Welcome to the Real-time Trace for Keystone Devices training module.

This module covers the basics of Trace debugging and hands-on labs specific for Keystone Devices (C66x)
Trace is very useful to perform advanced debugging on embedded systems, as it expands the basic JTAG debugging by performing real-time data gathering of several aspects of the embedded processor.

In its most traditional form, Trace captures and stores all the instructions executed by CPU in real time, thus allowing finding complex or intermittent bugs in the system. This is called Core or Instruction Trace.

Certain TI embedded processors also can capture system events in real time such as memory interface throughput and power domain status, thus allowing finding system wide problems during runtime. This is called System Trace.
Multiple uses and scenarios can have Trace as a helping hand to debug problems in the system.

For example, the ability of Core Trace to “see” the entire past execution history allows finding places where the code behaves erratically or took an unexpected branch.

Also, if the system is running slow or is returning incorrect or corrupt data, it is possible to use a combination of Core and System Trace to track what can be happening.

In certain devices it is also possible to evaluate if cores are actually powered down and not hung waiting for interrupts or other events.
Expanding on the concept of Core Trace, the idea behind it is pretty straightforward: simply capture all the assembly instructions that ever get executed by the CPU and send them to the host PC for analysis. These are stored together with timestamps.

Once this data is available, CCS can correlate the assembly instructions with the source code and thus allow looking at the code execution more easily. In addition to that it can also perform a multitude of other operations. The most relevant are: code coverage analysis, which means finding out which routines were actually executed, and profiling, which means knowing how many times and for how long each instruction and routine executed.

However, one important detail defines its availability: since the execution speed of modern processors can reach billions of instructions per second, it is impossible to gather all this information without special hardware and some buffering between the device and the host PC. That is the reason why core trace is not available in all devices, and for the ones who have this feature there are two implementations with different levels of complexity: Embedded Trace Buffer, or ETB, and pin trace using an external emulator pod.
Embedded Trace Buffer or ETB is a small memory area that is tightly coupled to the processing core of the device, and contains a copy of all the instructions executed by it.

Being small, this memory is able to capture a smaller sample of instructions that pass through the core, and it only operates in circular mode, which means that old instructions are overridden by newer ones.

However, the largest advantage is that it allows performing trace with no need to add hardware to the board, and any XDS emulator can be used.
Pin Trace is a technology that features a trace buffer outside of the device without losing the ability to capture all instructions that are executed by the processing core.

Being outside of the device, the trace buffer can have any size and allows additional configurability such as operation in both circular and one-shot mode. The one-shot mode gives an additional degree of observability as it allows retrieving the exact history of execution up to a given point of interest.

However, the fact the trace buffer is outside the device adds the requirement of having a high-speed communications interface to capture all data, thus requiring additional hardware and limiting to one the choice of XDS emulator to be chosen: the XDS560v2 PRO Trace.
Expanding on the concept of System Trace, think of it as a type of trace that monitors the entire device and not only the core processor. In other words, it monitors the core processor status, internal buses on the device and peripherals via special modules called CP tracers. Depending on the subsystem they have different functions, such as statistics for peripherals and the external memory interface (EMIF) or a messaging system for processor cores. Also, there is a module attached to the device’s internal bus that monitors traffic across cores and in the internal shared memory.

Therefore this trace does not capture instructions but event messages (memory accesses, power status, etc.) and, similarly to Core Trace, it sends them to the host PC together with timestamps for further analysis.

Once this data is transferred to the host PC and processed by CCS, the event messages are correlated with the source code and the advanced visualization tools can be used to see how the code interacts with the system.

Just for comparison with Core Trace, system trace messages do not occur with the same frequency as instructions, therefore its implementation is much less complex, although they still require some hardware buffering. That is the reason why system trace is not available in all devices and, similarly as for Core Trace, for the ones who have this feature there are two implementations with different levels of complexity: Embedded Trace Buffer, or ETB, and STM pin trace using an external emulator pod.

[Alternative text]
Expanding on the concept of System Trace, the idea behind it is to monitor how the multiple components of a device interact in real-time. These components are single or multiple cores, peripherals, bus controllers, memory, etc. Therefore this trace does not capture instructions but event messages (memory accesses, power status, etc.) and, similarly to Core Trace, it stores them on the host PC together with timestamps for further analysis.

Once this data is transferred to the host PC and processed by CCS, the event messages are correlated with the source code and the advanced visualization tools can be used to see how the code interacts with the system.
As shown in the previous slide, system trace can be captured using the Embedded Trace Buffer or ETB. This is the same buffer used for Core Trace, with the difference that the information that fills this buffer is re-routed from the STM module then read by the external JTAG emulator.

Similarly to Core Trace, the same benefits and disadvantages of ETB Trace are applicable. ETB is able to capture a smaller sample of system messages, and it only operates in circular mode, which means that old messages are overridden by newer ones. No hardware changes are necessary and it allows performing STM trace with any XDS emulator and any JTAG connector.
On the other hand, when using STM Pin Trace the STM Buffer is external to the device and the data is exported through extra JTAG pins.

Similarly to Core Trace, STM Pin Trace has the same advantages and disadvantages of pin trace are applicable. The STM Buffer can be very large and allows additional configurability such as operation in both circular and one-shot modes, therefore allowing to retrieve the exact history of system messages up to a given point of interest. The additional hardware usually does not involve a major redesign and can be optional in several cases, although some devices do not support STM with the common 14-pin JTAG connector.

At last, the fact the trace buffer is outside the device requires the use of an emulator equipped with a STM Buffer and Decoder, and two models can be chosen: a XDS560v2 STM or a XDS PRO Trace.
Here is a summary of the supported trace capabilities per type of Trace, including the required emulators, expected buffer sizes and hardware modifications.
This table summarizes the Trace features available per emulator family, and the main advantages and disadvantages of each. Basically the XDS560v2 PRO Trace is able to perform any type of Trace, while XDS560v2 only lacks Core pin trace, and the others are only able to do Trace if the device has an ETB.
This other table shows the features available per device. In summary, all devices that have a C6400, C6600 or an ARM core have some type of Trace, although only the C6000 devices sport Core Trace with external pins. Also, all devices that have an ARM core also have ETB, which automatically allows performing Core Trace using ETB but only the most modern are also capable of performing System Trace.

<table>
<thead>
<tr>
<th>Type of Trace</th>
<th>Technology</th>
<th>Device family</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core / Instruction ETB</td>
<td>C64x+ (C645x, C647x, DM64x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C66x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARM9 (AM180x, OMAPL13x, DM3x, DM644x, DM646x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cortex A (AM33x, AM35x, AM37x, AM38x, DM37x, DM81x, OMAP4/5)</td>
<td></td>
</tr>
<tr>
<td>External pins</td>
<td>C64x (C641x, C645x, C647x, DM64x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C66x</td>
<td></td>
</tr>
<tr>
<td>System ETB</td>
<td>AM335x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AM38x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C66x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DM81x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OMAP4/5</td>
<td></td>
</tr>
<tr>
<td>External pins</td>
<td>AM335x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C66x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OMAP4/5</td>
<td></td>
</tr>
</tbody>
</table>
Starting with CCSv5.4, the Trace setup procedure from inside Code Composer Studio is very straightforward and allows performing the most common tasks.

These features are easily accessible through the menu Tools → Hardware Trace Analyzer, and several default Trace jobs are available depending on the hardware capabilities of both the emulator and the device.

Also, it is possible to access advanced settings to completely customize the trace collection.

Additional details will be shown at the practical section of this module.
Also starting with CCSv5.4, the tools to perform analysis and graphical visualization of the Trace data are widely improved, with several dedicated functions for profiling, throughput analysis, code coverage and others.

These advanced visualization tools are automatically opened when a specific Trace job is selected.

Additional details will be shown at the practical section of this module.
One last aspect that can be very useful to debug systems is to embed the Trace capabilities in the system’s firmware, which is possible with a library suite called CtoolsLib.

These libraries feature APIs for all the components of Trace like STM and ETB, as well as advanced debugging capabilities called Advanced Event Triggering or AET.

Embedding such advanced debugging capabilities in the system’s firmware allows implementing self-monitoring and remote debugging functionality to it – an example is a product that is deployed located in remote places, for example.

Details about this library will not be covered in this presentation, but the two links shown are good references about this.