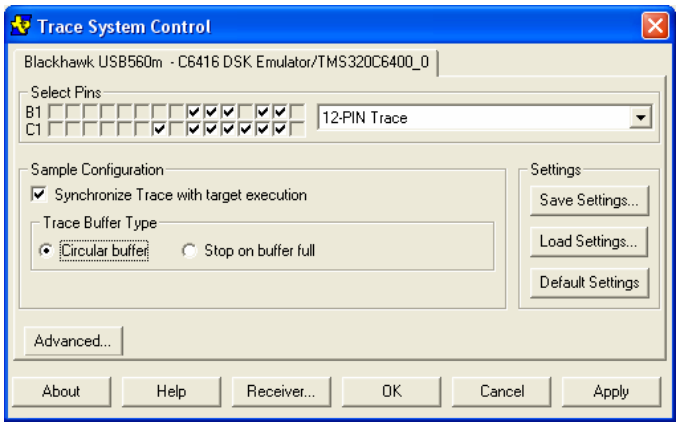

Figure 6 - Trace System Control Dialog

The Event Analysis setup dialog (Figure 7) defines how you start and stop trace collection. Using the advanced event triggering (AET) feature, the event analysis window allows you to configure when to start and stop trace collection and what data to capture.

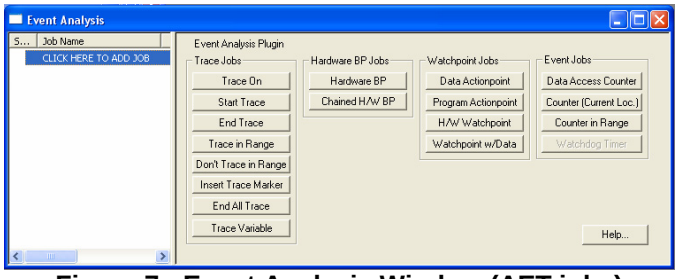

Figure 7 - Event Analysis Window (AET jobs)

For this example, we will configure a "Trace On" job and just save the program counter information. We can change that later to get more details, but right now we want to see the path resulting in the error. It is not necessary to specify an "End All Trace" job because we *synchronized trace with target execution* so that trace data collection will stop when we hit our breakpoint.

Now that we have the code reloaded and trace configured, we can run the application and wait for the breakpoint to be reached. When this happens, CCStudio will halt the processor and the collected trace data will be transferred from the trace module to the trace display window. Scrolling to the end of the data will show you the breakpoint location (Figure 4).

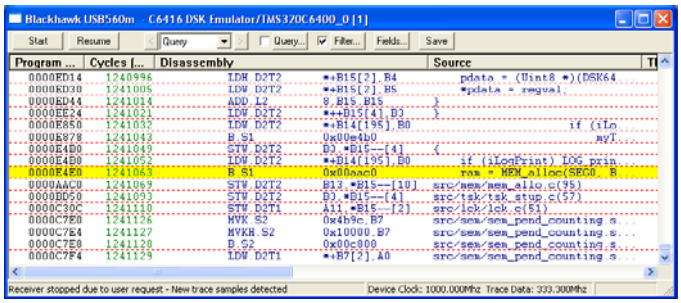

Figure 8 – Last system call by application

Then, scrolling backwards (up) through the data, you can see all the events that lead to this error. In Figure 8 we set a filter to display only source code (no assembly) to find the last line of our code that called a DSP/BIOS system call. In this example the failing system call is a memory function (`MEM_alloc`). And after some more analysis, we determined that this failure was caused by not freeing previously-allocated memory.

So, you can see the extent of information revealed in the trace display, not to mention over 1.2 million instructions (cycles column), by just tracing the program counter.

Can I use it for more than just debug?

Yes. Now that we found and fixed our bug, and the application is running, we can continue to benefit from trace using its profiling capability. By configuring trace jobs to collect specific trace data, such as from a particular algorithm, we can export this information to analyze (profile) offline. The offline data can now be viewed or processed using readily available tools (i.e. Microsoft Excel, Perl scripts) to look for delays, CPU stalls, cache hits, etc., which may be slowing your code. Profiling is a very powerful feature that can enable faster, more efficient applications

What is supported today?

As of this writing, TI has more than seventeen catalog devices and three development boards that are XDS560 Trace-ready. More devices and development boards are in the pipeline. You can verify that your device has trace capability by referring the TI website or part datasheet. There is a column or field listing if it is *trace enabled*.

Adapters are available from Blackhawk to interface legacy emulators with 14 and 20 pin JTAG connections to the 60-pin trace header for normal debug. So there is no reason to avoid placing the 60-pin header on your target board for trace capability.

Where do I go now?

The XDS560T ships with the new TI trace module and a Blackhawk XDS560 High-Speed USB 2.0 Emulator (Figure 1), plus all required cables and a driver CD ROM supporting installation on Windows 2000/XP/Vista for operation with Code Composer Studio v3.3 IDE (CCStudio).

In addition to the XDS560T package, you will need: CCStudio v3.3 or later and a target board with TI 60-pin JTAG connector (*see TI technical document SPRU655 for more information on the 60-pin emulation header*) interfaced to a TI DSP device that supports trace. Development boards that meet these requirements are available today for under $500, so getting started is easy.

Blackhawk is offering this advanced debug technology directly through TI distribution under part number: TMDSEMU560T for $9995 as well as through Blackhawk's world-wide reseller network under part number: BH-USB-560T. Please visit the Blackhawk web site, http://www.blackhawk-dsp.com, for the latest information, examples, and nearest reseller.

Written by: Andrew Ferrari, EWA Technologies, Inc.; John Fasso, EWA Technologies, Inc.