XDS560 Trace
Customers Win with eXpressDSP™ Software and Development Tools

- **TMS320DSP Algorithm Standard**
- **Robust, qualified Third Party solutions**
- **DSP/BIOS kernel with drivers and CSL**

- **CCStudio point and click IDE**
- **Open IDE and robust project manager**
- **Best in class C/C++ Compiler**

- **Real-time analysis**
- **H/W & S/W BP**
- **Advanced Event Triggering**
- **Fast Sims/Rewind**
- **Watch Windows**
- **Trace**

- **Rich optimization options and tooling**
- **Tuning Dashboard**
- **Profiling**
- **Cache Visualization and Analysis**
- **Power Management**

---

**Accelerates time to market**

**Enables differentiation**

**Reduces system cost**
XDS560 TRACE: Exposing the Toughest Real-time Bugs

Find previously “invisible” complex, intermittent, context-sensitive real-time bugs

- Detect scheduling issues, intermittent glitches, false interrupts and more without stopping the processor
- Fully integrated with CCStudio Advanced Event Triggering

Fine tune code performance and cache optimization of complex switch intensive multi-channel applications

- Real-time code and event profiling
- Fast and accurate code analysis with profiling, cache view and code coverage
- Support available today on: C641x, DM64x, C6455

Time Savings

Cost Savings
Debug Options Today – Catch 95% of Bugs

Effective for solving basic software issues

Better view inside the “chip”
Finding the Invisible Bugs

**Bug Name:** “Invisisect” from the Latin roots: L *invīsibilis* and L *insectum*

**Common name:** Pain in the Butt

**Phylum:** Anthropoda

**Class:** Insecta

**Description:** Virtually invisible to the naked eye, typically reveals itself under rare, non-predictable conditions

**Habitat:** Hides in cache routines, interrupt algorithms

**Type of Damage:** System intermittent glitches, race conditions between events, unpredictable system crashes

**Recommended Pest Control:** XDS560 Trace
Finding the Invisible Bugs

Difficult software problems
- Intermittent real time glitches
- Race conditions between events
- Crashes (Stack overflow, etc)
- Runaway code
- False interrupts

What is REALLY going on here?

And HERE when the problem occurs

DSP Chip
- DSP Core
  - Program Cache
  - Data Cache
  - L2 Cache
  - DSP Peripherals & Bus
XDS560 Trace Introduction

- Debug unit INSIDE DSP
  - Data collection
  - Complex event triggers

- Separate of DSP execution
  - Does not interfere
  - “Non-intrusive”

- Data is piped out a high-speed interface

- CCStudio IDE used to analyze data

- DSP Core
- Program Cache
- Data Cache
- L2 Cache
- DSP Peripherals & Bus
- Data Collection
- Advanced Event Trigger
- TRACE
How Did I Get Here?

“How did I get here?”

Trace will capture continuously through SWI, ISR1, ISR2, etc. until it is stopped.

Trace AET Trigger:
PC=0xISR2_ADDR

Undesired interrupt
How Trace Works

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00088</td>
<td>MVK.L2 1,B4</td>
</tr>
<tr>
<td>0x00089</td>
<td>B.S1 _SWI</td>
</tr>
<tr>
<td>0x00090</td>
<td>SUB.L2 B15,8,B15</td>
</tr>
<tr>
<td>0x0489C</td>
<td>MVKH.S2 0x0000,B6</td>
</tr>
<tr>
<td>0x048A0</td>
<td>ADD.L2 8,B15,B15 ; _ISR2</td>
</tr>
<tr>
<td>0x048A4</td>
<td>STH.D2T2 B4,*+B6[B5]</td>
</tr>
<tr>
<td>0x048A8</td>
<td>NOP 2</td>
</tr>
</tbody>
</table>

Trigger Points

- = 0x089 TRC ON
- = 0x48A0 TRC OFF
- = 0x0xx N/A

Trigger point set to turn Trace ON when PC = x089
PC ≠ match no action taken
How Trace Works

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00088</td>
<td>MVK.L2</td>
</tr>
<tr>
<td>0x00089</td>
<td>B.S1</td>
</tr>
<tr>
<td>0x00090</td>
<td>SUB.L2</td>
</tr>
<tr>
<td></td>
<td>0x089 AMOV</td>
</tr>
<tr>
<td></td>
<td>0x089 B.S1</td>
</tr>
<tr>
<td>0x0489C</td>
<td>MVK.H.S2</td>
</tr>
<tr>
<td>0x048A0</td>
<td>ADD.L2</td>
</tr>
<tr>
<td>0x048A4</td>
<td>STH.D2T2</td>
</tr>
<tr>
<td>0x048A8</td>
<td>NOP</td>
</tr>
</tbody>
</table>

--

Trigger Points

- = 0x089 TRC ON
- = 0x48A0 TRC OFF
- = 0x0xx N/A

Trigger generator enables trace
Trigger match with current PC

Advanced Event Analysis Unit

TRACE – Data Collection

Debug Port

XDS560 Trace

DSP Core

DSP Chip

Addr 0x089

Data B.S1
How Trace Works

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00088</td>
<td>MVK.L2 1,B4</td>
</tr>
<tr>
<td>0x00089</td>
<td>B.S1 _SWI</td>
</tr>
<tr>
<td>0x00090</td>
<td>SUB.L2 B15,8,B15</td>
</tr>
<tr>
<td>0x0489C</td>
<td>MVKH.S2 0x0000,B6</td>
</tr>
<tr>
<td>0x048A0</td>
<td>ADD.L2 8,B15,B15 ; _ISR2</td>
</tr>
<tr>
<td>0x048A4</td>
<td>STH.D2T2 B4,*+B6[B5]</td>
</tr>
<tr>
<td>0x048A8</td>
<td>NOP 2</td>
</tr>
</tbody>
</table>

Trace continues exporting trace.
Trigger setup to turn trace OFF when PC=0x48A0
PC ≠ match no action taken

![Diagram of DSP Chip, TRACE - Data Collection (Enabled), Debug Port, XDS560 Trace, Trigger Points: 0x089 TRC ON, 0x48A0 TRC OFF, 0x0xx N/A]
How Trace Works

<table>
<thead>
<tr>
<th>Address</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00088</td>
<td>MVK.L2</td>
</tr>
<tr>
<td>0x00089</td>
<td>B.S1</td>
</tr>
<tr>
<td>0x00090</td>
<td>SUB.L2</td>
</tr>
<tr>
<td>~~~</td>
<td>~~~~~</td>
</tr>
<tr>
<td>0x0489C</td>
<td>MVKH.S2</td>
</tr>
<tr>
<td>0x048A0</td>
<td>ADD.L2</td>
</tr>
<tr>
<td>0x048A4</td>
<td>STH.D2T2</td>
</tr>
<tr>
<td>0x048A8</td>
<td>NOP</td>
</tr>
</tbody>
</table>

Address | Disassembly        |
---------|---------------------|
0x00088  | MVK.L2 1, B4        |
0x00089  | B.S1 _SWI           |
0x00090  | SUB.L2 B15, 8, B15  |
~~~      | ~~~~~               |
0x0489C  | MVKH.S2 0x0000, B6  |
0x048A0  | ADD.L2 8, B15, B15 ; _ISR2 |
0x048A4  | STH.D2T2 B4,*+B6[B5]|
0x048A8  | NOP 2               |

Advanced Event Analysis Unit

Trigger Points

- 0x089 TRC ON
- 0x0xx TRC OFF
- 0x0xx N/A

Trigger Generator

Trigger generator disables trace

Trigger match with current PC
Trace Display

You will now have a trace display with source and disassembly data. Use "Query" to highlight lines that you are interested in. Use the "Fields" button to choose what you want to see and in what order. If you click on the cycles tab, you can choose various display options like detail timestamps, etc.

Program Address & Data

Use "Query" to highlight lines that you are interested in.

Disassembly

Use the "Fields" button to choose what you want to see and in what order.

Parallel cycle

Correlated Source Code

Tagged samples (gray) (ex: tag OR.L2)

CPU Stall

Timestamp

Disassembly

You will now have a trace display with source and disassembly data.
This window now just shows source because you filtered it.

If you move this line up and down with the cursor keys, the source code in the below window will move along with you.

This one shows the file you are in so you can see the context of the “C” source code line. The yellow-bars allow you to see the synchronization.
What Kind of Data Can TRACE Collect?

Choose from:
- Read Address
- Read Data
- Write Address
- Write Data
- Correlation with Program Address

Query to highlight address or data. Ex: 0x47F0

Optionally correlate the data access (read/write) to the Program Counter
XDS560 Trace Product Hardware Details

Host PC & CCStudio

XDS560 Emulator

Target Board

XDS560 Trace Product

Trace Cable

- C6416T DSK
- DM642 EVM
- C6455 EVM

TMDSEMU560T Blackhawk XDS560 USB Emulator and Trace Unit
XDS560 TRACE: Exposing the Toughest Real-time Bugs

Find previously “invisible” complex, intermittent, context-sensitive real-time bugs

- Detect scheduling issues, intermittent glitches, false interrupts and more without stopping the processor
- Fully integrated with CCStudio Advanced Event Triggering

Fine tune code performance and cache optimization of complex switch intensive multi-channel applications

- Real-time code and event profiling
- Fast and accurate code analysis with profiling, cache view and code coverage
- Support available today on: C641x, DM64x, C6455

Time Savings
Cost Savings
What Can I Do With Trace?

◆ Catching Problems
  ▪ Runaway Code
  ▪ Intermittent real-time glitches
  ▪ Bogus interrupts
  ▪ Race conditions between events
  ▪ Stack overflow context

◆ Profiling Code
  ▪ Fast, accurate code analysis
  ▪ Real-time code and event profiling

◆ Code Coverage
  ▪ Real-time code coverage testing

Most importantly it’s good for the really, really hard problems that show up the week or two before a release (last 5% that take 50% of your debug time)
What can I do with Trace which I cannot do with other tools?

◆ Show you your true execution path through code non-intrusively.
  - This will help you figure out whether an interrupt ran unexpectedly, whether a function path ran as expected, etc.
◆ Gather cycle information non-intrusively.
◆ Give you visibility into what the CPU is reading and writing to memory, what the CPU is seeing (stalled because of cache miss?)
  - You can get the data read/write address, data value, and size, correlated to the program line where it occurs
◆ Show you what is happening inside SPLOOP
◆ Get information at a line level granularity.
  - Cycles, etc.
Advanced Information
Trace Control Panel

• Checked: When CCS has the CPU HALTED, update the display. When CCS runs/steps the CPU, start the trace.
• Un-Checked: Starting and stopping are controlled from the Trace Display.

• “Stop on buffer full” Start collecting, stop when the buffer fills up.
• “Circular Buffer Mode” Keep collecting until either a trigger stops it, or you manually tell it to stop.

Hint: When you click Apply, wait until the trace display has gone back to a blank “grey”. It will take a couple of seconds as the trace system is re-programmed.
Export Settings

• Checked: Stall the CPU to make sure Trace data gets out. This ensures there are no data gaps, but, the CPU gets stalled.
• Un-Checked: Do NOT stall the CPU. In this mode, you must be careful about what you trace, as you could run out of bandwidth, so there could be gaps in the data.

• Select what size of trace packet to send.

• Keep collecting trace if the CPU is halted, but you have some real-time interrupts running.
• How long you were halted
• Global trace stream enable / disable
Trace Capture Buffer: Trace ON

“Snapshot”

Ex: Capture from this point on

Trace Capture Buffer

ISR

SWI

IDL

Trace ON: PC=0xISR_ADDR
Trace Capture Buffer: Trace Range

“Capture only this area of code.”

Ex: Trace only when in ISR1 (start in ISR1, end at end of ISR1 reached)

Trace Trigger Range: 0xISR1_start_addr < PC < 0xISR1_end_addr

- Trace starts with beginning of ISR1 and stops at end of ISR1.
- ISR2, which pre-empted ISR, is also captured.
- Un-desired/unexpected pre-emption visible.

Undesired Pre-emption
Start Trace / End Trace

“Capture starting here, and end here.”

Ex: Trace everything from one location to the next, including child functions.

Trace Capture Buffer

- Trace starts at the beginning of ISR1 (0xISR1_start_addr) and ends at end of ISR2 (0xISR2_end_addr)
- SWI is also captured as it is run in between the trigger points.

Trace Trigger Range:

0xISR1_start_addr < PC < 0xISR2_end_addr
Circular Buffer

"Where was I before the bug?" or "How did I get here?"
Ex: ISR2 should never happen. If it does, where was the code before?

Trace Capture Buffer

ISR2
ISR1
SWI
IDL

Trace will capture continuously through SWI, ISR1, ISR2, etc. until it is stopped.

Undesired interrupt

Trace AET Trigger:
PC=0xISR2_ADDR
When to use “Stop on Buffer Full” versus “Circular Buffer” mode

◆ Stop on Buffer Full
  ■ “One-Shot” Trace capture
  ■ “Capture from the time I tell you to start to until buffer is full”
  ■ Good way to make sure you can see the earliest trace events
  ■ Good for profiling, because start point is known.
  ■ Good for looking “forward” in time.
  ■ Will cause “Data not verified” at end of buffer.

◆ Circular Buffer
  ■ Continuous Trace capture
  ■ “Capture trace until I tell you to STOP”
  ■ Older information will be overwritten with new trace information.
  ■ Good for looking “backward” in time.
  ■ Good for debugging, because stop point is known.
  ■ Will cause a single “Incomplete data” at start of buffer.
Basic Trace Tutorial
Setting up for a simple Trace.

- **Goal:**
  - Get a basic trace out of the system.

- **Instructions:**
  - Load the “sillyprog” project and program into the device.

- **Notes:**
  - While we are in Early Adopter for trace, reset your emulator before you connect.
  - This program is located in the “CCS\my projects” directory.
Activate UBM by going to Debug → Breakpoints
• Create a new “Trace” event
- Standard Trace has been selected. Click + to configure the information trace will collect.
• Select the types of trace that you want to get.
• In this case, Program address and timestamp are selected.
- After submit the job, the trace ON is now enabled.
- If this is the first trace event setup, then the XDS560 Trace system will start up.
• Create a new Trace Event. We will use this one to stop the XDS560 Trace collection upon a data write to a data variable.
• Change the “Action” to select “End All Trace”
• This will stop all trace collection.
• In “Trigger Type” select “Data Memory”.
• We want to stop all tracing if any application code writes to the Data_4 variable.
• Set the “Location Type” to be “Range” and set the start and end locations to be the start and end address for the “Data_4” variable.
• If the location range was the top of the stack, then this technique would allow us to examine trace on any stack overflow.
• Remember to click “Enable” AND “Submit”
• After running the trace system, and looking at the end of the trace buffer, we can see that we have stopped tracing upon the first data write to the Data_4 variable.
Now, go and run the target. Stop after a few seconds.

• The trace system starts up ready to collect information from the target.

• This button allows you to start / stop the trace system. When Stopped, it will not collect trace information.

• This will allow you to get to the advanced controls.
Results after Running Target

- You will now have a trace display with source and disassembly data.
- Use the "Fields" button to choose what you want to see and in what order.
- Use "Query" to highlight lines that you are interested in.
- If you click on the cycles tab, you can choose various display options like detail timestamps, etc.
- You will now have a trace display with source and disassembly data.
Filtering

Experiment with the filters. This is critical to understanding the information collected by XDS560 Trace.

*If you filter on a string that is blank, it will show just source code.*
The Trace Display

This window now just shows source because you filtered it.

If you move this line up and down with the cursor keys, the source code in the below window will move along with you.

This one shows the file you are in so you can see the context of the “C” source code line. The yellow-bars allow you to see the synchronization.
Tutorial: Configuring Event Trace

“Help me optimize my cache operations by showing me where I have cache misses in my code.”
If the trace type of “Event” is selected instead of “Standard”, then the trace system will be configured for “Event Tracing”. This will allow the user to select from several global categories: Stall, Memory, System, and External. Note that event types are determined by the device being used. This information should be in the datasheet.

In this case, I have chosen to view all types of CPU stalls as well as any type of L1P cache stall. Event 1 is setup for CPU stalls, and Event 2 is setup for L1P stalls. Thus, in the trace display, if there are any “1” in the stall event column, they will be CPU stalls. If there are any “2 in the stall event column, they will be L1P stalls. This information helps a user to determine the type of stall on a per-line basis.
From the trace output, we can see that we have CPU stalls and not L1P stalls. This is because the Stall Cycles display shows “1” instead of “2.” Note that the meaning of “1” and “2” is wholly dependent on how you configure the events.
Data Trace
Data Trace

Choose from:
- Read Address
- Read Data
- Write Address
- Write Data
- Correlation with Program Address

Note that Trace Bandwidth is limited. Can choose to enable Stall Mode if needed.

Optionally correlate the data access (read/write) to the Program Counter

Query to highlight address or data.
Ex: 0x47F0
ETB

Embedded Trace Buffer
ETB Basics

◆ What is ETB?
  ■ ETB stands for “Embedded Trace Buffer”
  ■ ETB is an on-chip memory buffer where the trace information is stored.
◆ Where is ETB available?
  ■ ETB is only available on certain devices.
◆ How big is the ETB buffer?
  ■ The size depends on the chip designer. Typically, it will be between 2-8K.
◆ Is an XDS560 Trace Hardware unit needed to use the ETB?
  ■ No
◆ Can ETB be used concurrently with the XDS560 Trace
  ■ Yes, but there are limitations
    ✷ A device with multiple cores has several operating options:
      ➢ All cores use their individual ETB
      ➢ One user selected core outputs Trace to the XDS560 Trace, while others use their individual ETB.
      ➢ One user selected core outputs Trace to the XDS560 Trace.
Using ETB

Select the type of Trace receiver

Additional Tabs for each core

The Trace control now shows ETB as the type of Trace Receiver
Overlays
Software Overlays

- Software Overlays have software modules share program memory
  - Can improve performance
- Mechanics:
  - The software modules will both run from the same physical memory
  - The software modules are both loaded into different physical memory locations
  - The user or application framework will copy the code from the load region into the run region
- XDS560 Trace
  - Supports automatic detection of the software module used in the overlay.
    - Manual override is available
  - Supports manual overlay software module selection
Software overlay

When Overlays are used, the load and run addresses are unique.

Trace messages indicate start of overlay section.

User can override the automatic overlay selection.
Software Overlays: Manual

The User can manually select the overlay that was used in the program.