Application Report
SPRAC17B–January 2016–Revised November 2017

# A Guide to Debugging With CCS on the DRA7x, TDA2x and TDA3x Family of Devices

*Piyali Goswami, Stanley Liu, Richard Woodruff*

**ABSTRACT**

Being able to look into the state of the device when an application fails to run as expected is a key enabler while debugging the application. This application report walks through the different steps required to setup the TI Code Composer Studio™ (CCS), as well as how to debug applications on the DRA7x, TDA2x and TDA3x family of devices. The document starts with describing basic CCS debugging techniques and goes on to highlight advanced non-intrusive ways to debug software.

**Contents**

1 Different Stages of Debug ..... 3
2 Getting Started With CCS ..... 5
3 Configuring the Device With GEL Files ..... 12
4 Getting Used to the CCS GUI ..... 14
5 Breakpoints ..... 17
6 Processor Trace ..... 22
7 Throughput and Data Traffic Profiling ..... 34
8 References ..... 39

**List of Figures**

1 Debugging Techniques at Different Stages of Software Development ..... 3
2 Selecting Code Composer Studio v6 Updates in the CCS Install Pop Up Window ..... 6
3 Choosing Automotive Device Support for CSP Package Installation ..... 7
4 Hardware Setup to Get Started With Debugging With CCS ..... 8
5 Steps to Create and Launch a .ccxml File, Steps 1–6 ..... 10
6 Steps to Create and Launch a .ccxml File, Steps 7–14 ..... 11
7 Default GEL Files Associated With CPUs in the .ccxml File ..... 12
8 Selectively Running GEL Scripts ..... 13
9 Functions of Different Shortcuts in the Debug Window ..... 14
10 Register View From Cortex-A15 View in DRA7x ..... 15
11 Register View Find Pop Up Window ..... 15
12 Memory View From Cortex-A15 View in DRA7x ..... 16
13 Memory and Cache View From C66x DSP View in TDA2x ..... 16
14 DAP Debug SS Memory View ..... 17
15 Setting Breakpoints From the Breakpoint Window ..... 18
16 Configuring the Breakpoint From the Breakpoint Window ..... 19
17 Breakpoint Window Selection to Enable Hardware Watchpoint ..... 20
18 Hardware Watchpoint Configuration Window ..... 20
19 Breakpoint Properties Window to Configure the Cross Trigger Settings ..... 21
20 Steps to Start the Cortex-A15 PC Trace ..... 22
21 Cortex-A15 PC Trace Hardware Trace Analysis Configuration and Advanced Properties Windows ..... 23
22 Trace Viewer Window and Controls ..... 23


Copyright © 2016–2017, Texas Instruments Incorporated

23 Output of the Cortex-A15 PC Trace ..... 25
24 Steps to Enable the C66x DSP PC Trace ..... 26
25 C66x DSP PC Trace Hardware Trace Analysis Configuration and Advanced Properties Windows ..... 27
26 C66x DSP PC Trace Viewer Output ..... 27
27 Steps to Save the Trace Data to a CSV File ..... 27
28 CSV Export Data Pop Up Window ..... 28
29 Function Execution Graph and Program Address vs Cycle Graphical C66x DSP Trace Representation ..... 28
30 Steps to Enable the EVE SMSET Trace ..... 29
31 EVE SMSET Hardware Trace Analysis Configuration and Advanced Properties Settings ..... 30
32 Example EVE SMSET Trace Viewer Output ..... 31
33 Example EVE Analyzer Output ..... 31
34 Example Trace Viewer Output of EVE Software Messaging ..... 33
35 Steps to Enable Throughput and Data Traffic Profiling ..... 35
36 Throughput Profiling Hardware Trace Analysis Configuration and Advanced Properties Settings ..... 36
37 Example Output of the Throughput Profiling for an EDMA Transfer to Interleaved EMIF ..... 37
38 Sample Trace Viewer Output of an Average Transaction Latency of a DMA Transfer From DDR to OCMC . 38
39 OCP Watch Point Hardware Trace Analysis Configuration and Advanced Properties Settings ..... 39
40 Sample Trace Viewer Output of the OCP Watch Point Trace ..... 39

**List of Tables**

1 Some Key GEL Files for Device Initialization ..... 12

## Trademarks

Code Composer Studio is a trademark of Texas Instruments.
Cortex, ARM are registered trademarks of ARM Limited.
Android is a trademark of Google Inc.
Linux is a registered trademark of Linus Torvalds.
All other trademarks are the property of their respective owners.


Copyright © 2016–2017, Texas Instruments Incorporated

# 1 Different Stages of Debug

Based on the stage of software development, different techniques may be employed to perform debug on TI platforms. Figure 1 shows the debugging techniques at different stages of software development.

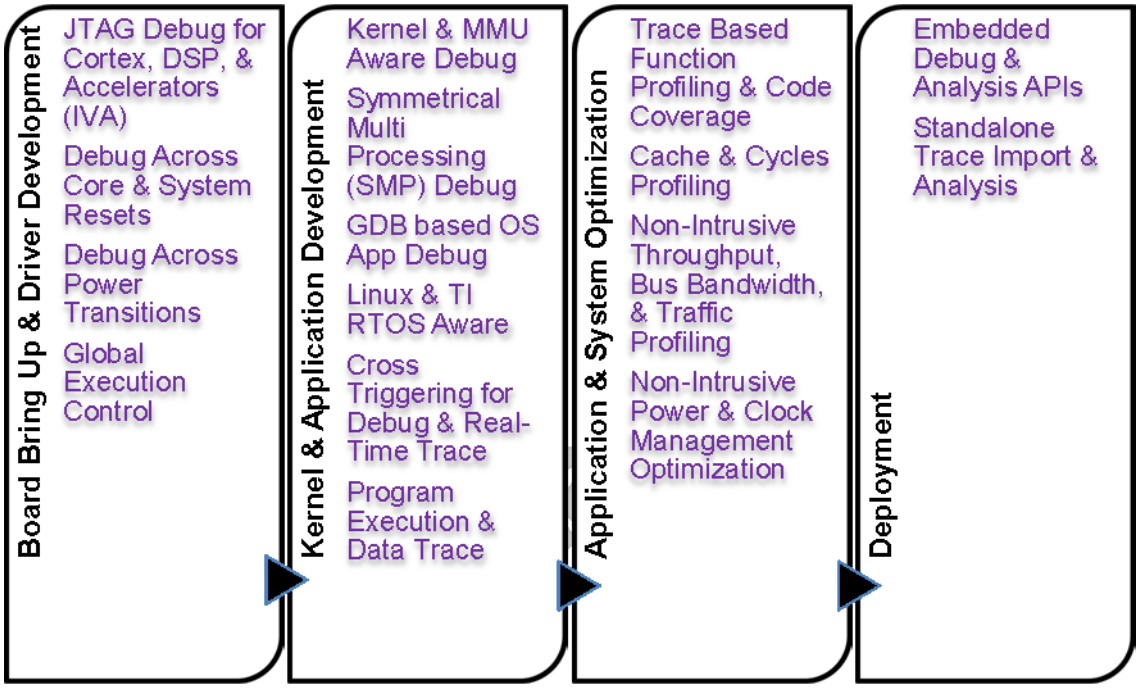



**Figure 1. Debugging Techniques at Different Stages of Software Development**

Debugging techniques may be intrusive or non-intrusive. When intrusive, the debug techniques may change the timing of execution of software and may or may not require software changes. Non-intrusive debugging techniques allow debugging software without changing the run time characteristics, or having to make any software modifications for debugging.

CCS allows multiple levels of intrusive and non-intrusive debugging techniques. Major debugging techniques available in the DRA7x, TDA2x and TDA3x family of devices are:

**Stop Mode Debug:**

- JTAG (IEEE 1149.1) support
- C66x, Cortex®-A15, Cortex-M4, EVE (ARP32), IVA (ARM9), and PRU debug
- Support for HW breakpoints, watch points, and performance counters
- Debug code from reset vector
- CPU and system reset debug
- Symmetrical multi-processing (SMP) debug
- Global run and halt across processors
- Debug across low-power transitions

**Processor Trace and Profiling:**

- DSP Trace – PC, cycles, data, and events
- Cortex-A15 – PC and cycles
- Embedded trace buffer (ETB)
- DSP cache and pipeline stalls profiling
- PC based function level profiling
- CToolsLib embedded trace instrumentation


Copyright © 2016–2017, Texas Instruments Incorporated

**SoC Instrumentation and Analysis:**

- HW assisted print
- Embedded Trace Buffer (ETB) and off-chip collection of system instrumentation
- Cache performance measurement for IPU, and EVE (SCTM)
- Bus bandwidth and latency profiling (Stats Collector)
- Bus interconnect traffic profiling (OCPWP)
- Power and clock management profiling
- IVAHD accelerators execution profiling
- CToolsLib for embedded debug and trace

**Run Mode Debug:**

- Linux®, Android™, and RTOS support
- Process level execution control
- Process level breakpoints

For the exact list of debugging capabilities of the device, see the device-specific TRM.


Copyright © 2016–2017, Texas Instruments Incorporated

## 2 Getting Started With CCS

TI Code Composer Studio can be downloaded from http://processors.wiki.ti.com/index.php/Download_CCS. The latest version of CCS (per this release) is CCS v6.1.1. The instructions to install CCSv6 from the installer can be downloaded from the Download_CCS link. Once CCS is installed on your system, the next step is to install the Chip Support Package (CSP) for the device you will be working with.

There are two ways to install the CSP package:

Option 1

1. In CCS window, Click on Help → Install New Software
2. In the pop up window select “Code Composer Studio v6 Updates” in the Work with: field as shown in Figure 2 and Figure 3.
3. Click on Next and follow the instructions in the pop up windows to install the CSP package.

Option 2

1. Download the package corresponding to the Automotive CSP from http://processors.wiki.ti.com/index.php/Device_support_files#Automotive.
2. Unzip the package and copy the folders “common” and “emulation” in the <CCS INSTALL DIR>\ccs_base folder.
3. Select merge folders while copying the folders.


Copyright © 2016–2017, Texas Instruments Incorporated

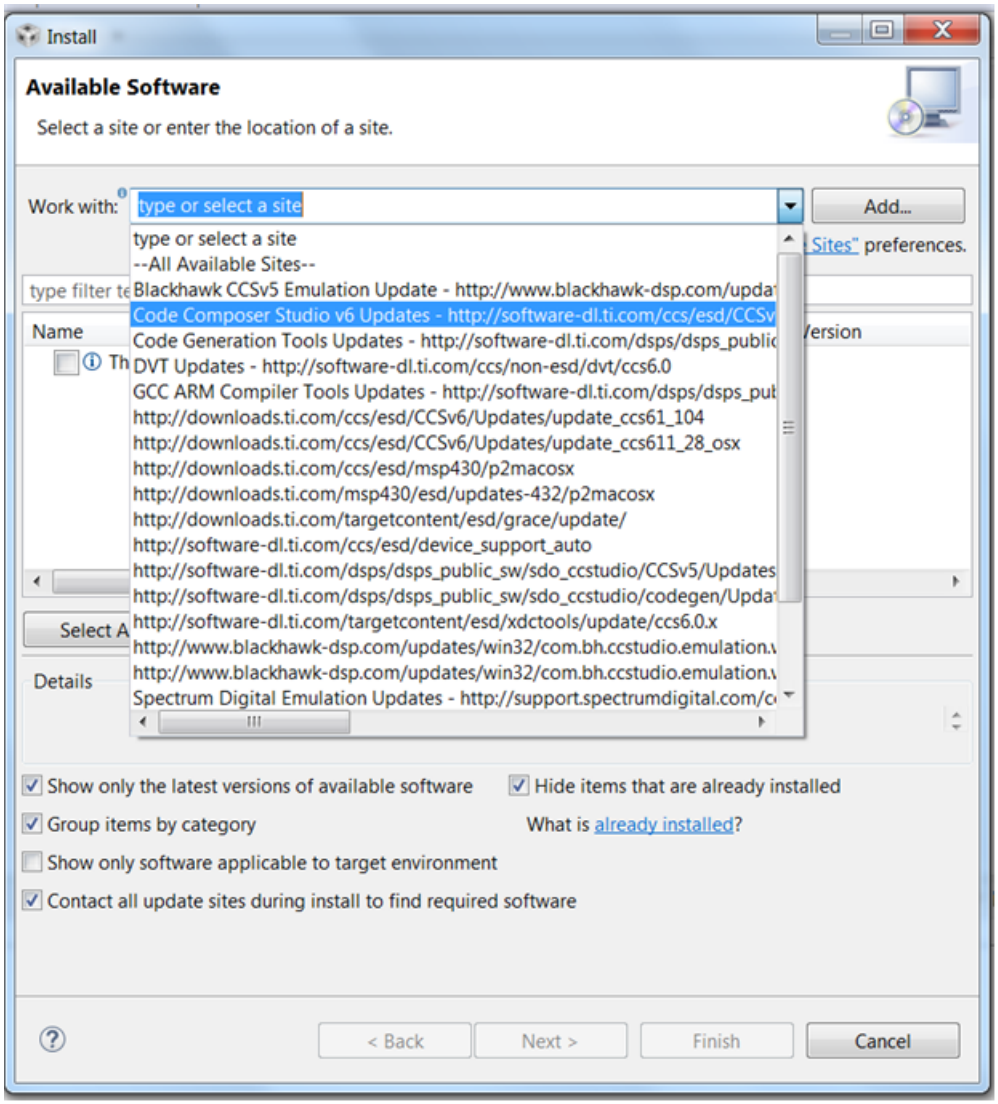



**Figure 2. Selecting Code Composer Studio v6 Updates in the CCS Install Pop Up Window**


Copyright © 2016–2017, Texas Instruments Incorporated

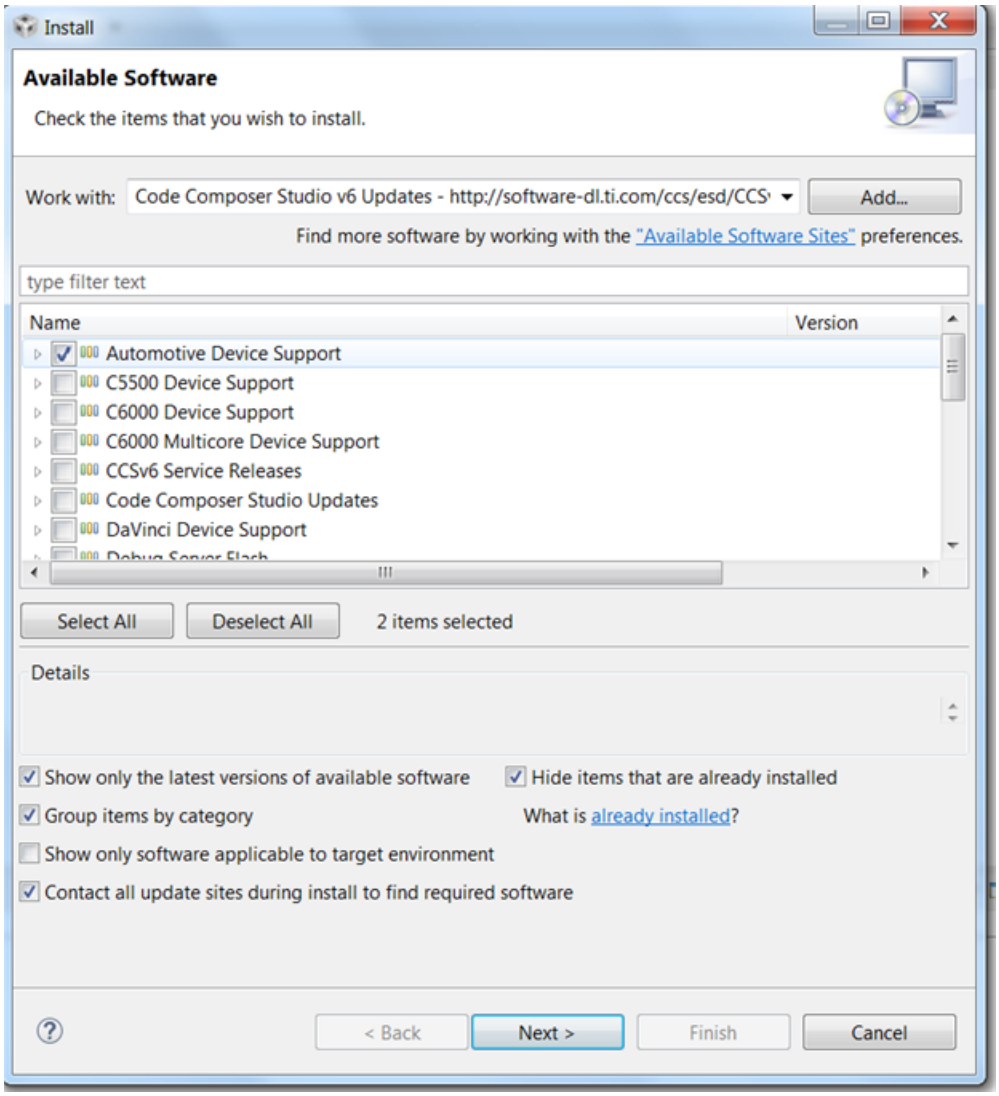



**Figure 3. Choosing Automotive Device Support for CSP Package Installation**


Copyright © 2016–2017, Texas Instruments Incorporated

## 2.1 Emulator Setup

The basic emulator setup to get started with debugging using CCS is shown in Figure 4.

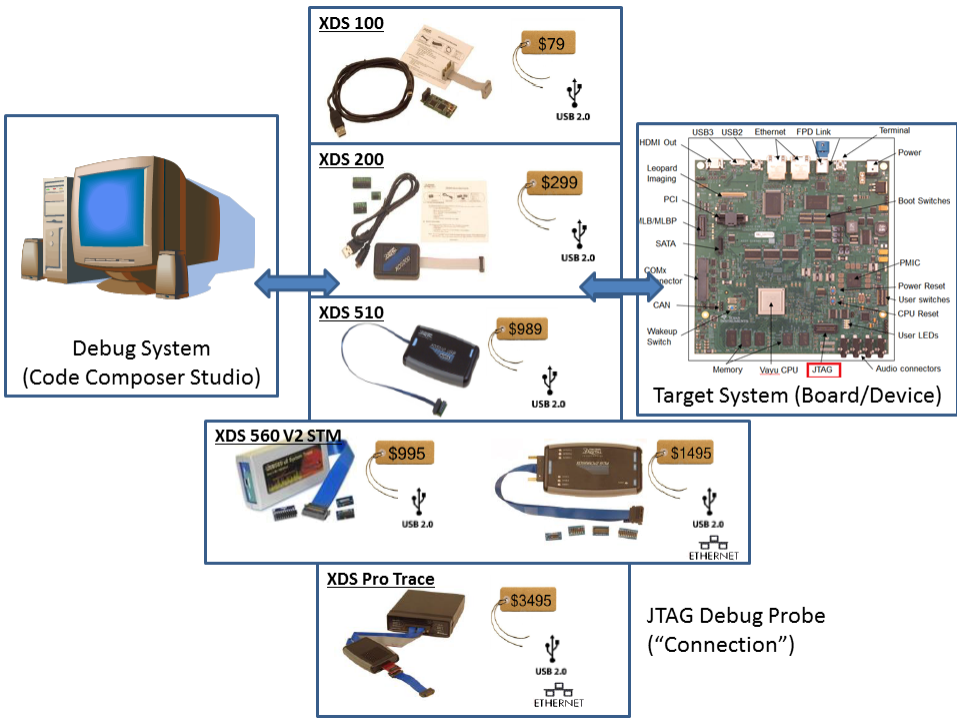



**Figure 4. Hardware Setup to Get Started With Debugging With CCS**

Different emulators are supported with the DRA7x, TDA2x and TDA3x devices. Each emulator has a set of features that it supports. Based on the debugging requirements, you can choose which emulator to connect with the device. High-level features of each supported emulator are summarized in the following list.


Copyright © 2016–2017, Texas Instruments Incorporated

**XDS100 v2:**

- Entry-level JTAG for hobbyists and universities
- CCS Cortex-A download speed is approximately 30 KB per second
- USB 2.0
- TI 14 and CTI 20 native
- Open HW reference design

**XDS200:**

- Performance JTAG at low-cost for users
- CCS Cortex-A download speed is approximately 300 KB per second
- ARM® Serial Wire Debug (SWD) and Serial Wire Output (SWO) support
- USB 2.0 and optional ENET
- TI, MIPI, and ARM connector option

**XDS510:**

- Performance JTAG
- USB 2.0
- TI 14 and CTI 20 native
- 3P EPK licensed

**XDS560 V2 STM:**

- High-performance JTAG and cJTAG for professional users
- CCS Cortex-A download speed is approximately 600 KB per second
- Low bandwidth trace receiver (STM and Cortex-M)
- USB 2.0 and ENET
- 4-pin at 100 MHz with auto-skew and jitter calibration
- 128 MB of storage

**XDS Pro Trace Receiver:**

- High-performance JTAG and cJTAG for professional users
- CCS Cortex-A download speed is approximately 600 KB per second
- High bandwidth dual-channel trace receiver (DSP, Cortex and STM)
- USB 2.0 and ENET
- 32-pin at 250 MHz DDR with auto-skew and jitter calibration
- 2 GB trace storage buffer

## 2.2 Creating the Target Configuration File

In order to debug DRA7x, TDA2x, or TDA3x family of devices, you must create a target configuration file (*.ccxml) (see Figure 5 and Figure 6). The list of steps to create a ccxml file are:

1. In the CCS window, click View.
2. Select Target Configuration.
3. In the Target Configuration window, right click and select New Target Configuration.
4. In the new Target Configuration window, give a name to the *ccxml file*.
5. Click Finish.
6. Select the connected emulator from the Connections drop-down list.
7. In the Board or Device field, type the first few letters to see a reduced board list.
8. Choose the desired board.
9. Save the file.
10. Right click on the *ccxml*.


Copyright © 2016–2017, Texas Instruments Incorporated

11. Launch the *ccxml*.
12. Right click on the device host CPU.
13. Click Connect.

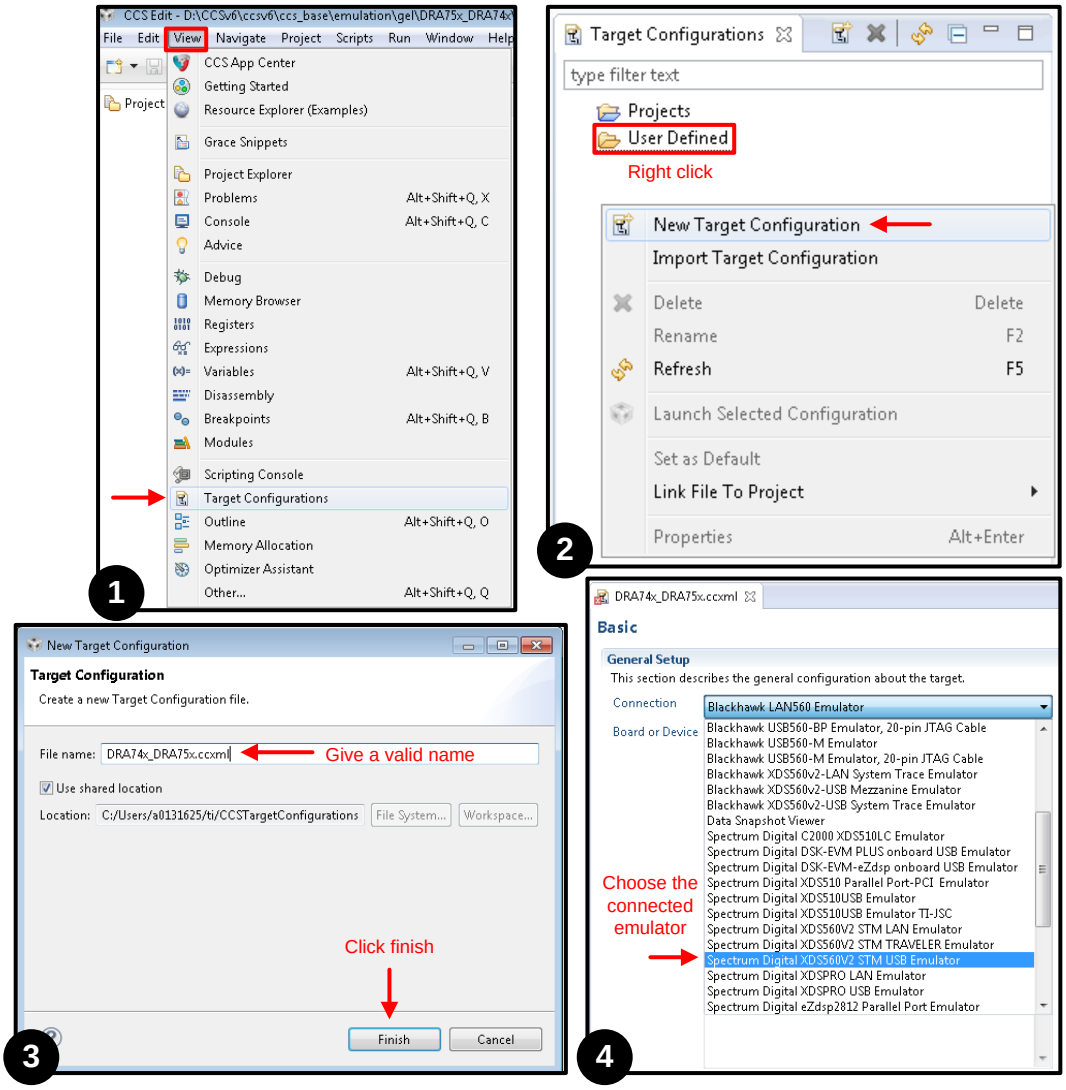



**Figure 5. Steps to Create and Launch a .ccxml File, Steps 1–6**


Copyright © 2016–2017, Texas Instruments Incorporated

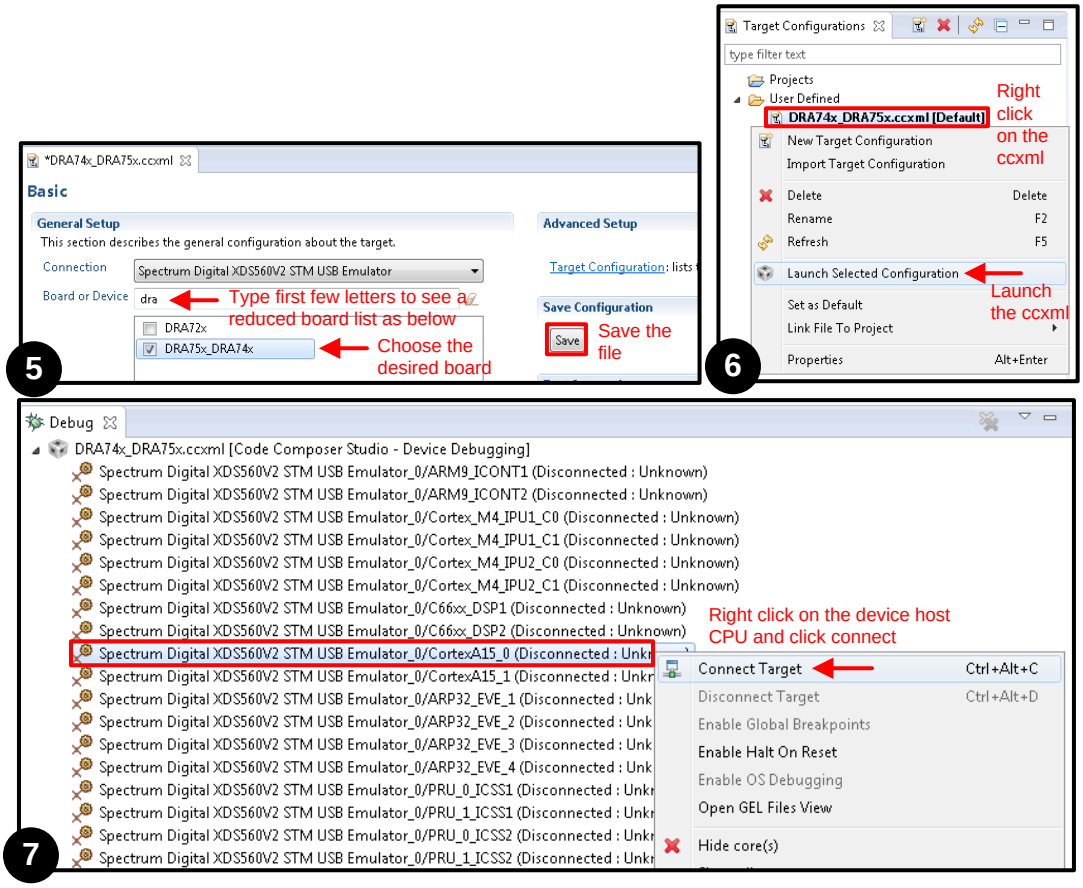



**Figure 6. Steps to Create and Launch a .ccxml File, Steps 7–14**

In cases when connecting to the device host fails, see the set of common issues in the JTAG connectivity at http://processors.wiki.ti.com/index.php/Debugging_JTAG_Connectivity_Problems.

**NOTE:** On DRA7x and TDA2x devices, the CCS connection is successful to the IPU2 only when the PM_CORE_PWRSTCTRL register's LOWPOWERSTATECHANGE bit is 0. This is zero by default, if you are using CCS GEL scripts for device initialization. However, in case you are booting with Linux/Android/QNX and trying to connect to the IPU2 via CCS, make sure you perform the following steps on the device terminal.

```
omapconf read 0x4AE06700
03FF0F17
omapconf write 0x4AE06700 0x3FF0F07
omapconf read 0x4AE06700
03FF0F07
```


Copyright © 2016–2017, Texas Instruments Incorporated

# 3 Configuring the Device With GEL Files

General Extension Language (GEL) files are used to automate the device initialization. GEL is an interpretive language similar to C that lets you create functions to extend the CCS IDE's usefulness. For details on how to write GEL files and the GEL syntax, see Creating Device Initialization GEL Files.

The DRA7x, TDA2x, and the TDA3x CSP packages include the device initialization GEL files that are linked to the different processors by default. The GEL files are located in the folder <CCS Installation Directory>\ccs_base\emulation\gel\<SOC Name>. Table 1 lists key GEL files for device initialization.

**Table 1. Some Key GEL Files for Device Initialization**

| GEL FILE NAME | DESCRIPTION |
|---|---|
| <SOC Name>_prcm_config.gel | PRCM functions, DPLLs, power and clocks initialization |
| <SOC Name>_ddr_config.gel | Configure EMIFs for DDR3 at 532 MHz or 400 MHz |
| <SOC Name>_pad_config.gel | Initialize PADCONF registers based on the TI EVM |
| <SOC Name>_multicore_reset.gel | Provides options to enable and reset another CPU core |
| <SOC Name>_startup_common.gel | Contains the initial GEL functions that run when the device host CPU is connected |

When the device host CPU is connected, a portion of the GEL initialization script (that is defined in the <SOC Name>_startup_common.gel file) is executed automatically.

Each CPU has a <SOC Name>_<CPU Name>_startup.gel file that loads the rest of the GEL files that are associated with the CPU.

The device .ccxml file stores the location of the <Soc Name>_<CPU Name>_startup.gel file that is loaded when the .ccxml file is launched.

The GEL file associated with the CPU core may be seen in the Advanced tab of the .ccxml file as shown in Figure 7. In case you do not want to run any GEL files, clear the initialization script field entry corresponding to each CPU in the .ccxml Advanced tab.

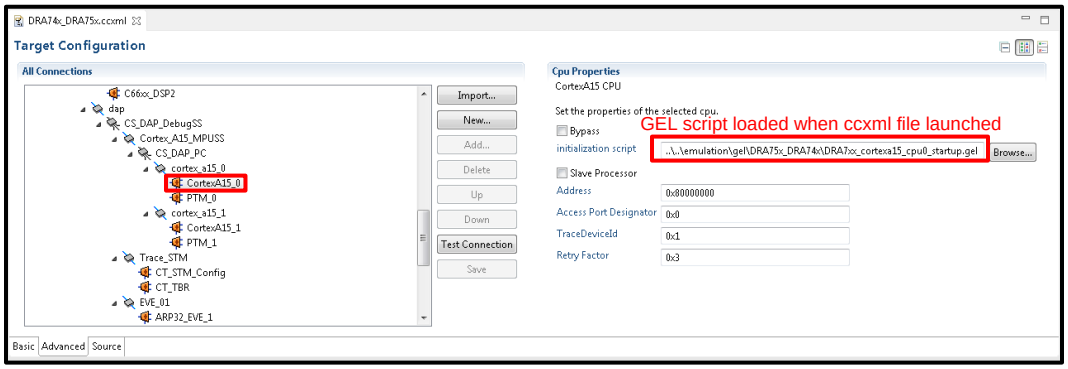



**Figure 7. Default GEL Files Associated With CPUs in the .ccxml File**


Copyright © 2016–2017, Texas Instruments Incorporated

To run a GEL file, click on the desired options from the Scripts menu. Click on the CPU where the GEL script is executed from the Debug window, as shown in Figure 8.

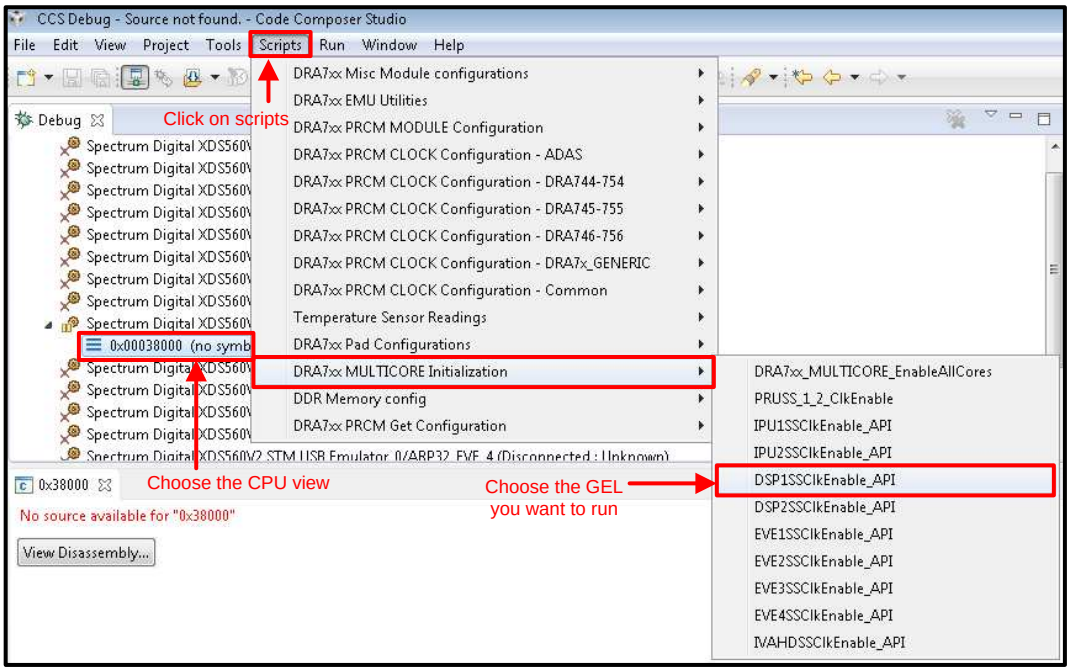



**Figure 8. Selectively Running GEL Scripts**


Copyright © 2016–2017, Texas Instruments Incorporated

# 4 Getting Used to the CCS GUI

CCS provides multiple views and windows to enable you to debug the application. This section discusses the basic windows and views that enable you to debug a particular CPU core, reset it, and look at register or memory content.

## 4.1 Debug View

The Debug window enables you to choose the CPU core to debug. The Debug window also provides multiple shortcuts to allow the core to run, halt, single-step a C instruction, step over an assembly function, and call and rest the CPU or whole device. A description of all buttons in the Debug window is shown in Figure 9. The same options are available in the Run menu option as well.

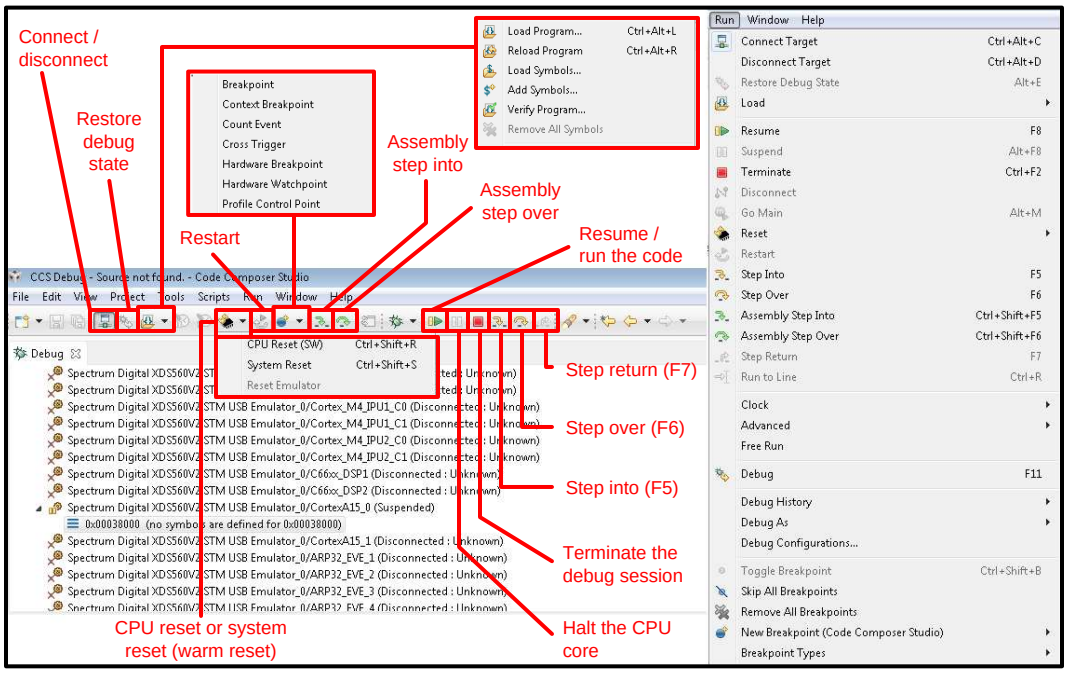



**Figure 9. Functions of Different Shortcuts in the Debug Window**

The Load Program option is used to load the software in the device memory for the CPU to start executing it. CCS moves the CPU PC to the entry point of the code when loaded through the Load Program option, whereas Load Symbols load only debug information present in the executable. Load Symbols are typically used when the CPU code is loaded through a boot medium and debug symbols are loaded using CCS.


Copyright © 2016–2017, Texas Instruments Incorporated

## 4.2 Register View

To know the values of registers of a CPU core, open the Register view from the View → Registers menu option. The Registers window shows the values of all CPU GP registers and control registers. Additionally, for the device host CPU, peripheral registers are also shown with bit fields and brief text describing them (see Figure 10).

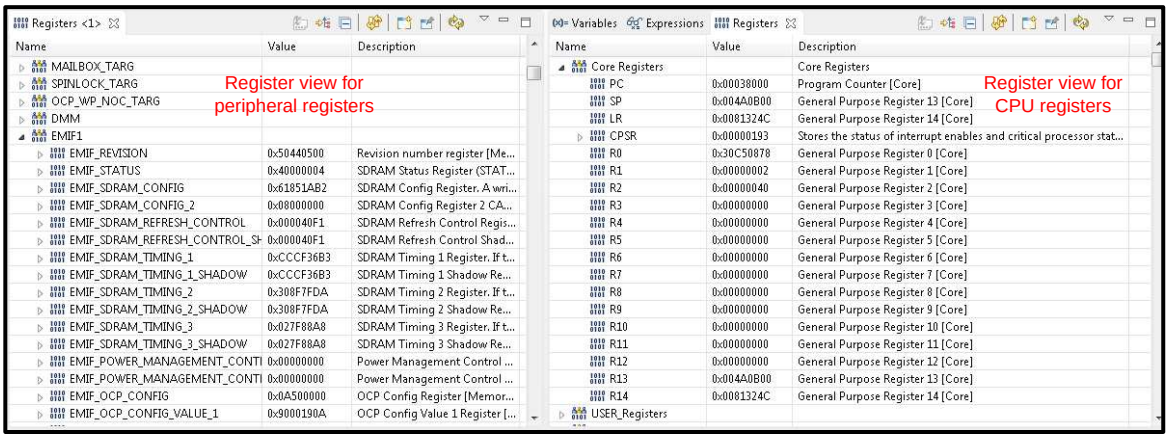



**Figure 10. Register View From Cortex-A15 View in DRA7x**

1. Type Ctrl+F or right click → Find to search for any register in the register window.
2. Figure 11 shows the register view find pop up window.
3. In this pop up, enter a portion of the string of the register name and find the register you are searching for.

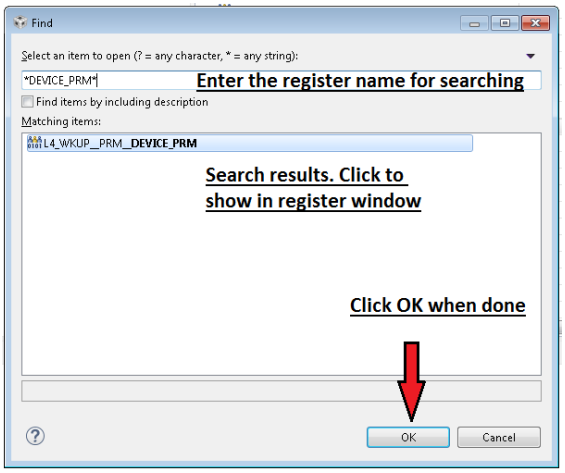



**Figure 11. Register View Find Pop Up Window**


Copyright © 2016–2017, Texas Instruments Incorporated

## 4.3 Memory and Cache View

The Memory View allows viewing CPU view, intermediate physical view (in the Cortex-A15 hypervisor applications) and physical view of the memory space. Memory Mapped Registers (MMRs) and memory content can be viewed in the following steps (see Figure 12).

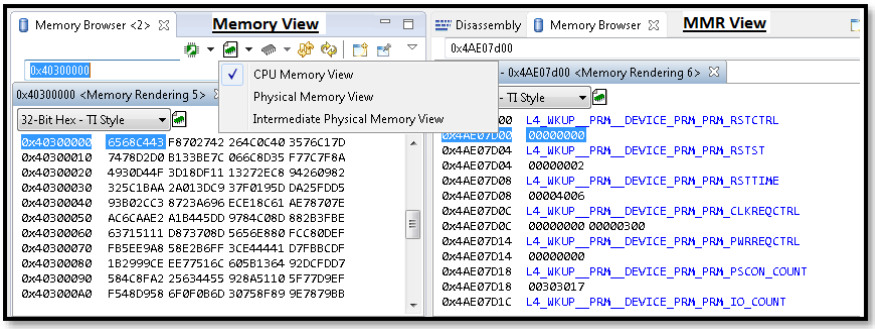



**Figure 12. Memory View From Cortex-A15 View in DRA7x**

1. Choose Memory Browser from the menu option View.
2. In the Memory Browser window, type the address in the field Enter Location.
3. Choose the Memory View.
4. You can also view the portion of memory in the L1 and L2 cache for the DSP C66x.
5. Select the menu optionView → Other and a pop up window appears.
6. Select cache to analyze the DSP L1P, L1D, and L2 Cache contents and cache line properties.

The memory view is also color coded to indicate if the data that is viewed is L1 or L2, or cached and non-cached data.

Figure 13 shows the memory and cache view from the C6xx DSP view in TDA2x.

**NOTE:** The cached view is available only for the C66x DSP.

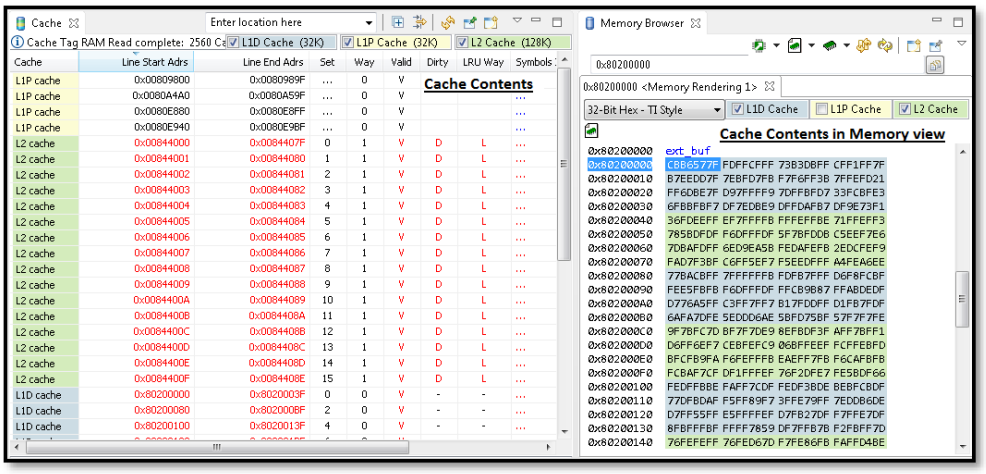



**Figure 13. Memory and Cache View From C66x DSP View in TDA2x**


Copyright © 2016–2017, Texas Instruments Incorporated

### 4.4 DebugSS View

In some cases while debugging if the CPU core is hung or you want to see memory or register contents from a system view while the CPU is running, the DAP_DebugSS view can be used. The DAP_DebugSS view can be enabled with the following steps.

1. Right click on the connection in the Debug view
2. Click on Show All Cores
3. Memory view (system view) shows system view of the memory (same as what you saw from the Cortex-A15 context in the memory view).
4. Memory view may be changed to APB view from the memory windows. Figure 14 shows the DAP debug SS memory view.

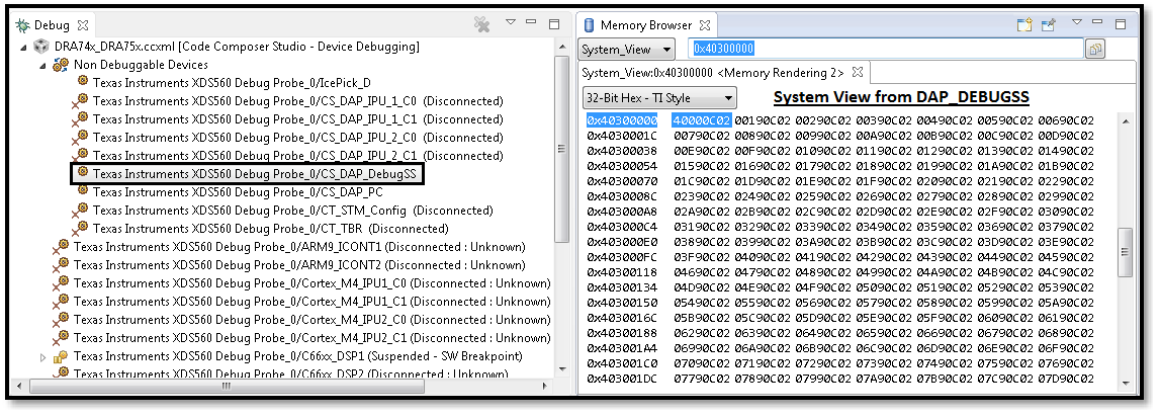



**Figure 14. DAP Debug SS Memory View**

### 4.5 Disassembly View

The CPU opcodes of the software executing on the target can be viewed in the Disassembly window (View → Disassembly) of the CPU. Assembly step into and step over buttons in the debug window can be used to step through the disassembly.

## 5 Breakpoints

DRA7x, TDA2x, and TDA3x support different types of *breakpoints* that halt the execution of the CPU core. Each ISA supports a different list of breakpoint options. The different types of breakpoints supported by each CPU ISA from the breakpoint window (View → Breakpoints) are as given:

- **Cortex-A15:**
  - Breakpoint
  - Count Event
  - Cross Trigger
  - Hardware breakpoint
  - Hardware watchpoint
  - Profile Control Point
- **Cortex-M4:**
  - Breakpoint
  - Hardware breakpoint
  - Profile Control Point


Copyright © 2016–2017, Texas Instruments Incorporated

- **C66x DSP:**
  - Breakpoint
  - Chained Breakpoint
  - Count Event
  - Data access count
  - Hardware breakpoint
  - Hardware watchpoint
  - Profile Control Point
- **ARP32 EVE:**
  - Breakpoint
  - Hardware breakpoint
  - Hardware watchpoint

## 5.1 Software and Hardware Breakpoints

Breakpoints are program locations where the processor must halt so that debugging can occur. The link http://processors.wiki.ti.com/index.php/How_Do_ Breakpoints_Work gives information regarding how software and hardware breakpoints work.

Both hardware and software break points allow the CPU to halt at a given PC location. Hardware breakpoints can be used regardless of whether the code being executed is in RAM or ROM. However, Software breakpoints can be used, only volatile memory where the contents of the memory are modified to indicate the debugger to halt at the PC location.

The advantage of the SWBP is that you can set an unlimited number of them in as many places desired. The disadvantage is that you cannot put them in non-volatile memory, such as ROM, FLASH, and more, because CCS cannot write the opcode to the location. The disadvantage of the HWBP is because they are implemented in hardware, they are limited in number.

Double clicking on the PC location of the C code or the assembly instruction in the disassembly window creates a software breakpoint or hardware breakpoint based on the memory type automatically. Alternatively, you can set the breakpoint from the breakpoint window (see Figure 15).

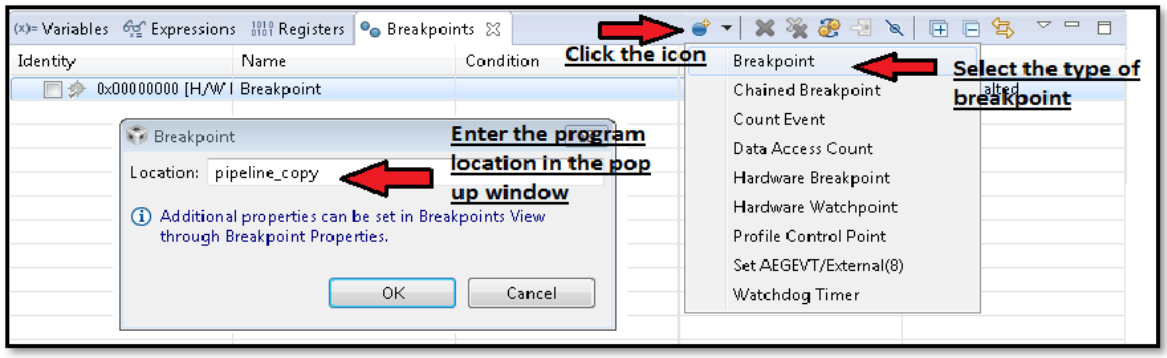



**Figure 15. Setting Breakpoints From the Breakpoint Window**


Copyright © 2016–2017, Texas Instruments Incorporated

Optionally, you can set alternate functions when the CPU hits the desired PC location by changing the breakpoint options (see Figure 16).

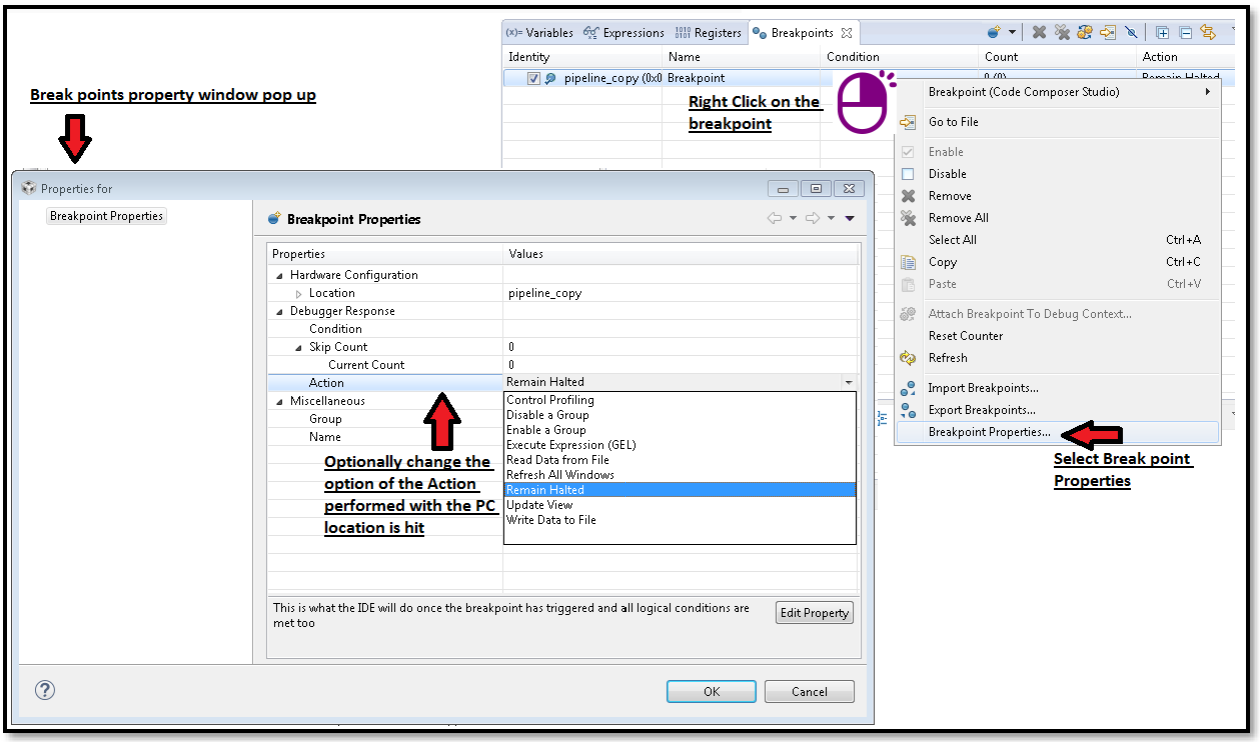



**Figure 16. Configuring the Breakpoint From the Breakpoint Window**

## 5.2 Chained Breakpoint

The Event Analysis tool allows you to chain breakpoints. For example, if you want to track a bug in frequently-executing code that only surfaces after the execution of a separate piece of corrupt code. It is not required to break every time the frequently-executed code is run. The code does not need to break after the corrupt code is executed.

Configuring a chained breakpoint is very similar to a normal break point, except that you must give two locations. Only when the first location is executed, the code halts at the second location.

## 5.3 Count Event

The Count Event functionality allows counting multiple CPU events which give insight into the different system events, such as cache hits and misses, pipeline stalls, and more. The A15 and DSP support counting events. The A15 event to be profiled can be selected while defining the Count Event break point. For DSP, the event to be profiled can be selected after creating the Count Event breakpoint and changing the property by right clicking on the breakpoint, and selecting Breakpoint Properties.


Copyright © 2016–2017, Texas Instruments Incorporated

## 5.4 Hardware Watchpoint

The Hardware Watchpoint can be used to halt the CPU when the CPU reads or writes from or to a particular memory location. Use the following steps to setup a hardware watchpoint:

1. Open the breakpoint dialog: View → Breakpoints in the Debug Perspective.
2. Select Hardware Watchpoint in the pull down menu.

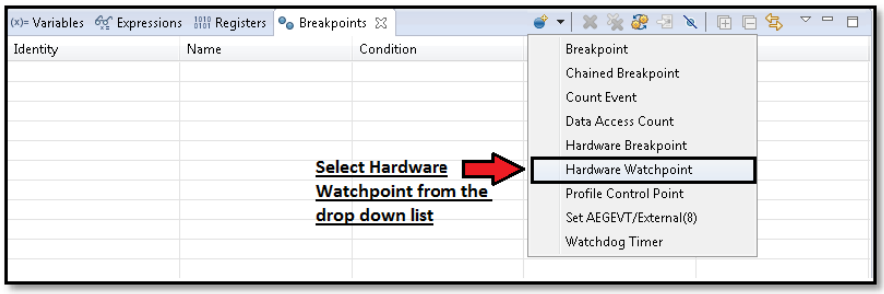



**Figure 17. Breakpoint Window Selection to Enable Hardware Watchpoint**

3. In the location field, specify the address that you want to watch, and if it stops on a read or write access. For example, if you want to watch when a global variable named *ext_buf* is written to.
4. You can specify *ext_buf* for the location and write for the memory access.

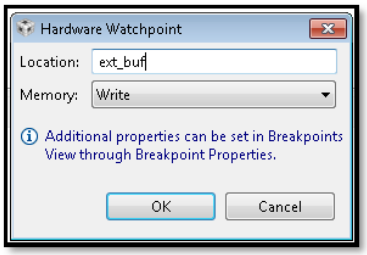



**Figure 18. Hardware Watchpoint Configuration Window**

5. If needed, the watchpoint can be further customized by right clicking on the Watchpoint and selecting Properties. From this dialog box, you can:
   1. Configure a data read and write of a particular value from the address
   2. Configure the size of the data
   3. Configure a mask value (bits that may be ignored)

## 5.5 Profile Control Points

Profile Control Points are used to enable and disable collection for function profiling or code coverage. These points can be used to exclude a certain range from getting profiled and vice versa. More information on profile control points is present in http://processors.wiki.ti.com/index.php/Profiler#Profile_Control_Points. Profile control points are supported for Cortex-A15, Cortex-M4, and DSP.


Copyright © 2016–2017, Texas Instruments Incorporated

## 5.6 Cross Trigger

The device supports a cross-triggering feature, which provides a way to propagate debug (trigger) events from one processor subsystem and module to another. For example, a given subsystem A can be programmed to generate a debug event, which can then be exported as a global trigger across the device. Another subsystem B may be programmed to be sensitive to the trigger line input and to generate an action upon trigger detection.

1. Cross-triggering can be configured from the Cortex-A15 view. In the breakpoint window, select Cross Trigger to create a dummy cross Trigger Breakpoint.
2. The properties of this cross trigger breakpoint can be altered if you right click the Cross Trigger Breakpoint and select Properties.
3. A pop up window appears on this action.
4. The property window allows configuring four cross trigger channels.
5. For each channel, configure the Event Watcher, which decides what event in the device causes the trigger to be generated.
6. The Action Trigger decides what action is taken when the trigger occurs.

Figure 19 shows the Breakpoint Properties window to configure the cross trigger settings.

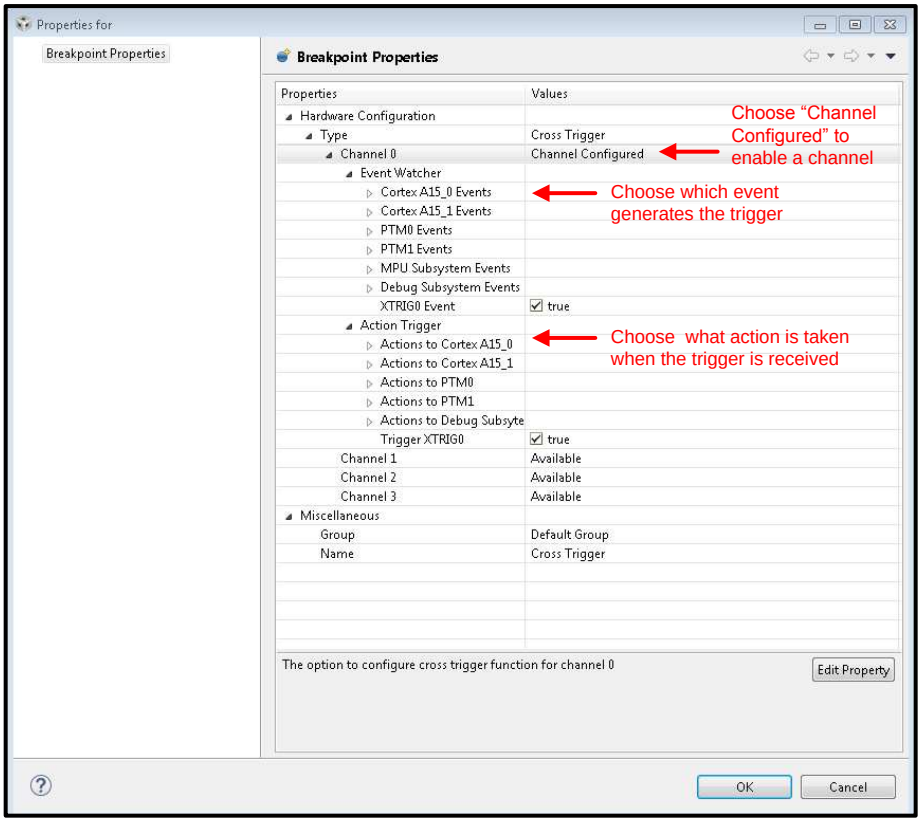



**Figure 19. Breakpoint Properties Window to Configure the Cross Trigger Settings**


Copyright © 2016–2017, Texas Instruments Incorporated

# 6 Processor Trace

The device supports processor trace for the Cortex-A15 and the DSP processors. Steps to enable the Cortex-A15, DC processor trace and interpret output data using CCS are discussed in the following subsections.

## 6.1 Cortex-A15 PC Trace

The Cortex-A15 supports a non-intrusive PC trace that indicates what instruction runs with each cycle. The trace can be directed to the internal device 32 KB buffer (the ETB) or can be directly sent to the emulator through TPIU.

**NOTE:** The ETB Buffer is a circular buffer and holds the last portion of the CPU trace data when the buffer is full and wraps around.

1. To enable PC trace, keep the debug context to the Cortex-A15.
2. From the Tools menu option, select Hardware Trace Analyzer → PC Trace (see Figure 20).

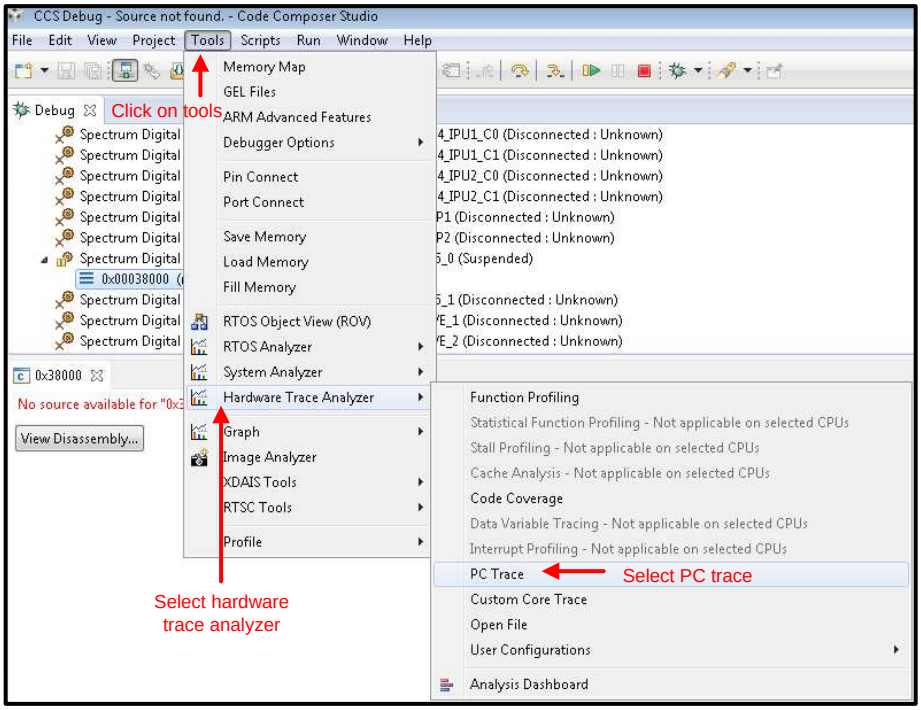



**Figure 20. Steps to Start the Cortex-A15 PC Trace**


Copyright © 2016–2017, Texas Instruments Incorporated

3. This action opens a pop up window that can be used to further configure the PC trace feature. Figure 21 shows the Cortex-A15 PC trace hardware trace analysis configuration and Advanced Properties windows.

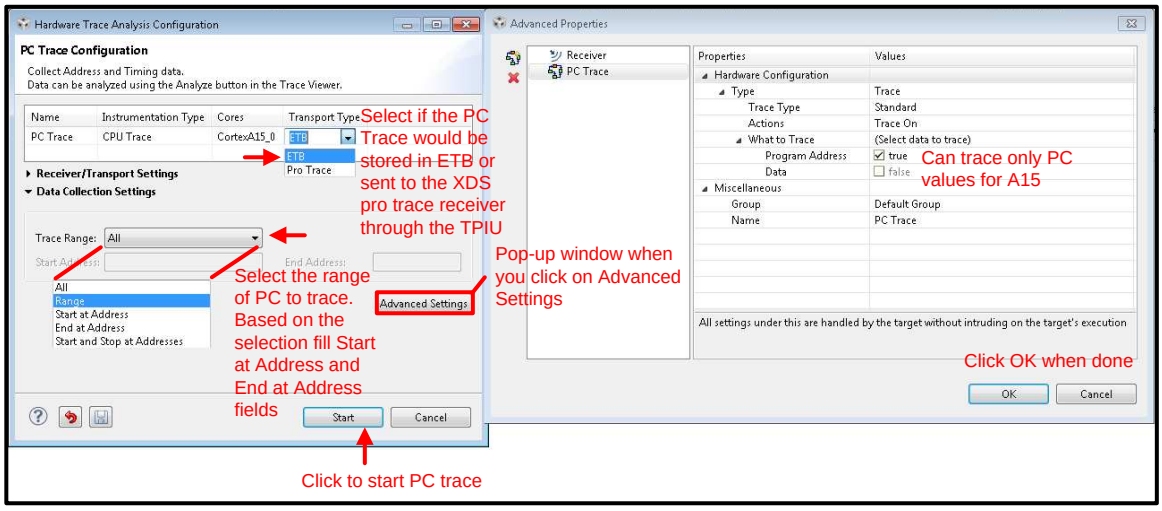



**Figure 21. Cortex-A15 PC Trace Hardware Trace Analysis Configuration and Advanced Properties Windows**

4. On starting the trace functionality, CCS opens a Trace Viewer window (see Figure 22).

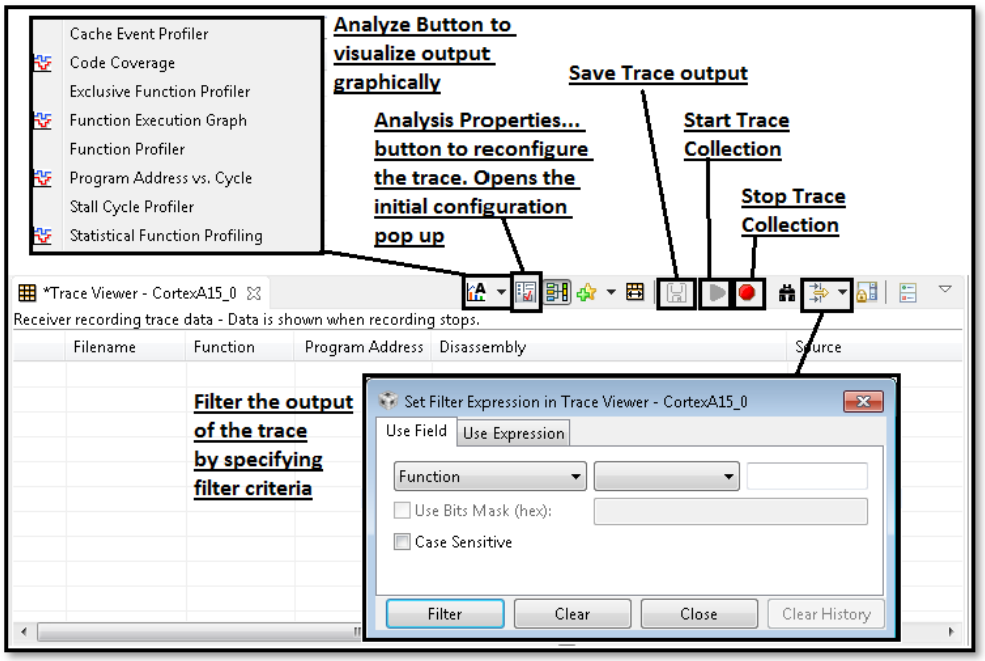



**Figure 22. Trace Viewer Window and Controls**

5. Once the CPU core is run, the Trace Viewer captures the PC Trace. It provides a tabular output that describes how many CPU cycles an assembly instruction of a particular function took.
6. The cycle count is given by the number of A15 CPU cycles.


Copyright © 2016–2017, Texas Instruments Incorporated

The cycle count is updated at every waypoint. A waypoint is a point where instruction execution by the processor may involve a change in the program flow. The CoreSight Processor Trace Macrocell (PTM) traces only the following waypoints:

- All indirect branches
- All conditional and unconditional direct branches
- All exceptions
- Any instruction that changes the instruction set state of the processor
- When halting debug mode is enabled, entering or leaving debug state
- Synchronization primitives

CCS can then reconstruct the complete instruction flow from these waypoints to create a graphical representation of the code flow in Figure 29. Additional analysis can then be performed by looking at the Function Profiler: Summary window, which gives details on the number of cycles spent in every function (inclusive and exclusive).

***Exclusive time*** is the amount of execution time that passed within that function *excluding* the time spent in functions called from that function.

***Inclusive time*** is the amount of execution time that passed within that function *including* the time spent in function called from that function. Figure 23 shows the output of the Cortex-A15 PC trace.


Copyright © 2016–2017, Texas Instruments Incorporated

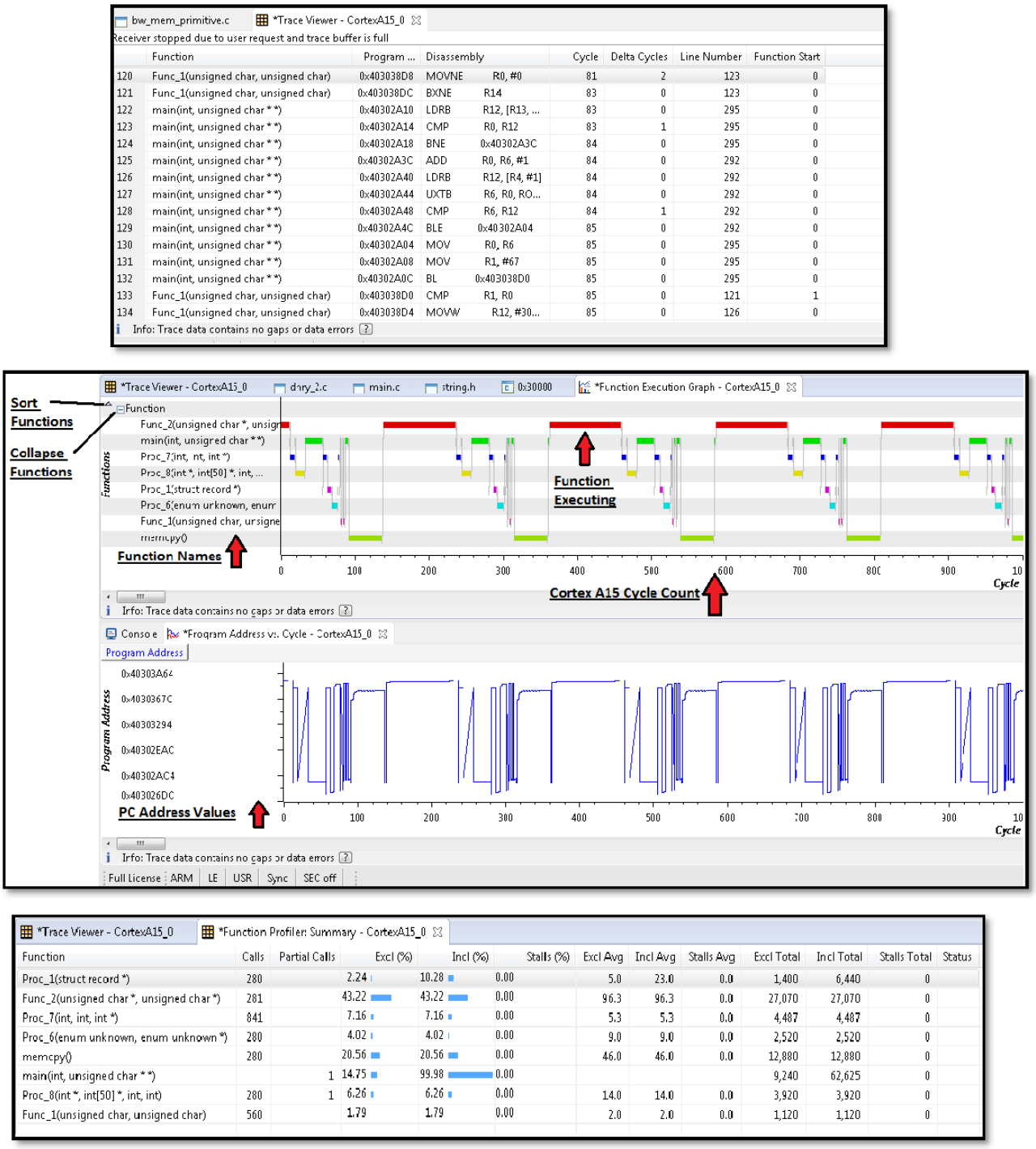



**Figure 23. Output of the Cortex-A15 PC Trace**

Right click on the Trace Viewer window to save the output data in CSV format for offline analysis.


Copyright © 2016–2017, Texas Instruments Incorporated

## 6.2 C66x DSP PC Trace

The DSP allows the tracing of the ***program counter, data access address and values*** non-intrusively. The infrastructure component uses bus snoopers to collect and export trace data using hardware dedicated to the trace function. Advanced Event Triggering facilities provide a means to restrict the trace data exported to data of interest to maintain the non-intrusive aspect of trace. This reduces the export bandwidth requirements and facilitates the successful collection of the data of interest.

1. To start the DSP core trace, click on Tools → Hardware Trace Analyzer → PC trace (see Figure 24).
2. Ensure debug scope is kept to C66x_DSP1/2 core in the Debug window.

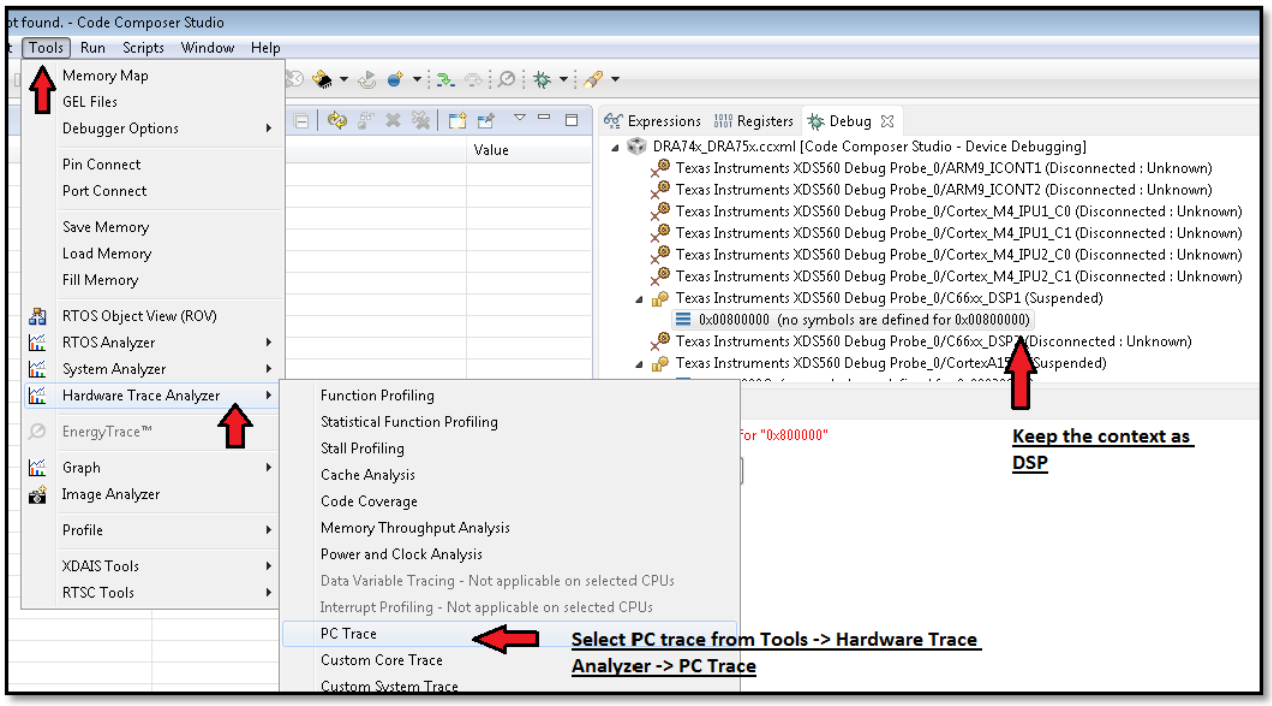



**Figure 24. Steps to Enable the C66x DSP PC Trace**

3. On starting the PC trace, the Hardware Trace Configuration pop up window shows up.
4. You can configure the transport type to ETB (internal 32 KB circular buffer) or XDS Pro Trace dump from the TPIU to the emulator buffer.

**NOTE:** The ETB buffer is a circular buffer and holds the last portion of the CPU trace data when the buffer is full and wraps around.

5. You can optimally define the PC address range across which to trace the DSP by using the Trace Range drop down menu.
6. Click on Advanced Settings to open the Advanced Properties pop up window.
7. The Receiver ETB properties can be updated in the Advanced Properties pop up.
8. In most cases, the default values are good for DSP tracing.
9. In the PC Trace trigger context, you can define the different data you want to trace.
10. Once selected, click Ok.
11. Click Start in the Hardware Trace Analysis Configuration pop up window to open the Trace Viewer. There is a tradeoff between the number of instructions traced and the amount of data traced for each instruction because the amount of trace memory is limited.


Copyright © 2016–2017, Texas Instruments Incorporated

Figure 25 shows the C66x DSP PC trace hardware trace analysis configuration and advanced properties windows.

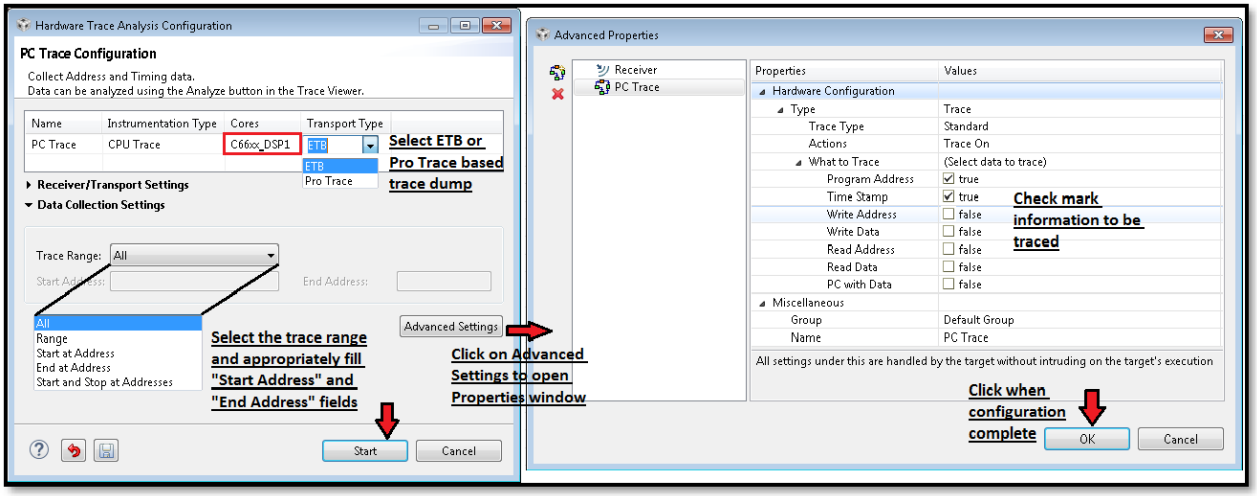



**Figure 25. C66x DSP PC Trace Hardware Trace Analysis Configuration and Advanced Properties Windows**

12. A Trace Viewer window starts capturing trace information once the DSP core starts running.
13. The Trace Viewer is populated with trace information once the DSP core is halted.
14. If you see the message Receiver recording trace data, the Trace Buffer wrapped around. Data is shown when the recording stops, the ETB circular buffer is full and has wrapped around.
15. On halting the DSP core, or clicking on the Stop Trace button, the Trace viewer is populated with the trace data entries (see Figure 26). Unlike the Cortex-A15, the cycle count is updated for each instruction. The Delta Cycles column shows the number of cycles taken by each instruction. Pipeline stall information is also shown.

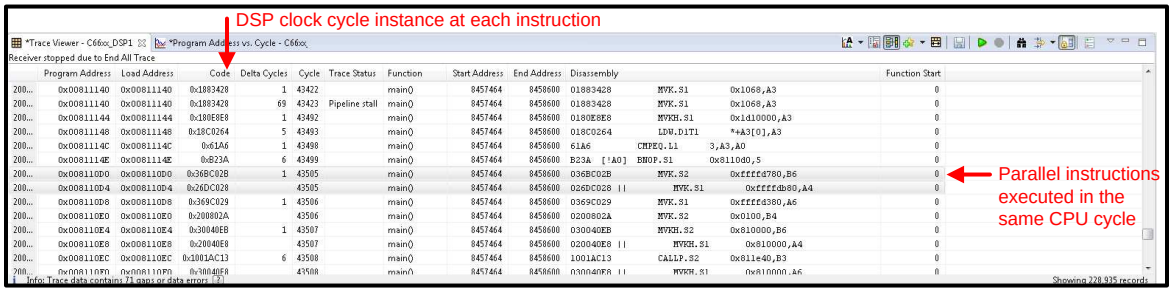



**Figure 26. C66x DSP PC Trace Viewer Output**

16. To save the trace data to a CSV file for off line analysis, right click in the Trace Viewer Window. Choose Data → Export (see Figure 27).

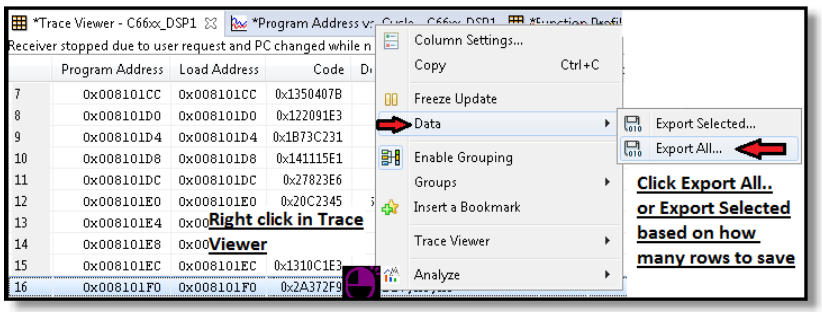



**Figure 27. Steps to Save the Trace Data to a CSV File**


Copyright © 2016–2017, Texas Instruments Incorporated

17. An Export Data pop up opens (see Figure 28). You can choose which columns to save and the file name through this window.

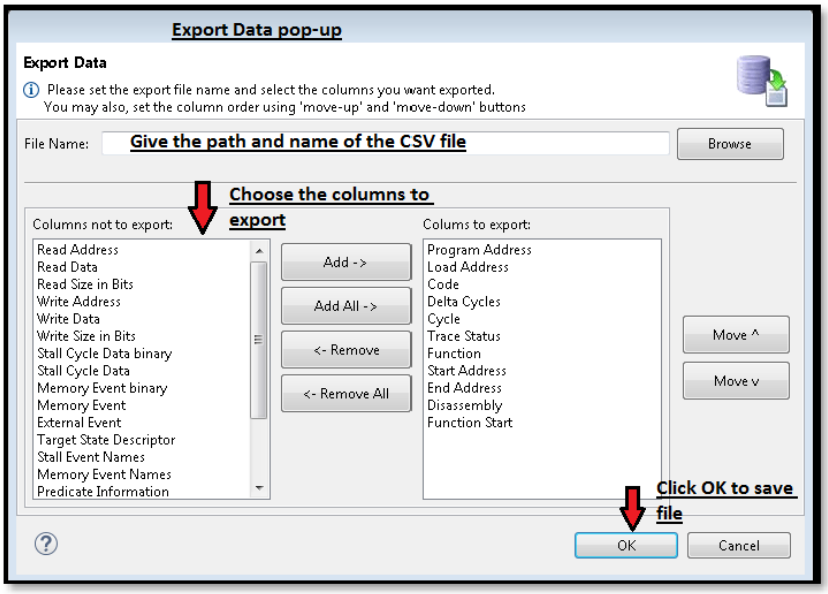



**Figure 28. CSV Export Data Pop Up Window**

18. To analyze the trace output graphically, click on the Analyze button in the Trace Viewer window. An example of the Function Execution Graph and the Program Address vs. Cycle is shown in Figure 29.

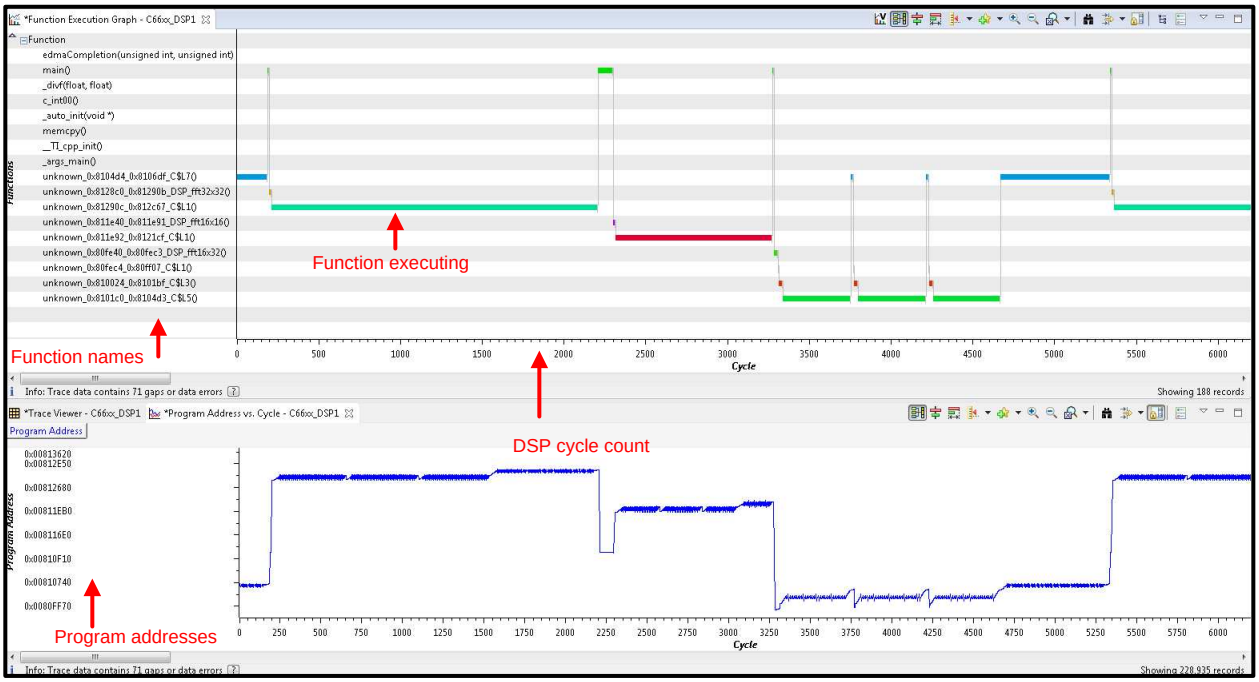



**Figure 29. Function Execution Graph and Program Address vs Cycle Graphical C66x DSP Trace Representation**

Similar to the Cortex-A15, the *Function Profiler: Summary* can also be generated from the DSP trace data, as shown in the function profiler summary in Section 6.1.


Copyright © 2016–2017, Texas Instruments Incorporated

## 6.3 EVE SMSET Trace

### 6.3.1 EVE System Event Trace

The Software Messaging and System Event Trace (SMSET) is an IP block that allows non-intrusive tracing of key EVE system events, as well as software messages from the ARP32 CPU. The SMSET module accepts software messages through its OCP target port, and accepts key system events through the system event input. These messages and events are queued locally in SMSET and written to the chip-level Software Trace Module (STM) through the SMSET (and EVE) OCP Debug Initiator port. The STM module then traces these messages along with trace content from other chip-level agents. For more information on the EVE SMSET, see the device-specific TRM.

1. To enable EVE SMSET tracing, select Custom System Trace from the Tools menu (see Figure 30).

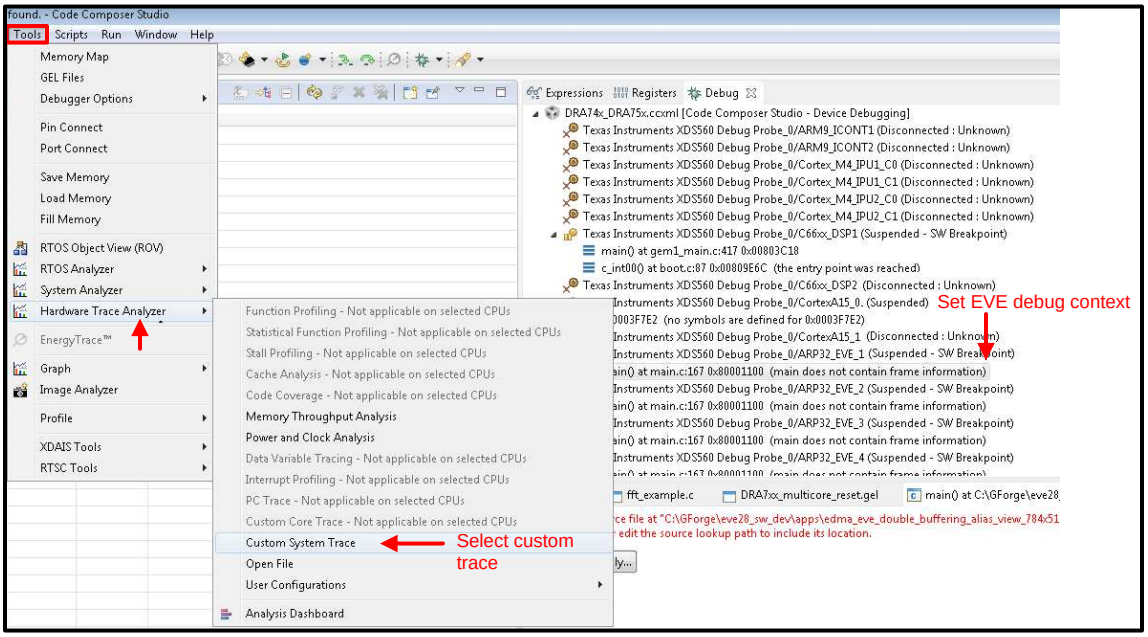



**Figure 30. Steps to Enable the EVE SMSET Trace**

2. This action opens a Hardware Trace Analysis Configuration pop up window.
3. Click on Advanced Settings to open the Advanced Properties pop up window.


Copyright © 2016–2017, Texas Instruments Incorporated

4. Click on the New Trace Trigger icon to create a new trace (see Figure 31).

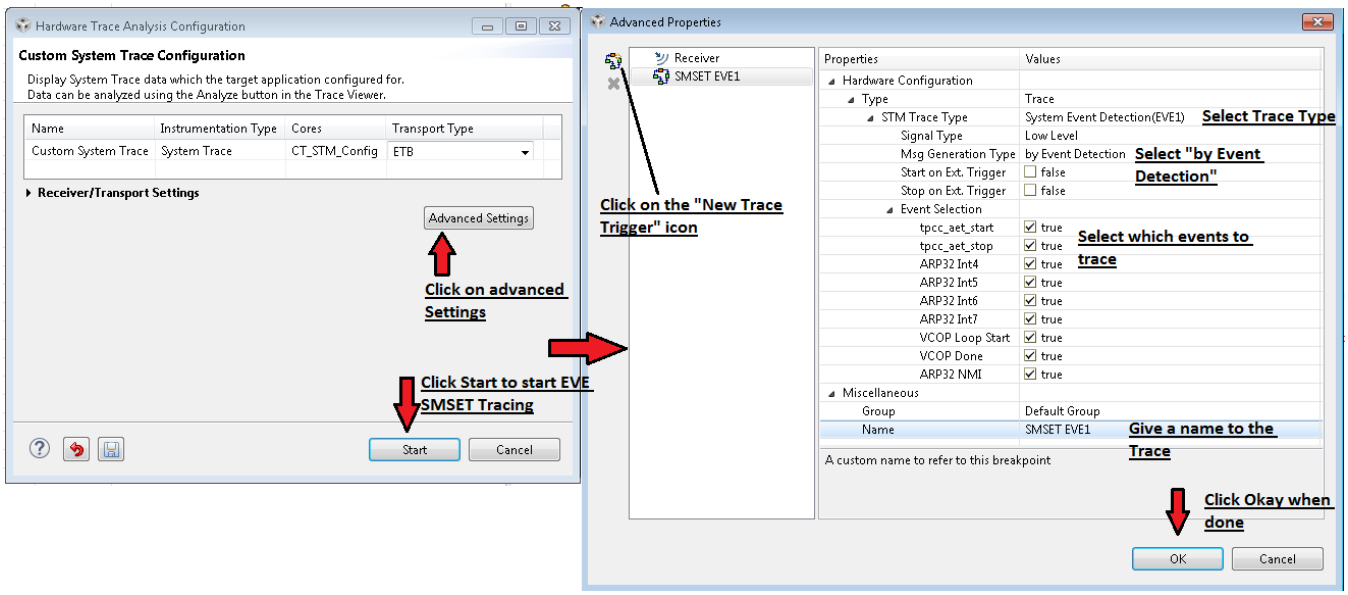



**Figure 31. EVE SMSET Hardware Trace Analysis Configuration and Advanced Properties Settings**

5. Update the properties to select the trace type, message generation type, and event selection to configure the *EVE SMSET*.
6. Repeat the same for *EVE2/3/4* as applicable.
7. Once the *SMSET* is configured, the code on the *EVE* must be run to allow the Trace Viewer to capture events.
8. The Trace Viewer is updated when the *EVEs* stop running or the Stop Trace Collection button is clicked.

   An example output of the Trace Viewer correlated with theoretical calculation of execution time:

   a. 7×7 Gaussian filter of a 768×512 image divided into 128×64 blocks and apportioned across 4 *EVE* cores.

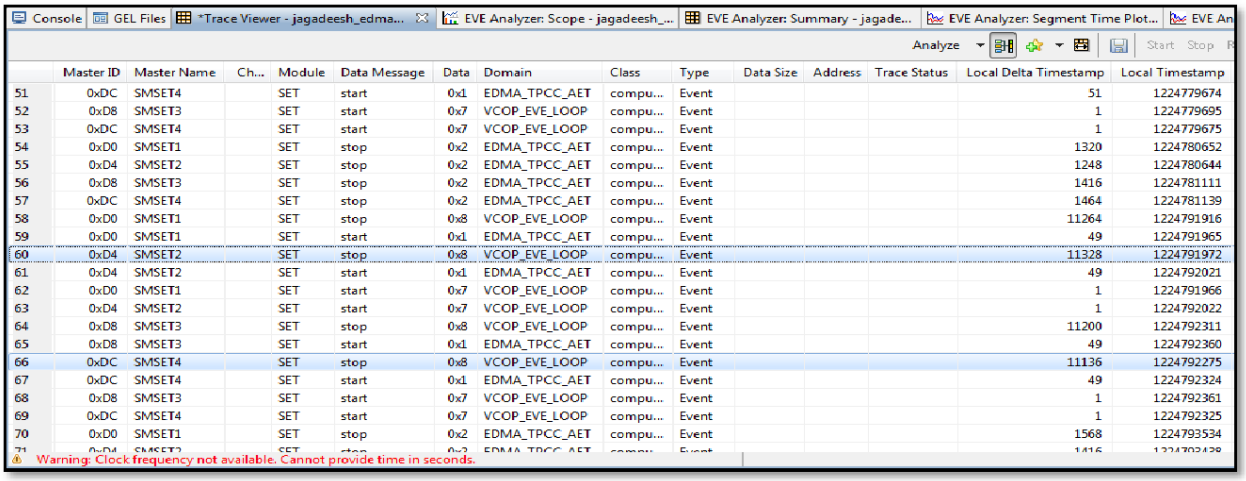
   b. Four EVE's working horizontally with the first *EVE* is allocated with 1/4$^{th}$ of the data, and the next *EVE* is allocated with 1/4$^{th}$ of the data, and so on.

   c. From a compute, expect at least 128×64×49/16×1/2= 12,544 *SMSET* (ARP32) cycles with 19 cycles of pipeline overhead + 6-7 cycles of parameter decode+6 cycles of command decode= 12,544+32= 12,576.

   d. From a DMA, you must bring in a 130×66 and produce a 128×64= 16,722 bytes, which at 5.88 bytes per *VCOP* cycle must be= 16,772×1÷5.88×1÷2= 1,445 cycles.


Copyright © 2016–2017, Texas Instruments Incorporated

Figure 32 shows the example EVE SMSET trace viewer output.



Console | GEL Files | *Trace Viewer - jagadeesh_edma... | EVE Analyzer: Scope - jagadeesh_... | EVE Analyzer: Summary - jagade... | EVE Analyzer: Segment Time Plot... | EVE An

Analyze | Start | Stop

| | Master ID | Master Name | Ch... | Module | Data Message | Data | Domain | Class | Type | Data Size | Address | Trace Status | Local Delta Timestamp | Local Timestamp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 51 | 0xDC | SMSET4 | | SET | start | 0x1 | EDMA_TPCC_AET | compu... | Event | | | | 51 | 1224779674 |
| 52 | 0xD8 | SMSET3 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224779695 |
| 53 | 0xDC | SMSET4 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224779675 |
| 54 | 0xD0 | SMSET1 | | SET | stop | 0x2 | EDMA_TPCC_AET | compu... | Event | | | | 1320 | 1224780652 |
| 55 | 0xD4 | SMSET2 | | SET | stop | 0x2 | EDMA_TPCC_AET | compu... | Event | | | | 1248 | 1224780644 |
| 56 | 0xD8 | SMSET3 | | SET | stop | 0x2 | EDMA_TPCC_AET | compu... | Event | | | | 1416 | 1224781111 |
| 57 | 0xDC | SMSET4 | | SET | stop | 0x2 | EDMA_TPCC_AET | compu... | Event | | | | 1464 | 1224781139 |
| 58 | 0xD0 | SMSET1 | | SET | stop | 0x8 | VCOP_EVE_LOOP | compu... | Event | | | | 11264 | 1224791916 |
| 59 | 0xD0 | SMSET1 | | SET | start | 0x1 | EDMA_TPCC_AET | compu... | Event | | | | 49 | 1224791965 |
| 60 | 0xD4 | SMSET2 | | SET | stop | 0x8 | VCOP_EVE_LOOP | compu... | Event | | | | 11328 | 1224791972 |
| 61 | 0xD4 | SMSET2 | | SET | start | 0x1 | EDMA_TPCC_AET | compu... | Event | | | | 49 | 1224792021 |
| 62 | 0xD0 | SMSET1 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224791966 |
| 63 | 0xD4 | SMSET2 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224792022 |
| 64 | 0xD8 | SMSET3 | | SET | stop | 0x8 | VCOP_EVE_LOOP | compu... | Event | | | | 11200 | 1224792311 |
| 65 | 0xD8 | SMSET3 | | SET | start | 0x1 | EDMA_TPCC_AET | compu... | Event | | | | 49 | 1224792360 |
| 66 | 0xDC | SMSET4 | | SET | stop | 0x8 | VCOP_EVE_LOOP | compu... | Event | | | | 11136 | 1224792275 |
| 67 | 0xDC | SMSET4 | | SET | start | 0x1 | EDMA_TPCC_AET | compu... | Event | | | | 49 | 1224792324 |
| 68 | 0xD8 | SMSET3 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224792361 |
| 69 | 0xDC | SMSET4 | | SET | start | 0x7 | VCOP_EVE_LOOP | compu... | Event | | | | 1 | 1224792325 |
| 70 | 0xD0 | SMSET1 | | SET | stop | 0x2 | EDMA_TPCC_AET | compu... | Event | | | | 1568 | 1224793534 |

Warning: Clock frequency not available. Cannot provide time in seconds.

**Figure 32. Example EVE SMSET Trace Viewer Output**

Click the Analyze → EVE analyzer option to graphically analyze the output from the *EVE SMSET* trace. Figure 33 shows the *VCOP* loop for each *EVE* that runs for approximately 12,600 ARP32 cycles and the *EDMA* that runs for approximately 1,500 cycles.

Figure 33 shows the example EVE analyzer output.

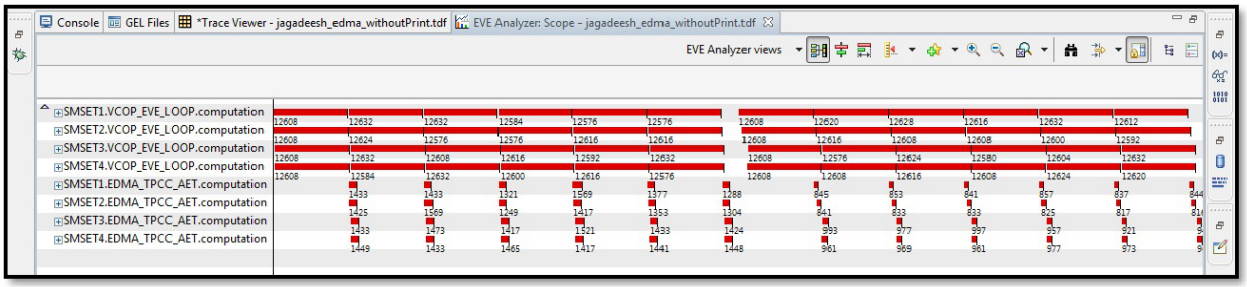



**Figure 33. Example EVE Analyzer Output**

**NOTE:** The *EVE AETCTL* register bit fields *STRTEVT* and *ENDINT* must be populated with the *DMA* channel numbers from the software to capture the beginning of the series of the *EDMA* transactions, and the end of the *EDMA* transactions.


Copyright © 2016–2017, Texas Instruments Incorporated

### 6.3.2 EVE Software Messaging

The STM and SDTI Target Library exports a *C* interface to the customer's application. You can leverage these functions to send software messages to the trace by making calls into these APIs. These SM messages and SET events are shown by the CCS Trace Viewer utility. Link to download STM Libraries (STMLib) is http://processors.wiki.ti.com/index.php/CToolsLib#Download. An example software sequence to use the EVE software messaging from the ARP32 CPU core is:

```
/*--------------------------------------------------------------------------*/
/*  Standard header includes for c environment.                             */
/*--------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "stdlib.h"
#include "StmLibrary.h"
#include "arp32.h"


int test_main()
{
    STMHandle * pSTMHandle;
#ifdef _CIO
    STMBufObj STMBUFInfo = { NULL,    // pFileName: if NULL all STM output directed to STDOUT
                             false,   //fileAppend: if true new data appended to pFilename
                             "EVE"    //pMasterId: char * to master name
                            };
    STMBufObj * pSTMBufInfo = &STMBUFInfo;
#else
    STMBufObj * pSTMBufInfo = NULL;
#endif
    STMConfigObj STMConfigInfo;
    int stmChn = 1;
    int num = 0x12345678;
    unsigned short data[] = {250,251,252,253};

    printf("STM Example Start");

    STMConfigInfo.optimized_prinf =  true;
    STMConfigInfo.STM_BaseAddress = 0x40089000;
    STMConfigInfo.STM_ChannelResolution = 8;
    STMConfigInfo.pCallBack = NULL;

    pSTMHandle = STMXport_open(pSTMBufInfo, &STMConfigInfo );

#ifndef _COMPACT
    STMXport_printf(pSTMHandle, stmChn, "STMXport_printf: Hello World %d", num);
#endif

    STMXport_logMsg0(pSTMHandle, stmChn, "logMsg0: Hello World");

    STMXport_logMsg1(pSTMHandle, stmChn, "logMsg1: Log message single arg -
 STM base address is %x",
                                                 STMConfigInfo.STM_BaseAddress);

    STMXport_logMsg2(pSTMHandle, stmChn, "logMsg2: Log message two args -
 num is %d num+1 is %d\0",
                                                 num, num+1 );

    STMXport_logMsg(pSTMHandle, stmChn, "logMsg: Log Message > two args -
 num is %d, num+1 is %d num+2 is %d\0",
                                                 num, num+1, num+2);

    STMXport_logMsg0(pSTMHandle, stmChn, "logMsg0: Dumping 4 shorts\0");
    STMXport_putBuf (pSTMHandle, stmChn, (void *)data, eShort, sizeof(data)/sizeof(short);

    STMXport_logMsg0(pSTMHandle, stmChn, "logMsg0: Dumping 1st byte of num, 1st short of num, and
```


Copyright © 2016–2017, Texas Instruments Incorporated

```
num\0");
    STMXport_putByte (pSTMHandle, stmChn, (char)num);
    STMXport_putShort (pSTMHandle, stmChn, (short)num);
    STMXport_putWord (pSTMHandle, stmChn, num);

#ifndef _COMPACT
    STMXport_printf(pSTMHandle, stmChn, "%s->Line Number %d\0",__FUNCTION__,__LINE__);
#endif

    //STMXPort_close sends extra message to get all user messages to ETB
    STMXport_close(pSTMHandle);
    printf("STM Example Done");
    return(1);
}
```

The steps to enable trace on the *EVE SMSET* for the software messaging are the same as the sequence described in Section 6.3.1. The output obtained from the example sequence:

| | Master ID | Master Name | Ch... | Module | Data Message | Data | Domain | Class |
|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | 0xFD | | |
| 16 | 0x4C | EVE4 P1 | 0x4 | STM | logMsg0: Dumping 1st byte of num, 1st short of num, and num | | SW | Target function: Printf() |
| 17 | 0x4C | EVE4 P1 | 0x4 | STM | | 0x78 | SW | Target function: PutBuf() |
| 18 | 0x4C | EVE4 P1 | 0x4 | STM | | 0x5678 | SW | Target function: PutBuf() |
| 19 | 0x4C | EVE4 P1 | 0x4 | STM | | 0x12345678 | SW | Target function: PutBuf() |
| 20 | 0x4C | EVE4 P1 | 0x4 | STM | SMSET_SWMSG_TestFunction->Line Number 135 | | SW | Target function: Printf() |
| 21 | 0x40 | EVE1 P1 | 0x1 | STM | STMXport_printf: Hello World 305419896 | | SW | Target function: Printf() |
| 22 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg0: Hello World | | SW | Target function: Printf() |
| 23 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg1: Log message single arg - STM base address is 40089000 | | SW | Target function: Printf() |
| 24 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg2: Log message two args - num is 305419896 num+1 is 305419897 | | SW | Target function: Printf() |
| 25 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg: Log Message > two args - num is 305419896, num+1 is 305419897 num+2 is 305419898 | | SW | Target function: Printf() |
| 26 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg0: Dumping 4 shorts | | SW | Target function: Printf() |
| 27 | 0x40 | EVE1 P1 | 0x1 | STM | | 0xFA | SW | Target function: PutBuf() |
| 28 | | | | | | 0xFB | | |
| 29 | | | | | | 0xFC | | |
| 30 | | | | | | 0xFD | | |
| 31 | 0x40 | EVE1 P1 | 0x1 | STM | logMsg0: Dumping 1st byte of num, 1st short of num, and num | | SW | Target function: Printf() |
| 32 | 0x40 | EVE1 P1 | 0x1 | STM | | 0x78 | SW | Target function: PutBuf() |
| 33 | 0x40 | EVE1 P1 | 0x1 | STM | | 0x5678 | SW | Target function: PutBuf() |
| 34 | 0x40 | EVE1 P1 | 0x1 | STM | | 0x12345678 | SW | Target function: PutBuf() |
| 35 | 0x40 | EVE1 P1 | 0x1 | STM | SMSET_SWMSG_TestFunction->Line Number 135 | | SW | Target function: Printf() |
| 36 | 0x44 | EVE2 P1 | 0x2 | STM | STMXport_printf: Hello World 305419896 | | SW | Target function: Printf() |
| 37 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg0: Hello World | | SW | Target function: Printf() |
| 38 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg1: Log message single arg - STM base address is 40089000 | | SW | Target function: Printf() |
| 39 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg2: Log message two args - num is 305419896 num+1 is 305419897 | | SW | Target function: Printf() |
| 40 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg: Log Message > two args - num is 305419896, num+1 is 305419897 num+2 is 305419898 | | SW | Target function: Printf() |
| 41 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg0: Dumping 4 shorts | | SW | Target function: Printf() |
| 42 | 0x44 | EVE2 P1 | 0x2 | STM | | 0xFA | SW | Target function: PutBuf() |
| 43 | | | | | | 0xFB | | |
| 44 | | | | | | 0xFC | | |
| 45 | | | | | | 0xFD | | |
| 46 | 0x44 | EVE2 P1 | 0x2 | STM | logMsg0: Dumping 1st byte of num, 1st short of num, and num | | SW | Target function: Printf() |
| 47 | 0x44 | EVE2 P1 | 0x2 | STM | | 0x78 | SW | Target function: PutBuf() |
| 48 | 0x44 | EVE2 P1 | 0x2 | STM | | 0x5678 | SW | Target function: PutBuf() |
| 49 | 0x44 | EVE2 P1 | 0x2 | STM | | 0x12345678 | SW | Target function: PutBuf() |
| 50 | 0x44 | EVE2 P1 | 0x2 | STM | SMSET_SWMSG_TestFunction->Line Number 135 | | SW | Target function: Printf() |
| 51 | 0x48 | EVE3 P1 | 0x3 | STM | rt | | SW | Target function: Printf() |
| 52 | 0x48 | EVE3 P1 | 0x3 | STM | 305419896 | | SW | Target function: Printf() |
| 53 | 0x48 | EVE3 P1 | 0x3 | STM | rld | | SW | Target function: Printf() |
| 54 | 0x48 | EVE3 P1 | 0x3 | STM | 12345678 | | SW | Target function: Printf() |

**Figure 34. Example Trace Viewer Output of EVE Software Messaging**


Copyright © 2016–2017, Texas Instruments Incorporated

# 7 Throughput and Data Traffic Profiling

The L3 interconnect supports a built-in non-intrusive performance monitoring feature by implementing a statistics collector (STATCOLL) component, which computes traffic statistics within a user-defined window and periodically reports through the MIPI-STM interface.

## 7.1 L3 Statistics Collector

CCS supports multiple use cases of the L3 statistic collector that helps understand the traffic generated by initiators and the traffic reaching slaves of the L3 interconnect while the application is running in a non-intrusive fashion. The list of supported use cases are shown below.

**Average Burst Length:** Calculate the average size (bytes) in a burst transaction. A normal single word access results in 4 bytes of data per burst (transaction access). DMA transaction, in general, can transfer data up to 128 bytes per transaction, in general, can transfer data up to 128 bytes per transaction. This calculation helps to optimize data transaction and improves bus use. For example, to transfer a large buffer using DMA efficiency, the average burst length for DMA must be close to 128 bytes.

**Throughput per Sampling Period:** Calculate number of transaction (bytes) per sampling window (cycles). It is commonly used in bus bandwidth analysis. You can convert the results to bytes per second by multiplying the above result by statistic collector's operation frequency. For example, on the TDA2x, DRA7x, and the TDA3x (statistic collector is operated at 266 MHz) if the result is 4,095 (bytes per sampling period) for a sampling period of 0×FFF the conversion yields 4,095×266 M÷0×FF= 266 Mbytes per second.

**Link Occupancy for All Transactions:** Calculate the percentage of time the module is in active (non-idle) state. When the module is in active state, it may be transferring data, doing arbitration, or analyzing header.

**Arbitration Conflicts for Request:** Calculate the percentage of time the module is in active (non-idle) state. When the module is in active state, it can be transferring data, doing arbitration, or analyzing header.

**Initiator Busy on Response:** Calculate the percentage of time the module is in busy state so that it cannot accept any read data from a target.

**Underflow on Request:** This statistic allows determination of the cycles during a write transaction when the initiator is not able to send data to a target at the rate at which the target can accept it.

**NOTE:** This support may not be available at all probe types.

**Histogram of Pressure Distribution:** Amount of data bytes for low and high priority transferred during the sample window period. Show usage of N counters to determine traffic priority attribute of each burst per sampling period are in a range of 0..n, n+1…m, m+1…k for different transaction types.

**Average Latency Distribution:** Calculate average read and write latency for each data packet. It is only available on the NTTP probe type.

**Histogram of Latency Distribution:** Show usage of N counters to determine latency of read and write transaction per sampling period are in a range of 0..n, n+1…m, m+1…k for different transaction types.

In order to enable L3 statistic-based analysis, select Memory Throughput Analysis or Custom System Trace. This opens the Hardware Trace Analysis pop up window. Based on the L3 statistic use case, further configuration in the Advanced Settings options may vary.

In the following subsections, the ways to measure the Throughput per Sampling Period and Average Latency. The configuration for the rest of the use cases is similar.


Copyright © 2016–2017, Texas Instruments Incorporated

Figure 35 shows the steps to enable throughput and data traffic profiling.

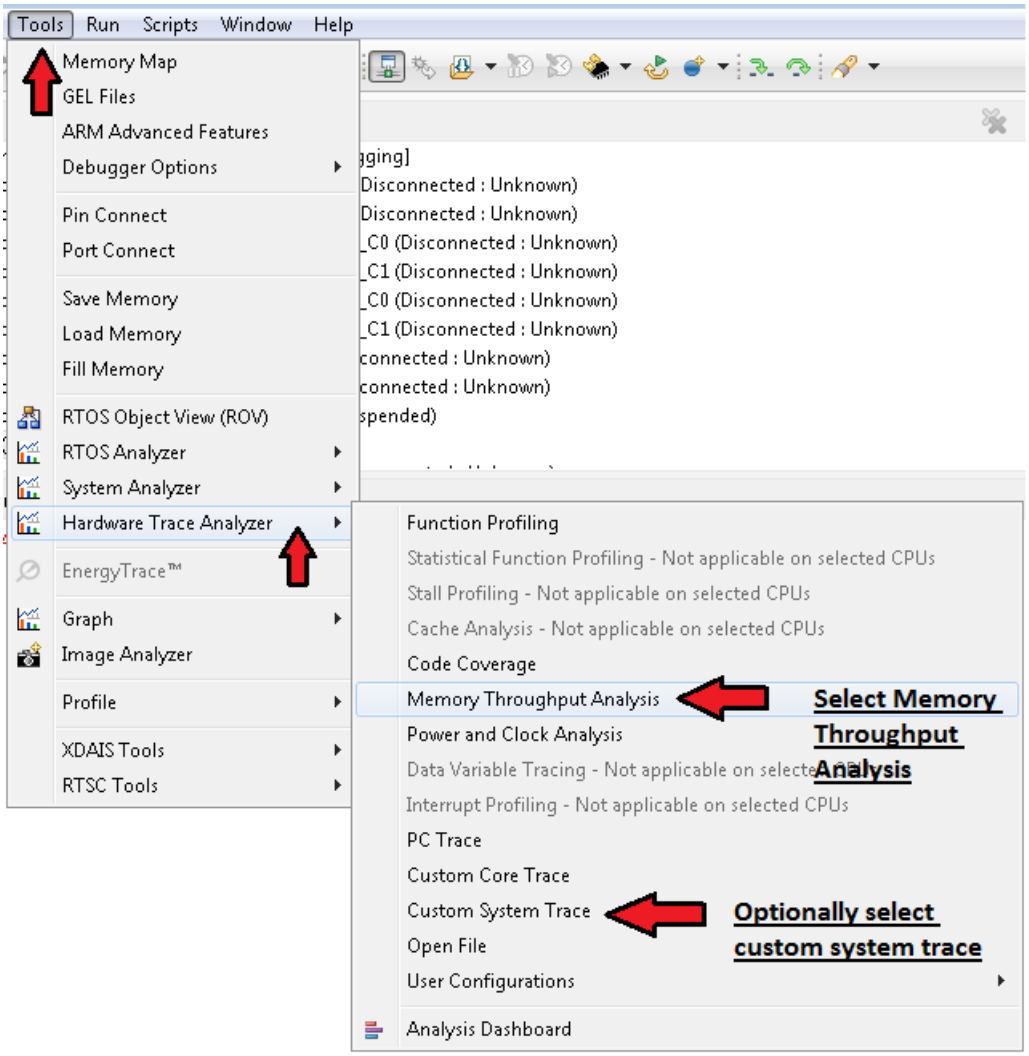



**Figure 35. Steps to Enable Throughput and Data Traffic Profiling**


Copyright © 2016–2017, Texas Instruments Incorporated

### 7.1.1 Throughput

For throughput a default configuration of the EMIF1 and EMIF2 SYS ports is present. You can choose the Subsystem Type to another L3 slave or initiator to be able to profile the Throughput per sampling period. You can also set additional filters with respect to reads only, reads and writes and writes only. Figure 36 shows the throughout profiling Hardware Trace Analysis Configuration and Advanced Properties settings.

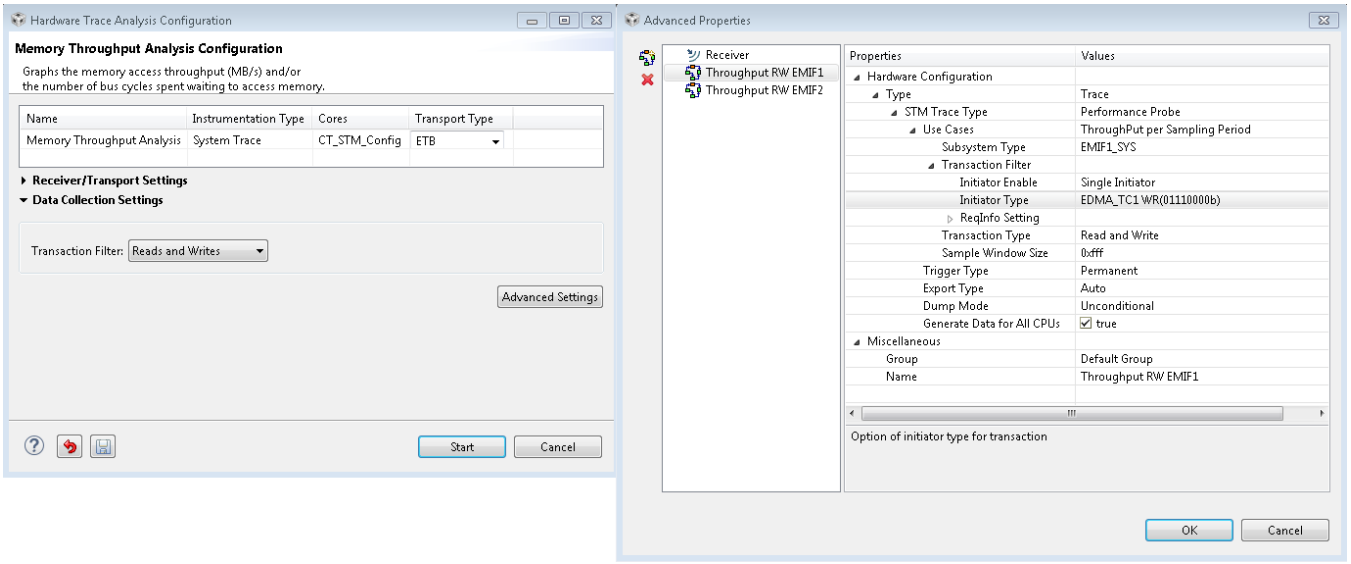



**Figure 36. Throughput Profiling Hardware Trace Analysis Configuration and Advanced Properties Settings**

Additional transaction filters can be applied, which allows filtering traffic based on an initiator.

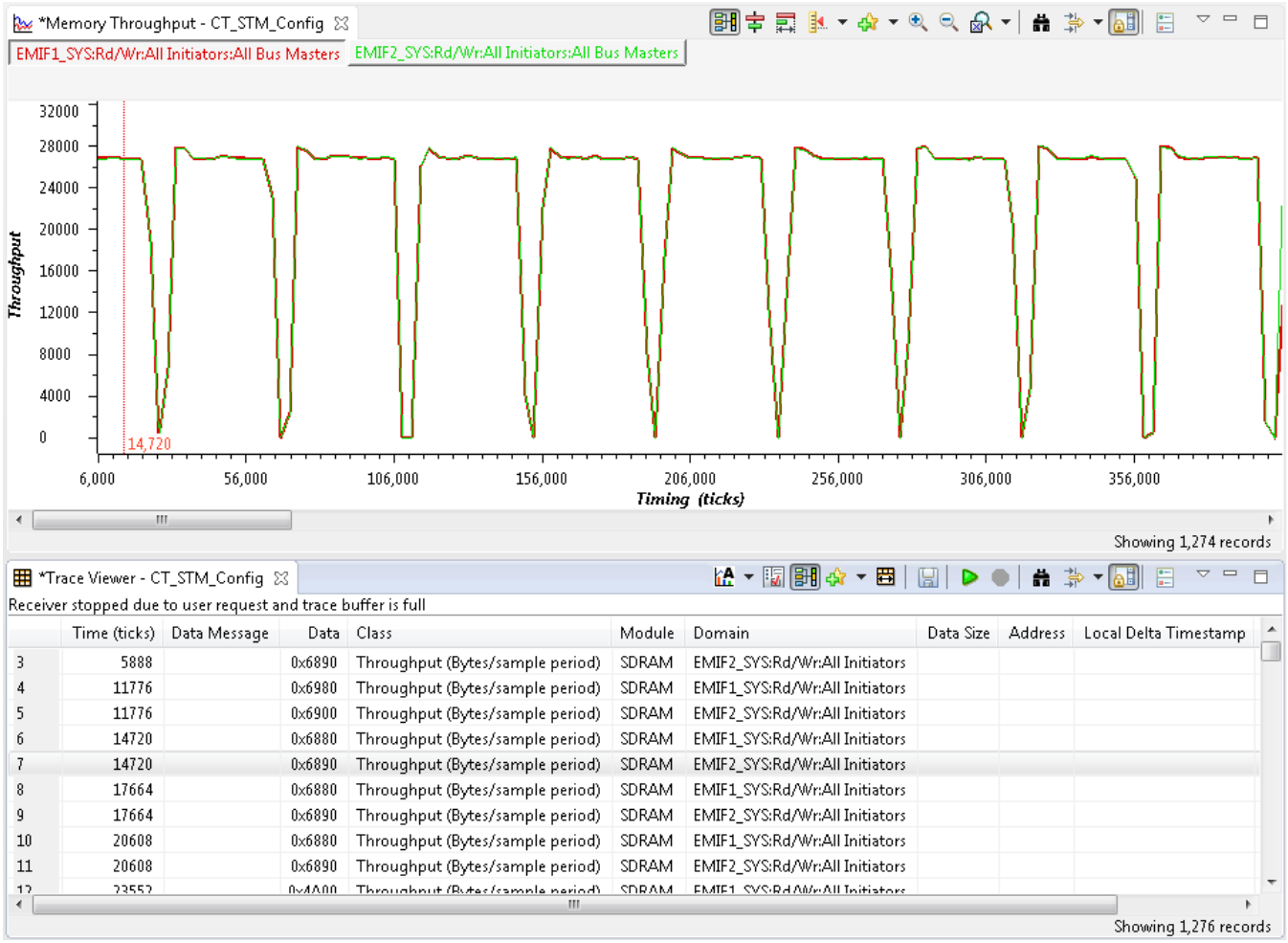
The choice of the sampling window size has a direct impact on the resolution of capturing bandwidth peaks and calculating the device throughput. The number gives the number of L3 clock cycles after which the data is sent to the device STM. Thus, to calculate the time interval the Sampling window is divided by the L3 frequency (266 MHz).

Both reads and writes are captured in this example. The Trace Viewer gives the bytes transferred every sampling widow.

The Y-axis of the memory throughput gives the data is bytes per sampling period. To understand the throughput, the number in the Y-Axis must be multiplied with the L3 frequency and divided by the sampling window size. Every point in the X-axis corresponds to one sampling interval. Thus, the ticks correspond to the number of samples.


Copyright © 2016–2017, Texas Instruments Incorporated

An example output for an EDMA transfer to interleaved EMIF is shown in Figure 37.




| | Time (ticks) | Data Message | Data | Class | Module | Domain | Data Size | Address | Local Delta Timestamp |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 5888 | | 0x6890 | Throughput (Bytes/sample period) | SDRAM | EMIF2_SYS:Rd/Wr:All Initiators | | | |
| 4 | 11776 | | 0x6980 | Throughput (Bytes/sample period) | SDRAM | EMIF1_SYS:Rd/Wr:All Initiators | | | |
| 5 | 11776 | | 0x6900 | Throughput (Bytes/sample period) | SDRAM | EMIF2_SYS:Rd/Wr:All Initiators | | | |
| 6 | 14720 | | 0x6880 | Throughput (Bytes/sample period) | SDRAM | EMIF1_SYS:Rd/Wr:All Initiators | | | |
| 7 | 14720 | | 0x6890 | Throughput (Bytes/sample period) | SDRAM | EMIF2_SYS:Rd/Wr:All Initiators | | | |
| 8 | 17664 | | 0x6880 | Throughput (Bytes/sample period) | SDRAM | EMIF1_SYS:Rd/Wr:All Initiators | | | |
| 9 | 17664 | | 0x6890 | Throughput (Bytes/sample period) | SDRAM | EMIF2_SYS:Rd/Wr:All Initiators | | | |
| 10 | 20608 | | 0x6880 | Throughput (Bytes/sample period) | SDRAM | EMIF1_SYS:Rd/Wr:All Initiators | | | |
| 11 | 20608 | | 0x6890 | Throughput (Bytes/sample period) | SDRAM | EMIF2_SYS:Rd/Wr:All Initiators | | | |
| 12 | 23552 | | 0x4A00 | Throughput (Bytes/sample period) | SDRAM | EMIF1_SYS:Rd/Wr:All Initiators | | | |

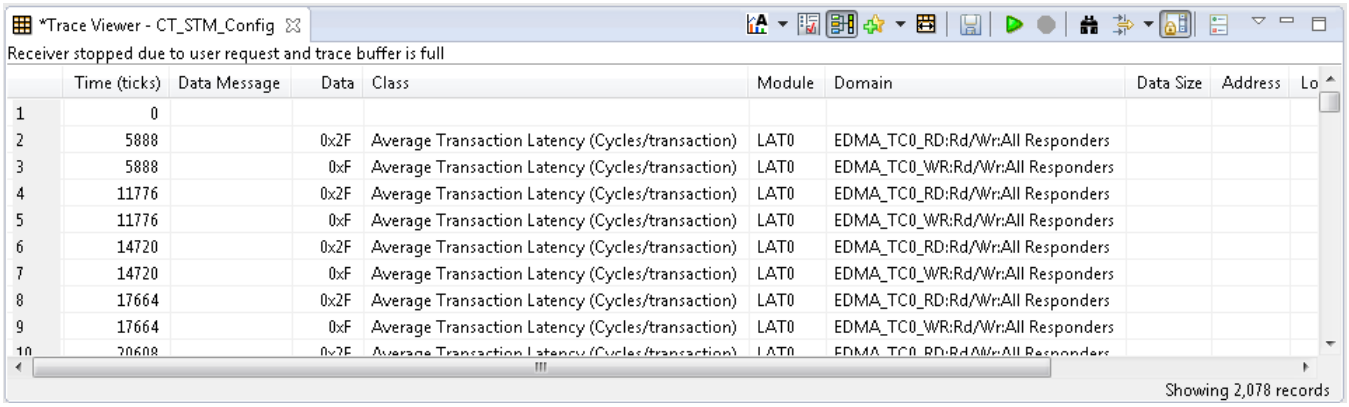
**Figure 37. Example Output of the Throughput Profiling for an EDMA Transfer to Interleaved EMIF**


Copyright © 2016–2017, Texas Instruments Incorporated

### 7.1.2 Average Latency

The average latency can be measured by choosing Average Latency Distribution use case in the advanced properties tab. The average latency is measured over a sampling window determined by the sampling window size configuration.

A sample output of the average latency measurement for an EDMA transaction from DDR to OCMC RAM is shown in Figure 38. The output is represented in the L3 cycles. In Figure 38, the output took an average of 0x2F cycles for a read request from TC0 to get a response from DDR and 0xF cycles for a write request from TC0 to OCMC RAM to get a success response.




| | Time (ticks) | Data Message | Data | Class | Module | Domain | Data Size | Address | Lo |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | | | | | |
| 2 | 5888 | | 0x2F | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_RD:Rd/Wr:All Responders | | | |
| 3 | 5888 | | 0xF | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_WR:Rd/Wr:All Responders | | | |
| 4 | 11776 | | 0x2F | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_RD:Rd/Wr:All Responders | | | |
| 5 | 11776 | | 0xF | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_WR:Rd/Wr:All Responders | | | |
| 6 | 14720 | | 0x2F | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_RD:Rd/Wr:All Responders | | | |
| 7 | 14720 | | 0xF | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_WR:Rd/Wr:All Responders | | | |
| 8 | 17664 | | 0x2F | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_RD:Rd/Wr:All Responders | | | |
| 9 | 17664 | | 0xF | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_WR:Rd/Wr:All Responders | | | |
| 10 | 20608 | | 0x2F | Average Transaction Latency (Cycles/transaction) | LAT0 | EDMA_TC0_RD:Rd/Wr:All Responders | | | |

**Figure 38. Sample Trace Viewer Output of an Average Transaction Latency of a DMA Transfer From DDR to OCMC**

## 7.2 OCP Watchpoint

The L3 interconnect provides functional probes embedded and attached to the following L3 targets:

- GPMC
- L4-PER
- L4-CFG
- DMM_P1 (DMM target port 1)
- DMM_P2 (DMM target port 2)
- OCMC RAM

The probes output are multiplexed together and then sent to the L3 interconnect debug port. A component called OCP-WP is used to collect data from functional probes and then transmit captured data to the STM module.

The OCP-WP provides the following main features. For all the features of the OCP WP, see the device-specific TRM.

- Monitoring the OCP traffic originated by all initiators that can access the selected target where the probe is attached.
- Filtering OCP monitoring bus traffic by:
  - Address range
  - Initiator-ID (see the L3 Interconnect specification for initiator-ID mapping)
  - Transaction type
  - Transaction qualifier


Copyright © 2016–2017, Texas Instruments Incorporated

Figure 39 shows the OCP Watch Point Hardware Trace Analysis Configuration and Advanced Properties settings.

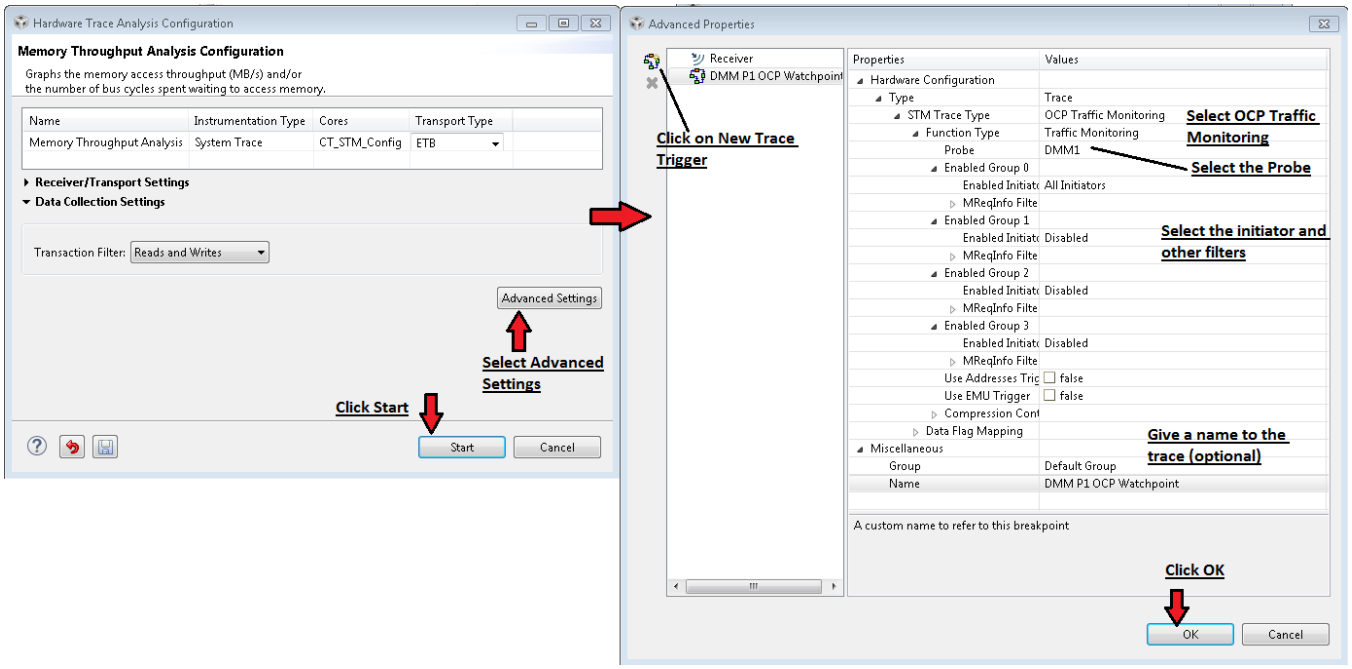
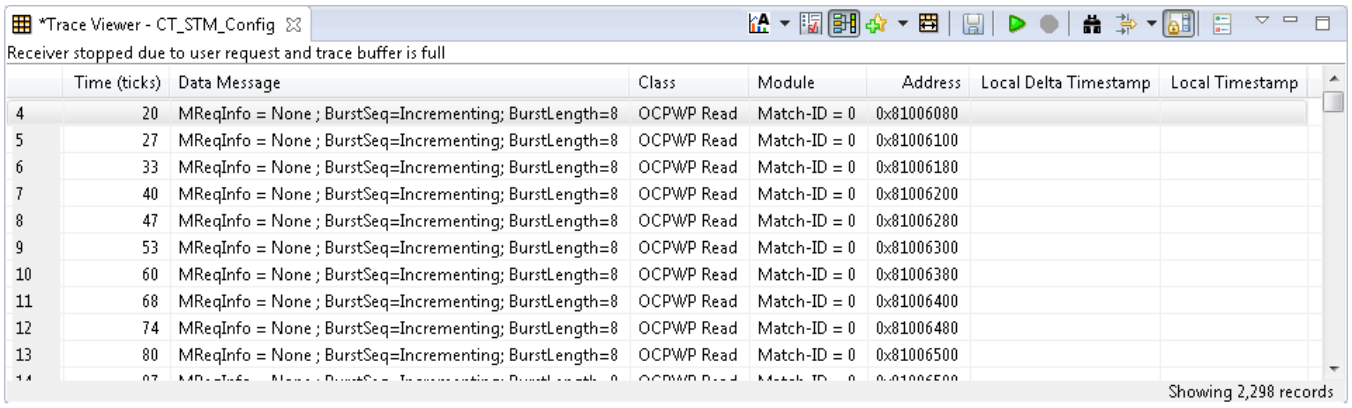



**Figure 39. OCP Watch Point Hardware Trace Analysis Configuration and Advanced Properties Settings**

Figure 40 shows the sample Trace Viewer output of the OCP watch point trace.




| | Time (ticks) | Data Message | Class | Module | Address | Local Delta Timestamp | Local Timestamp |
|---|---|---|---|---|---|---|---|
| 4 | 20 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006080 | | |
| 5 | 27 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006100 | | |
| 6 | 33 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006180 | | |
| 7 | 40 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006200 | | |
| 8 | 47 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006280 | | |
| 9 | 53 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006300 | | |
| 10 | 60 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006380 | | |
| 11 | 68 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006400 | | |
| 12 | 74 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006480 | | |
| 13 | 80 | MReqInfo = None ; BurstSeq=Incrementing; BurstLength=8 | OCPWP Read | Match-ID = 0 | 0x81006500 | | |

**Figure 40. Sample Trace Viewer Output of the OCP Watch Point Trace**

## 8 References

- Creating Device Initialization GEL Files
- CCSv6 Getting Started Guide wiki


Copyright © 2016–2017, Texas Instruments Incorporated

## Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

| Changes from A Revision (August 2016) to B Revision | Page |
|---|---|
| • Update was made in Section 2.2. | 9 |


Copyright © 2016–2017, Texas Instruments Incorporated

## IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2017, Texas Instruments Incorporated