AURIX Trace Overview and Use-Cases

Publish Date: 03/26/2019



This document and all documents accompanying it are copyrighted by iSYSTEM and all rights are reserved. Duplication of these documents is allowed for personal use. For every other case, written consent from iSYSTEM is required.

Copyright © iSYSTEM, AG. All rights reserved. All trademarks are property of their respective owners.

iSYSTEM is an ISO 9001 certified company

Table of Contents

1	Intro	duction	. 2
1	1.1 1.2 1.3 1.4 1.5 1.6 1.7	Overview of the AURIX On-Chip Trace Architecture Trace Multiplexer (MUX) Processor Observation Block (POB) Bus Observation Block (BOB) Multi-Core Cross Connect (MCX) Emulation Memory Debug Access Port (DAP)	. 2 . 3 . 4 . 5 . 7 14 14
	1.8	AURORA Gigabit (AGBT) Interface	15
	1.9	Usage of Initialization (.INI) Files	17
2	Trace	e Use-Cases	18
	2.1 2.2 2.3	Multi-Core OS Profiling via DAP Multi-Core OS & Program Trace via AGBT Function Specific Program Trace	19 32 49
3	Tech	nical support	51
	3.1 3.2	Online resources Contact	51 51

1 Introduction

The application note provides an overview of the on-chip trace architecture and capabilities of the Infineon AURIX micro controller family. Furthermore, this document discusses some common use-cases of the AURIX trace infrastructure in combination with the iSYSTEM On-Chip Analyzers iC500, iC5700 or iC6000.

1.1 Overview of the AURIX On-Chip Trace Architecture

The AURIX on-chip trace architecture is based on a central trace infrastructure, which can be connected to various on-chip system resources like CPUs or buses by means of multiplexors (MUX). This trace infrastructure is part of the so-called Multi-Core Debug System (MCDS) and is only available on special Emulation Devices.

Figure 1 shows a simplified block diagram of a TC3x emulation device, including MCDS on-chip trace infrastructure. The components within the grey box are only available on Emulation devices.



Figure 1: Simplified Block Diagram of a TC3x Emulation Device

The main components of the MCDS trace infrastructure are:

- Trace Multiplexer (MUX)
- Processor Observation Block (POB)
- Bus Observation Block (BOB)
- Multi-Core Cross Connect (MCX)
- Memory Controller (DMC)
- Emulation Memory (EMEM)
- Debug Access Port (DAP), optional DAPE on TC3x on Emulation Devices
- Optional AURORA Gigabit (AGBT) Interface

In the following chapters the individual MCDS components are describes in more details.

1.2 Trace Multiplexer (MUX)

The MUX allows to connect the POBs to the various TriCore CPUs implemented on the chip. The TC2x AURIX generation implements two POBs. However, some TC2x derivatives (TC2xxTx) include three cores. Thus, in these cases only two out of three cores can be connected to a POB.

There are additional multiplexers attached to the Bus Observation Block (BOB) connected to the System Resource Interconnect (SRI). These multiplexers allow to select specific SRI slaves to be connected to the SRI BOB, e.g. LMU RAM or specific CPU Local RAMs (CPU0 PSPR, DSPR...).

Within the winIDEA trace configuration dialog, the Trace Multiplexers are represented by the configuration section depicted in Figure 2 (menu: "Analyzer Configuration – Hardware – Configure... - MCDS").

MCDS	TriCore X	TriCore Y	SRI	SPB	MCX	iNET				
– Trig Tri	ger gger Positio	n B	egin	•	~					
-MUX Wi	MUX Which SRI slave is seen by SRI1 LMU (LMU SRAM, EMEM)									
W	Which SRI slave is seen by SRI2					CPU0 (PSPR,DSPR)				
W	Which processor core is seen by POB X					CPU0				
W	hich process	sor core is s	een by P	OB Y	CPU1				\sim	
W	hich process	sor core is s	een by P	OB Z					\sim	

Figure 2: winIDEA Analyzer - MCDS Trace Multiplexer Configuration (TC2x)

The Processor Observation Block X (POB X) can, for instance, be connected to either nothing, to CPU0, CPU1 or CPU2 (see Figure 3).

Note: With the AURIX TC2x family only the two POBs X and Y are available, whereas the AURIX family TC3x implements three POBs X, Y and Z.

MUX		
Which SRI slave is seen by SRI1	LMU (LMU SRAM, EMEM)	~
Which SRI slave is seen by SRI2	CPU0 (PSPR,DSPR)	~
Which processor core is seen by POB X	CPU0	Y
Which are served and is seen by DOB V	Nothing	N
which processor core is seen by POB f	CPU0	
Which processor core is seen by POB 7	CPU1	
	CPU2	

Figure 3: winIDEA Analyzer – POB X Processor Selection Options (TC2x)

The Bus Observation Block connected to the SRI (BOB SRI 1/2) can be hooked up to various SRI slaves, for instance the processor local RAMs or LMU RAM (see Figure 4).

MUX	
Which SRI slave is seen by SRI1	LMU (LMU SRAM, EMEM)
Which SRI slave is seen by SRI2	CPU1 (PSPR,DSPR) CPU2 (PSPR,DSPR)
Which processor core is seen by POB \ensuremath{X}	LMU (LMU SRAM, EMEM) PMU0 DFlash, BootROM, CTRL Reg
Which processor core is seen by POB \ensuremath{Y}	PMU0 PFlash0 PMU0 PFlash1
Which processor core is seen by POB Z	XBar

Figure 4: winIDEA Analyzer – POB SRI1 SRI Slave Selection Options (on TC277TF)

1.3 Processor Observation Block (POB)

Each of the POBs can be connected to one of the TriCore CPUs. The POB can monitor the instruction execution and the data transactions performed by the CPU. Thus, a POB can generate trace messages for program flow trace and for data access trace.

In addition, a POB offers various types of hardware comparators which allow to limit/focus trace to particular areas of interest, e.g. limit data access trace of specific data address ranges or limit program trace to specific program code areas (e.g. functions).

The POB hardware configuration options are represented in winIDEA by a configuration dialog as shown in Figure 5.

on (double click to edit)	Event (double click to edit)	Trigger (double click to edit)
	EVT0 ptu_trig_0 EVT1 - EVT2 - EVT3 - EVT5 - EVT6 - EVT7 - EVT8 - EVT12 - EVT13 - EVT12 - EVT13 - EVT14 - EVT15 -	core_crevt core_trig1 core_trig1 core_trig2 core_trig3 core_trig4 core_trig5 core_trig6 core_trig7 core_trig7 core_trig7 core_trig7 core_trig7 core_trig6 core_trig7 core_trig7 core_eachine pbu_trig_1 ptu_trig_2 ptu_trig_3 ptu_trig_4 ptu_trig_5 ptu_trig_6 ptu_trig_7 atWAYS ptu_trig_7 atWAYS otu_trig0 atWAYS otu_trig1 atWAYS otu_trig2 atWAYS dtu_ea_trig2 dtu_ea_trig3 atWAYS dtu_ea_trig5 atWAYS dtu_ea_trig6 atWAYS dtu_ea_trig7 atWAYS dtu_ea_trig7 atWAYS dtu_ea_trig7 atWAYS <
U_TC program trace enable Advanced		dtu_dat_trig_1 ALWAYS dtu_dat_trig_2 ALWAYS dtu_dat_trig_3 ALWAYS dtu_acc_wr dtu_acc_wr

Figure 5: Manual Trace Configuration Dialog for a Processor Observation Block (POB X)

A manual configuration process of a POB is basically done from right to left. The right most column lists all available trace triggers of a POB. Such triggers are generated by hardware comparators implemented in a POB. The comparators are typically configured to generate a trigger on an address or data match, i.e. when the CPU executes an instruction located at a specific address, when the CPU accesses (read/write) specific memory locations or when the CPU read/writes a specific data value to/from memory.

A trigger can then be mapped to one or multiple events. Events can also be formed by an logical AND combination of multiple triggers (e.g. write access to a specific address AND write of a specific data value).

An event can finally be connected to one or multiple actions. Such actions can for instance be the start of program trace or the capture of an address and value of a data write access.

Figure 5 illustrates this concepts based on two examples.

Example 1 (blue):

Use-case:

Program trace starts once the CPU connected to POB X executes the first instruction of the function BswM_MainFunction.

POB X Configuration:

The trigger is generated by using an address comparator of the POB Program Trace Unit (PTU). This address comparator monitors the CPU Instruction Pointer (IP). As soon as the IP matches the address value of BswM_MainFunction, the ptu_trig_0 is asserted.

The ptu_trig_0 is mapped to Event EVT0.

Event EVTO is routed to the Action ptu_enable, thus Program Trace Unit (PTU) gets enabled (generates program trace messages) when EVTO is active.

Example 2 (red):

Use-Case:

Data trace records all write access to the global variable isystem_trace, performed be the CPU connected to POB X.

POB X Configuration:

The trigger is generated by using an address comparator of the POB Data Trace Unit (DTU). This address comparator monitors the addresses of data read/write transactions of the CPU connected to POB X. When the data read/write address matches the address (range) of the global variable isystem_trace the dtu_ea_trig_0 trigger is asserted.

The dtu_ea_trig_0 trigger is mapped t Event EVT10.

Event EVT10 is routed to the Actions dtu_wdat and dtu_wadr, thus the Data Trace Unit (DTU) of POB X captures the data write data value (wdat) and the data write address (wadr) when EVT10 is active.

1.4 Bus Observation Block (BOB)

The MCDS implements two Bus Observation Blocks, BOB_SRI and BOB_SPB. The BOB_SPB is connected to the SPB peripheral bus. The two sub-blocks of the BOB_SRI can be connected to two slaves of the SRI bus.

The BOBs monitor the data transactions, performed by an bus master, over the SRI or SPB, respectively. Thus, a BOB can generate trace messages for data access trace.

In addition, a BOB offers various types of hardware comparators which allow to limit/focus trace to particular areas of interest, e.g. limit data access trace of specific data address ranges.

The BOB hardware configuration options are represented in winIDEA by a configuration dialog as shown in Figure 6.

ction (double click to edit)	Event (double click to edit)	Trigger (double click to edit)
ction (double click to edit) icu_enable icu_sync icu_sync itu_ucation itu_enable_1 itu_ucation itu_ucation	Event (double dick to edit) EVT0 dtu1_ea_trig_0 EVT2 - EVT3 - EVT4 - EVT5 - EVT6 - EVT7 - EVT8 - EVT9 - EVT10 - EVT10 - EVT11 - EVT12 - EVT13 - EVT13 - EVT15 dtu2_ea_trig_0 & dtu2_acc_trig_0 EVT15 dtu2_ea_trig_0 & dtu2_acc_trig_0 & dtu2_ac	Ingger (double dick to edit) dtu1_ea_fine dtu2_ea_fine dtu1_ea_trig_0 Isystem_trace) dtu1_ea_trig_2 ALWAYS dtu1_ea_trig_0 ALWAYS dtu1_ea_trig_0 ALWAYS dtu1_dat_trig_0 ALWAYS dtu1_dat_trig_1 ALWAYS dtu1_dat_trig_2 ALWAYS dtu1_act_trig_0 ALWAYS dtu1_act_trig_1 ALWAYS dtu1_act_trig_1 ALWAYS dtu1_act_trig_1 Access dtu1_acc_trig_1 Access dtu1_acc_trig_3 ALWAYS dtu2_ea_trig_0 [isystem_trace_runnable] dtu2_ea_trig_1 ALWAYS dtu2_ea_trig_1 ALWAYS dtu2_ea_trig_1 ALWAYS dtu2_ea_trig_2 ALWAYS dtu2_ea_trig_2 ALWAYS dtu2_ea_trig_2 ALWAYS dtu2_ea_trig_1 ALWAYS dtu2_ea_trig_2 ALWAYS dtu2_ea_trig_3 ALWAYS dtu2_acc_trig_3 Access dtu2_acc_trig_3 Access dtu2_acc_trig_3 Access

Trigger - [Advanced Coverage Trigger]

Figure 6: Manual Trace Configuration Dialog for a SRI Bus Observation Block (BOB_SRI)

A manual configuration process of a BOB is basically done from right to left. The right most column lists all available trace triggers of a BOB. Such triggers are generated by hardware comparators implemented in a BOB. The comparators are typically configured to generate a trigger on an address match, e.g. when a CPU or other SRI bus masters such as a DMA controller accesses (read/write) a specific memory locations. There are also other types of comparators available which monitor data values of SRI bus transactions or monitor which bus master performs the bus transaction.

A trigger can then be mapped to one or multiple events. Events can also be formed by and AND combination of multiple triggers (e.g. write access to a specific address AND write of a specific bus master).

An event can finally be connected to one or multiple actions. Such actions can for instance be the start of data trace, i.e. capturing address and value of a data write transactions.

Figure 6 illustrates this concepts based on two examples.

Example 1 (blue):

Use-case:

Data write trace of the global variable isystem_trace (using for instrumented OS profiling). In this example the variable isystem_trace resides in the LMU RAM, i.e. write transactions of all CPUs are performed via the SRI bus and thus can be monitored by the BOB_SRI.

BOB_SRI Configuration:

The trigger is generated by using an address comparator of the BOB_SRI Data Trace Unit 1 (DTU 1). This address comparator monitors the addresses of data read/write transactions via the SRI (performed by any SRI bus master). When the data read/write address matches the address (range) of the global variable isystem_trace the dtu1_ea_trig_0 trigger is asserted.

The dtu1_ea_trig_0 trigger is mapped to Event EVT0.

Event EVT0 is routed to the Actions dtu1_wdat and dtu1_wadr, thus the Data Trace Unit 1 (DTU 1) of BOB_SRI captures the data write data value (wdat) and the data write address (wadr) when EVT0 is active.

×

Example 2 (red):

Use-Case:

Data trace records all write access to the global variable isystem_trace_runnable, performed by CPU1. In this example the variable isystem_trace_runnable resides in the LMU RAM, i.e. write transactions of all CPUs are performed via the SRI bus and thus can be monitored by the BOB_SRI.

The bus master ID of CPU1 is 0x2.

In terms of AUTOSAR profiling, this means that only Runnables executed by CPU1 are profiled. The variable isystem_trace_runnable is used for instrumented Runnable trace of all CPUs.

BOB_SRI Configuration:

Two types of triggers are used in this example.

One trigger is generated by using an address comparator of the BOB_SRI Data Trace Unit 2 (DTU 2). This address comparator monitors the addresses of data read/write transactions via the SRI (performed by any SRI bus master). When the data read/write address matches the address (range) of the global variable isystem_trace_runnable the dtu2_ea_trig_0 trigger is asserted.

The second trigger is generated by using a special comparator type which can monitor which SRI bus master performs the SRI bus transaction (a so called "masked magnitude comparator"). Whenever the SRI bus transactions is performed by the bus master with ID=0x2 (i.e. CPU1) the dtu2_acc_trig_0 trigger is asserted.

Both triggers, dtu2_ea_trig_0 and dtu2_acc_trig0 are mapped to Event EVT15. The Event is asserted only in case both triggers are active, i.e. they form an AND combination. Thus, EVT15 is only asserted when CPU1 performs the data transaction to isystem_trace_runnable.

Event EVT15 is routed to the Actions dtu2_wdat and dtu2_wadr, thus the Data Trace Unit 2 (DTU 2) of POB_SRI captures the data write data value (wdat) and the data write address (wadr) when EVT15 is active.

1.5 Multi-Core Cross Connect (MCX)

The functionality of the MCX can be divided into three categories:

- 1. Time Stamp Message Generation
- 2. Message Storage Control
- 3. Event Counters
- 4. Trigger Feedback to Observation Blocks (POB/BOB)

1.5.1 Time Stamp Message Generation

In order to understand the time stamping concepts of the MCDS it is essential to understand that the trace messages delivered by the POBs and BOBs do not contain any timing information, i.e. there is no such thing as a time stamp within in each trace message.

Timing information is added by the MCX by adding dedicated time stamp messages.

Another important aspect is, that the whole message storage and time stamping approach is based on the underlying concept that trace data can be stored on-chip (in the Emulation Memory, EMEM) and read out by a trace tool at some later stage.

There are several time stamping concepts available, but typically either one of the following two concepts is used.

For both concepts the time stamps are derived from the counter structure depicted in Figure 7.



Figure 7: Time Stamp Counter Structure of the MCX

Time Stamping Concept 1 - Ticks:

The basic idea behind tick time stamps is an incremental timing information between two subsequent trace message, by means of an 8-bit wide "Tick" message. A "Tick" message represents one MCDS clock cycle. When two subsequence trace message are, for instance, generated four MCDS clock cycles apart from each other, the "space" in between these trace messages is filled with a " four Ticks" messages. If these is no new trace message for 255 MCDS clock cycles, then a so called "Multick" is automatically generated. When a trace tools reads out the on-chip trace buffer, it can incrementally derive the exact relative time for each trace message by adding the number of MCDS clock cycles between the trace messages as defined by the number of "Tick" messages

"Tick" based timestamp can be enabled via the Hardware, MCDS and MCX configuration dialogs as shown in Figure 8.

Hardware		M	anual	Trigger/r	ecorder c	onfiguratior	n – Configur	e MCX
Property V	alue	м	CDS	TriCore X	TriCore Y	SRI		
Recorder								
Start Ti	gger Immediately		Action	n (double cli	ick to edit)			
Recording Size Limit 1	GB		tsu_r	rel_en	-			
Trigger Position B	ain		tsu_r	rel_sync	-			
Timer Interpolation			tsu_a	abs_en	-			
Generate time synchronization messages	10		tsu_a	abs_sync	-			
Upload while campling]		wtu_	enable_0	-			
opioda while sampling	1	2	wtu_	enable_1	-			
			witu_	enable_2	-			
			wtu_	enable 4				
			wtu	enable 5	-			
			wtu	enable 6	-			
			wtu	enable_7	-			
Manual Trigger/recorder configuration	– Configure M	1CDS	wtu_	cnt_0	-			
Time stamps			wtu_	cnt_1	-			
			wtu_	cnt_2	-			
Assume source to be tick 📐 🗸			wtu_	cnt_3	-			
			wtu_	cnt_4	-			
TSUPRSCL 1 HEX			wtu_	cot 6				
			wtu	cnt 7	-			
Reference clock Main PLL 🗸			wtu	cnt 8	-			
			wtu	cnt 9	-			
			wtu	cnt_10	-			
			wtu	cnt_11	-			
			wtu_	cnt_12	-			
			wtu_	cnt_13	-			
			wtu_	cnt_14	-			
			wtu	cnt_15				

Figure 8: Hardware, MCDS and MCX Configuration Dialogs enabling "Tick" based Time Stamping

This "Tick" based time stamping is the most accurate trace method (timing resolution is MCDS clock, which is typically CPU clock divided by 2), but may consume more trace buffer compared to the TSU_REL based concept described below.

The resolution of a Tick cannot be configured, i.e. it is fixed to one MCDS clock cycle.

The Tick time stamp based trace timing reconstruction concept is also depicted in Figure 9.



Figure 9: Trace Timing Reconstruction using Tick Time Stamps (no Timer Interpolation)

Note: Tick based time stamping does not support a time correlation (synchronization) of the AURIX trace to other trace sources, such as another processor or an iC5700 Add-on module (CAN/LIN or ADIO).

Time Stamping Concept 2 - TSU_REL:

When using TSU_REL based time stamping, full 32-bit time stamps (contents of the TSUEMUCNT counter) are inserted into the trace stream/storage upon specific triggers. Such triggers can either be a periodic expiration of the TSUPRSCL prescaler (TSU_TC_TRIG) or other triggers generated by a POB or BOB and forwarded to the MCX.

There are two typical use-cases.

Use-case 1: Periodic TSU_REL generation

In this case the pre-scaler TSUPRSCL is used to generate a period TSU_TC_TRIG trigger. This trigger can be mapped to a MCX event and the event finally causes the action of generating a TSU_REL message (Relative Time Stamp Sync message). All trace massages which occur in between two consecutive TSU_REL messages are interpolated (equally distributed) between the TSU_REL messages by the trace tool. Therefore, the trace timing accuracy depends on the TSU_TC_TRIG frequency. The higher the frequency the higher the timing accuracy. The highest accuracy is achieved by setting the TSUPRSCL pre-scaler value to 1.

If the TSUPRSCL value is 1 and CPU clock (also Main PLL clock) is for instance 200MHz, this means that a TSU_REL time stamp message is generated after every 240ns. For a TSUPRSCL value of 4, this would mean a TSU_REL message every 600ns.

Figure 10 show the corresponding MCDS and MCX configuration dialog settings for this use-case.



Figure 10: Hardware, MCDS and MCX Configuration Dialogs enabling "TSUREL" based Time Stamping

The periodic TSU_REL time stamp based trace timing reconstruction concept is also depicted in Figure 11.



Use-case 2: TSU_REL generation upon specific triggers

This kind of time stamp generation is not recommended.

1.5.2 Message Storage Control (in EMEM)

Trace data can either be stored in the so-called Emulation Memory (EMEM) or it is streamed out via the AGBT interface.

The structure of the EMEM (for a TC2x emulation device) is depicted in Figure 12.



Figure 12: Simplified Structure of the Emulation Memory (EMEM) of TC2x

The EMEM consists of three parts, the Extended Calibration Memory (XCM), the Trace/Calibration Memory (XCM) and the Extended Trace Memory (XTM).

The XCM is used by the calibration software of the application (i.e. CPU). It cannot be used for trace (MCDS).

The TCM can either be used by the software (CPU) or be the MCDS. The TCM is separated into so called tiles. These tiles can be assigned to either trace or calibration. All the tiles assigned to trace form the on-chip trace buffer. The MCX manages how the trace data is written into the TCM trace buffer. The storage operation in the buffer is basically performed in two phases, the so called "Pre-Trigger Phase" and the "Post-Trigger Phase".

Pre-Trigger Phase:

The storage always starts in Pre-Trigger phase. In this phase, the MCX uses a user-defined portion of the available trace buffer (i.e. tiles assigned to trace) as a circular buffer. The size of this circular buffer is defined by the Trigger Position. It can either be:

- Begin => No circular buffer available in Pre-Trigger phase.
- Center => Half of the trace buffer is used as circular buffer available in Pre-Trigger phase.
- End => The entire trace buffer is used as circular buffer available in Pre-Trigger phase.

Upon the occurrence of a user-defined Trigger Event, the storage operation switches into the Post-Trigger phase. This Trigger Event is either generated by the MCX itself or can originate from a Trigger-Event-Action generated by a POB or BOB.

Typical Trigger Events are:

- A CPU executes an instruction at a specific address location, such as an entry into a function.
- A CPU performs a memory write access to a specific memory address location using a specific data value. Such an event could for instance be generated by the OS signaling that a specific OS task is running.

Post-Trigger Phase:

In this phase the remaining portion of the trace buffer is filled with trace messages which are generated after the occurrence of the Trigger Event (trace_done). Once the allocated trace buffer is full, trace is automatically stopped (the CPU operation is not influenced).

The concept is also depicted in Figure 13.



Figure 14 shows the EMEM TCM Trigger Position configuration dialog in the MCDS tab.

Trigger - [Advanced Coverage Trigger]

MCDS	TriCore X	TriCore Y	SRI	SPB	MCX	INET		
Tric	ider							
	igei	_			-			
Tr	igger Positio	n B	Begin 🔨					
		В	leain		J.			
MLI	v	C	enter					
MO.	^	E	ind					
W	hich SRT sla	ve is seen h	v SRT1		TMU () M	U SRAM	. FM	

Figure 14: EMEM TCM Trigger Position Dialog

Figure 15 shows the EMEM TCM Trigger Event configuration dialog in the MCX tab.

Trigger - [Advanced Coverage Trigger]

MCDS TriCore X	TriCore Y SRI	SPB	MCX	INET		
Action (double cli	ick to edit)				Event	(double click to edit)
tsu_rel_en tsu_rel_sync tsu_abs_en tsu_abs_sync wtw_enable_0 wtw_enable_1 wtw_enable_2 wtw_enable_3 wtw_enable_4 wtw_enable_4 wtw_enable_5 wtw_enable_6 wtw_enable_7 wtw_ent_0 wtw_ent_1 wtw_ent_2 wtw_ent_3 wtw_ent_3 wtw_ent_3 wtw_ent_3 wtw_ent_4 wtw_ent_3 wtw_ent_4 wtw_ent_5 wtw_ent_6 wtw_ent_7 wtw_ent_6 wtw_ent_7 wtw_ent_9 wtw_ent_10 wtw_ent_11 wtw_ent_12 wtw_ent_11 wtw_ent_12 wtw_ent_11 wtw_ent_11 wtw_ent_11 wtw_ent_11 wtw_ent_11 wtw_ent_12 wtw_ent_11 wtw_e	ALWAYS ^EVT8 - - - - - - - - - - - - -	ger		*	EVT0 EVT1 EVT2 EVT3 EVT4 EVT5 EVT6 EVT7 EVT8 EVT9 EVT10 EVT11 EVT12 EVT11 EVT12 EVT13 EVT14 EVT15 EVT16 EVT16 EVT17 EVT18 EVT19 EVT20 EVT21	tcx_act_0
	-					

Figure 15: Sample trace_done Trigger Event Configuration (switch from Pre- to Post-Trigger mode upon occurrence of MCX EVTO, imported from POB X).

1.5.3 Event Counters

The MCX implements a set of event counters. These counters can be incremented, decremented and cleared upon input events, selectable from a pool of events. Such events may be "core instruction executed", "cache hit" or events imported to other MCDS components such as POBs or BOBs. The event counter can either be used a trigger course of MCX events or may be may be output via the Watchpoint Trace Unit (WTU).

1.5.4 Trigger Feedback to Observation Blocks

Events of the MCX may be exported and routed to observation blocks such as POBs or BOBs, where they can be used as triggers.

For example, program trace of a POBs could be enabled when an event counter of the MCX exceeds a certain threshold.

1.6 Emulation Memory

The structure of the Emulation Memory (EMEM) is shown in Figure 12. The TCM tiles can be used for the following trace use-cases:

1.6.1 Trace into EMEM until full, Read-out via DAP or JTAG Interface

This use-case is basically described in "Message Storage Control" chapter above.

1.6.2 Trace Streaming via DAP Interface using EMEM Tiles for interim Buffering ("Upload while Sampling")

In this case, the available EMEM tiles are managed by the iSYSTEM tool in a way that allows a permanent streaming of trace data. This so-called "Upload while Sampling" (UWS) mode allows a virtually unlimited trace recording, assuming that the trace data generation rate (be the MCDS) is less or equal the data throughput via the DAP interface.

UWS is operational with a minimum of 2 EMEM tiles. However, it is recommended to allocate a minimum of 3 EMEM tiles to trace when using UWS.

1.6.3 Trace Streaming via AGBT using a EMEM Tile as FIFO

In this case a TCM tile is used as a FIFO within the AGBT trace data path. Please note, the only on a few AURIX derivatives TCM tiles are used as AGBT FIFO. Typically, the AGBT uses the two XTM files as FIFO.

1.7 Debug Access Port (DAP)

The DAP is an Infineon proprietary interface. It can be used as either a 2-pin (DAP0, DAP1) or 3-pin (DAP0, DAP1, DAP2) bi-directional interface to communicate debug and trace information between the AURIX device and the tool. The DAP pins are multiplexed with the standard JTAG pins and are available on every AURIX device (also Production Devices).

Emulation devices of the TC3x family also offer a second DAP interface, the so-called DAPE.

The DAP interface can operate at clock frequencies of up to 160MHz. The maximum applicable frequency depends on the hardware setup, i.e. target board layout (e.g. distance between device and DAP connected on the ECU).

The iSYSTEM iC5700 allows to either access the DAP interface directly via a DAP cable adaptor or via a dedicated AURIX DAP Active Probe.

The operation mode and clock frequency can be configured in winIDEA with the "Hardware - CPU

PU Setup						
CPU1	CPU2	CPU3	CPU4	CPL	15	HSM
Reset [Debugging Analy	zer Aurora	SoC Advanced	SoC	SoC Events	G CPU0
Override s	tartup register values	3	HSM Enabled			
-Debug char	nnel					
Mode	DAP Wide	~ c	llock 10000 ki	Hz		
	DAP Standard					
	BRKIN drive HIG	н				
_						
✓ DAPE	DAP Wide	~ 0	lock 120000 k	Hz		
FLACU						
	- LASH modification or	lv during download	ł			
	JCB data flash progra	mmina (not recom	mended)			
Warnin	ng: Wrong UCB data	an lock the device	permanently!			
Allow N	Mass erase (NOT Rec	ommended!) ck the device perm	apontlyl			
warnii	ig. Mass erase carrio	ck the device perm	anenuy			
Trace Buffe	er					
Region	TCM 🗸					
⊡ Use all	available memory for	trace buffer	_			
First	tile 0 Num	ber of tiles 4]			
Use Pase	sword	0000000	0000000	FX		
0000000	00000000	00000000				
000000	0000000	0000000				
			Ok		Cancel	Help

Options... - SoC".

Figure 16: DAP Configuration Options in the winIDEA Dialog "CPU Setup – SoC"

1.8 AURORA Gigabit (AGBT) Interface

The AGBT interface is a very-high bandwidth trace streaming interface. It uses differential signaling in order to achieve transfer bit rates of several Gbit/s (Gbps). On AURIX devices the AGBT bandwidth is typically 2.5 Gbps. This makes the AGBT interface suitable to perform unconditional program trace and OS trace on multiple CPUs simultaneously.

However, as the interface runs at frequencies in the GHz range, high-frequency design rules need to be applied when using the AGBT interface on the target hardware.

The AURIX AGBT interface complies with the AURORA trace interface specification of the NEXUS 5001[™] Forum Standard (http://nexus5001.org/)

The operation mode and clock frequency can be configured in winIDEA with the "Hardware – CPU Options... - Aurora".

CPU Setup								×
CPU1	C	PU2	CPU3	CPU4	CF	VU5	Н	ISM
Reset	Debugging	Analyzer	Aurora	SoC Advanced	SoC	SoC Ev	ents	CPU0
Number o	flanes 1		\sim					
Baudrate	1.	25 Gbps	ĸ					
Gener	ate clock 2.	25 Gbps 5 Gbps						

Figure 17: AURORA AGBT Configuration Options in the winIDEA Dialog "CPU Setup – Aurora"

1.9 Usage of Initialization (.INI) Files

The Initialization file (.INI) is used to configure the following on-chip features:

- Multi-Core Synchronization
- Peripheral Freeze
- AGBT Trigger Output

iSYSTEM provides a INI file for each AURIX family, which covers the typical use-cases. However, a user may need to modify the INI file according to the particular use-case.

Listing 1 shows a typical INI file.

```
01 //TL1: For peripheral Suspend control
02 //TL2: for CPU HALT indication
03
04 // MULTI CORE SYNCHRONIZATION
05 // break out outputs from all cores are connected to TL1
06 // capture and hold on TL1 is enabled
07 // all cores are suspend targets
08 A CBS_TLCHE L 0x00000002 // TL1 capture and hold enabled
09 A CBS_TL1ST L 0x30000007 // all CPUs are suspended target
                                            // DMA is suspend target
10
11
                                            // HSSL is suspend target
12
13 A CBS_TRC0 L 0x00000102
                                            // BT1 - CPU0 is trigger source
                                             // HALT connected to TL2
14
15 //A CBS TRC1 L 0x00000100 // BT1 - CPU1 is trigger source
16 //A CBS TRC2 L 0x00000100 // BT1 - CPU2 is trigger source
17 A CBS_TLCL0x00000030// TL1 forced to active18 A CBS_TLCL0x00000000// TL1 force removed
19
20 // TRACE TRIGGER OUTPUT
21 // MCDS trig out 0 is connected to TL4 \,
22 // output is stretched to min 4PBs clocks
23 // TL4 line is connected to output port 4 (P32.6)
23A CBS_TOPR L0x00040000// TL4 connected to trig out pin 425// (port P32.6)26A CBS_TRMT L0x00000004// MCDS trigger out 0 connected to TL427A CBS_TOPPS L0x00000200// trigger output pulse stretched to 4PBs28A P32_PDR0 L0x30333333// port P32.6 - speed grade 4 (max)
29
30 // DISABLE TRACE TIME WHEN CPU IS STOPPED
31 // Master CPU (CPU0) connects HALT output to TL2
32 // MCDS break in connection
33 A CBS TRMC L 0x00200000 // MCDS Break in is connected to TL2
34
35 //-----
36 //STM suspend control

        37 A STM0_OCS
        L
        0x12000000

        38 A STM1_OCS
        L
        0x12000000

        38 A STM1_OCS
        L
        0x1200000

        39 A STM2_OCS
        L
        0x12000000

40 //-
41 ...
```

```
Listing 1: Sample AURIX Initialization (INI) File
```

2 Trace Use-Cases

This section discusses some common use-cases.

Use-cases:

- Multi-Core OS Profiling via DAP
- Multi-Core OS and Function Profiling via AGBT

2.1 Multi-Core OS Profiling via DAP

This section demonstrates the profiling of an AUTOSAR OS using all three cores of a TC27x device. The objective is to perform a timing analysis of the running tasks and running ISR2s of all three cores. The tasks and ISR2 shall be displayed in the same winIDEA Analyzer window. The trace recording should last several seconds and shall be performed via the DAP interface of the AURIX device.

2.1.1 What needs to be traced?

The Information Section of the corresponding ORTI file reveals that the OS data object Os_Cfg_Trace_OsCore_CoreX_Dyn is used for running Task ("CurrentTask") and running ISR2 ("CurrentIsr") trace.

```
* Information Section
 OS TC27x {
 vs SMP NUMCPU = "3";
 /* OS information for AUTOSAR core OsCore Core0 */
 vs_COREID[0] = "OsCfg_Core_OsCore_Core0.Core.Id";
 RUNNINGTASK[0] = "OsCfg_Trace_OsCore_Core0_Dyn.CurrentTask";
 RUNNINGISR2[0] = "OsCfg_Trace_OsCore_Core0_Dyn.CurrentIsr";
 /* OS information for AUTOSAR core OsCore Core1 */
 vs COREID[1] = "OsCfg Core OsCore Corel.Core.Id";
 RUNNINGTASK[1] = "OsCfg_Trace_OsCore_Core1_Dyn.CurrentTask";
 RUNNINGISR2[1] = "OsCfg Trace OsCore Core1 Dyn.CurrentIsr";
 /* OS information for AUTOSAR core OsCore Core2 */
 vs COREID[2] = "OsCfg Core OsCore Core2.Core.Id";
 RUNNINGTASK[2] = "OsCfg_Trace_OsCore_Core2_Dyn.CurrentTask";
 RUNNINGISR2[2] = "OsCfg_Trace_OsCore_Core2_Dyn.CurrentIsr";
}; /* OS */
```

```
Listing 2: Sample ORTI File of the AUTOSAR Demo Application
```

Displaying the Os_Cfg_Trace_OsCore_CoreX_Dyn objects in the winIDEA Watch window (see Figure 18) reveals that all of them are located in the local memory of CPU 2 (please refer to the memory map description of the Infineon AURIX user manual).

Watch 1								
😸 🌜 🔊 🚾 🖲 😭 🔚		~						
Name	Value	Туре	Address					
📮 OsCfg_Trace_OsCore_Core0_Dyn	(Ptr(0x8000'2F18)	Os_TraceCoreType	(Virtual)5000'AFAC					
⊕ CurrentTask	Ptr(0x8000'2F18)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual)5000'AFAC					
—	Ptr(0x0000'0000)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual)5000'AFB0					
OrtiCurrentServiceId	'p' (0x70)	unsigned char	(Virtual)5000'AFB4					
OsCfg_Trace_OsCore_Corel_Dyn	(Ptr(0x8000'2F24)	Os_TraceCoreType	(Virtual) 5000'AFE0					
—	Ptr(0x8000'2F24)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual) 5000'AFE0					
⊡ CurrentIsr	Ptr(0x0000'0000)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual)5000'AFE4					
OrtiCurrentServiceId	\x89 (0x89)	unsigned char	(Virtual)5000'AFE8					
OsCfg_Trace_OsCore_Core2_Dyn	(Ptr(0x8000'2F30)	Os_TraceCoreType	(Virtual)5000'B010					
—	Ptr(0x8000'2F30)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual)5000'B010					
⊡ CurrentIsr	Ptr(0x0000'0000)	<pre>Os_TraceThreadConfigType *</pre>	(Virtual)5000'B014					
OrtiCurrentServiceId	\x89 (0x89)	unsigned char	(Virtual) 5000'B018					

Figure 18: winIDEA Watch Window with OS Data Objects used for Tracing RunningTask and RunningISR2

This means that only CPU2 has local access to these objects. The other cores (CPU0 and CPU1) need to access their associated OS data object via the SRI. These access paths are relevant for determining which on-chip trace concept is most suitable for this setup.

As CPU2 accesses its OS object locally, only a POB connected to CPU2 can monitor these data transactions. The data transactions to the OS objects of all other CPUs can be monitored by a BOB_SRI connected to the SRI slave interface of DSPR of CPU2 (see the corresponding BOB_SRI MUX configuration in Figure 25).

Figure 19 depicts the AURIX internal data transaction paths of the individual CPUs to the OS data objects located in DSPR2 (red). It is also shows where the POB and BOB needs to be connected (via MUX) in order to monitor (i.e. trace) these transactions.



Figure 19: AURIX-internal Data Access Paths to the OS Data Objects and POB/BOB Observability

Please note, the memory allocation and the corresponding MCDS trace setup described above applies to this particular sample project. The relevant OS data objects in your AUTOSAR project may be located in different memory locations, such as LMU or DSPR of CPU0.

2.1.2 winIDEA Configuration

DAP Active Probe Detection

After the communication to the iC570 has been established, it is recommended to perform a detection of the connected Active Probe. This can be done via the menu "Hardware – Emulation Options – Probe". Select Active Probe and then click the "Refresh" button. Select the detected DAP Active Probe. In the Active Probe detection shown in Figure 20, the Active Probe has been given a alias "DAP_SLV".

Emulati	on Options		×
Probe	Hardware	CPU	
OD	гм		
A	ctive Probe	DAP_SLV V Refresh	

Figure 20: iC5700 Active Probe Detection.

DAP Width & Frequency Configuration

As the DAP interface is not only used for debug control communication but also for the transport of trace data (Upload-While-Sampling), it is essential to set the DAP to maximum performance, i.e. if possible use "DAP Wide" mode and apply maximum possible clock frequency.

The maximum supported DAP clock frequency supported by both the AURIX device and also the DAP Active Probe is 160MHz. However, the individual target board layout may not allow a DAP operation at 160MHz. Thus, the maximum applicable DAP clock be evaluated individually on each target setup.

In addition, for maximum Upload-While-Sampling performance, a minimum of 3 TCM tiles should be available as Trace Buffer.

CPU Setu	р						×
	CPU1			CPU2		HSM	
Reset	Debugging	Analyzer	Aurora	SoC Advanced	SoC	SoC Events	CPU0
Debug	channel						
Mode	DAP Wide		~	Clock 130000 k	Hz		
	BRKIN drive	HIGH	~				
	uick LBIST detectio	on					
Пня	SM Enabled						
🗌 Flu	ush trace when A	GBT is stoppe	d				
Trace f Regio	Buffer n TCM se all available mer	∽ mory for trac	e buffer				
F	irst tile 0] Number o	f tiles 16				

Figure 21: DAP Interface Configuration

Trace Port Selection

If the DAP interface is used for debug and trace data transfer, the trace data is still buffered in the onchip trace buffer, i.e. EMEM. Therefore, the Analyzer Operation mode "On-Chip" needs to be selected. The Cycle duration does not represent the CPU clock cycle duration, but the MCDS clock cycle duration (The MCSD clock is typically either equal or half the CPU clock.).

CPU Setup	0						×
	CPU1			CPU2		HSM	
Reset	Debugging	Analyzer	Aurora	SoC Advanced	SoC	SoC Events	CPU0
Opera ☑ Co Cycle SoC Ini Before	tion mode de missing from duration tialization e start [On-Chip download file s 10 ns Default	hould be rea	ad at run-time			

Figure 22: Trace Port Selection for Trace via DAP (On-Chip)

2.1.3 winIDEA Trace Analyzer Configuration

It is recommended to create a new trace configuration for each trace use-case.

C:\iSYSTEM_SAMPLES_SVN\Infineon\TriCore\T							
2							
	Default						
*	Create New Configuration						
$\mathbf{\lambda}$	Edit Analyzer Configuration List						

Figure 23: Creating a new winIDEA Analyzer Trace Configuration

A trace configuration for OS task and ISR2 profiling of three cores, could, for instance, look like depicted in Figure 24. The configuration should have a descriptive name, enabled **Profiler** Analysis and enabled **Manual** Hardware Trigger Configuration.

New Configuration	×
Name OS_Tasks_ISR2_TripleCore	
Analysis Profiler Coverage	
Trace - Always Performed	
Hardware Trigger Configuration Automatic Wizard Program flow + Instrumentation Template Manual Initial hardware trigger configuration is empty and user has to configure it. Wizard assistance is provided for typical use cases.	_
OK Cancel	

Figure 24: Sample new Trace Configuration

Figure 25 shows the MCDS configuration corresponding to the POB/BOB connectivity described in Figure 19.

Trigger - [Advanced Coverage Trigger]

MCDS TriCore X TriCore Y SRI SPB	MCX iNET						
Trigger Trigger Position End	~						
MUX							
Which SRI slave is seen by SRI1	CPU2 (PSPR,DSPR) ~						
Which SRI slave is seen by SRI2	CPU2 (PSPR,DSPR) V						
Which processor core is seen by POB X	CPU2 🗸						
Which processor core is seen by POB Y	Nothing ~						
Which processor core is seen by POB Z	~						
Time stamps							
Assume source to be tick	~						
TSUPRSCL 1	HEX						
Reference dock Main PLL	\sim						
Note: configure cycle duration in Hardware/CPU Setup/Debugging							
Options							
Enable trace during CPU reset Note: enabling this option, will disable the trigger (MCX/trace_done is set to NEVER)							
Continuous mode	lote: enabling this option, will disable UWS nd force MCX/trace_done to NEVER						

Figure 25: POB/BOB MUX and Timestamping Configuration

Trigger:

The "Trigger Position" setting is not relevant in this case, as we will use DAP Upload-While-Sampling.

MUX:

The BOB_SRI and POB X and Y MUXes are set in the following way:

- BOB_SRI1: CPU2 (PSPR, DSPR...), i.e. connected to the SRI interface of the CPU2 local memory.
- BOB_SRI2: CPU2 (PSPR, DSPR...), i.e. connected to the SRI interface of the CPU2 local memory.
- POB X: CPU2, i.e. connected to CPU2

Time stamps:

TICK time stamping is used ("Assume source to be tick").

In this case, the value entered for TSUPRSCL and also the "Reference clock" selection is irrelevant.

Options:

No additional options need to be enabled.

Figure 26 shows	the required P	OB X configuration	(POB X is connect	ted to CPU2).
-----------------	----------------	--------------------	-------------------	---------------

ction (double click to edit)	Event (double click to edit)	Trigger (double click to edit)	
to use the total of the second sec	EVT0 EVT1 EVT2 EVT3 EVT4 EVT5 EVT6 EVT7 EVT8 EVT10 dtu_ea_trig_2 EVT11 dtu_ea_trig_3 EVT15	core_trig0 core_trig1 core_trig2 core_trig3 core_trig4 core_trig5 core_trig6 core_trig7 c).Curre).Curre
DCU_TC status trace enable Advanced		dtu_ea_trig_5 ALWAY dtu_ea_trig_5 ALWAY dtu_ea_trig_6 ALWAY dtu_ea_trig_7 ALWAY dtu_dat_trig_0 ALWAY dtu_dat_trig_1 ALWAY dtu_dat_trig_2 ALWAY dtu_dat_trig_3 ALWAY dtu_dat_trig_3 ALWAY	rs rs rs rs rs rs rs rs rs rs

Figure 26: POB_X Configuration to trace Data Write Access of Core 2 to the OS Objects of Core 2

Trigger:

• Two magnitude comparators (address comparators) generate a trigger when CPU2 accesses the Running Task and Running ISR2 signaling variables of the OS.

Configuration of dtu_ea_trig_2 (Running Task):



Configuration of dtu_ea_trig_2 (Running ISR2):

Trigger	×
$X <= ADDR <= Y$ \lor	ОК
X Core_Core2_Dyn).CurrentIsr	Cancel
Entire Object	
Υ	

Events:

- Trigger dtu_ea_trig_2 is mapped to EVT10.
- Trigger dtu_ea_trig_3 is mapped to EVT11.

Actions:

• EVT10 or EVT11 both cause capturing the Write Access Data (dcu_wdat) and Write Access Address (dcu_waddr).

Figure 27 shows the required POB Y configuration (POB Y is not connected to any CPU).

ction (double click to edit)	Event (double click to edit)	Trigger (double click to edit)	
cu_enable - cu_sync - tu_wadr - tu_madr - tu_madr - tu_madr - tu_noble - tu_enable - tu_sync - tu_enable - tu_sync - tu_sync - tu_enable - tv_enable_1 - tvtu_enable_2 - tvtu_enable_4 - tvtu_enable_5 - tvtu_enable_7 - c_act_0 - c_act_1 - c_act_2 - c_act_3 - su_rel -	EVT0 - EVT1 - EVT2 - EVT3 - EVT4 - EVT5 - EVT5 - EVT7 - EVT8 - EVT9 - EVT10 - EVT10 - EVT11 - EVT11 - EVT12 - EVT13 - EVT14 - EVT15 -	Core_crevt Core_trig0 Core_trig1 Core_trig2 Core_trig4 Core_trig6 Core_trig6 Core_trig6 Core_trig7 Core_trig6 Core_trig7 Core_trig6 Core_trig17 Core_trig10 <	
DCU_TC status trace enable Advanced		dtu_dat_trig_1 ALWAYS dtu_dat_trig_2 ALWAYS dtu_dat_trig_3 ALWAYS dtu_acc_wr dtu_acc_wr	

Figure 27: POB_Y Configuration, not used in this use-case

Figure 28 shows the required BOB_SRI configuration. In this configuration the Data Trace Unit 1 (DTU1) of the BOB_SRI is used to trace the CurrentTask and CurrentIsr object of CPU0, DTU2 is used to trace the CurrentTask and CurrentIsr object of CPU1.

ction (double click to edit)	Event (double click to edit)	Trigger (double click	(to edit)
icu_enable - icu_sync - icu_sync - itu1_wadt EVT0 EVT1 itu1_radt - itu1_radr - itu2_wadt EVT15 EVT14 itu2_wadr EVT15 EVT14 itu2_wadr EVT15 EVT14 itu2_radr - vtu_enable_1 - vtu_enable_2 - vtu_enable_4 - vtu_enable_7 - ri_act_1 - vi_uenable_6 - vtu_enable_7 - ri_act_1 - ri_act_1 - ri_act_2 - vtu_enable_6 - vtu_enable_7 - ri_act_1 - ri_act_1 - ri_act_2 - ri_act_4 - ri_act_5 - ri_act_7 - su_rel - Su_rel - Su_rel -	EVT0 dtu1_ea_trig_0 EVT1 dtu1_ea_trig_1 EVT2 - EVT3 - EVT3 - EVT5 - EVT5 - EVT6 - EVT7 - EVT9 - EVT10 - EVT10 - EVT11 - EVT12 - EVT13 - EVT13 dtu2_ea_trig_0 EVT15 dtu2_ea_trig_1	Itul ea fine dtu2 ea fine dtu1 ea trig_0 dtu1 ea trig_1 dtu1 ea trig_1 dtu1 ea trig_1 dtu1 dat trig_2 dtu1 dat trig_3 dtu1 act trig_1 dtu1_acc trig_1 dtu1_acc trig_2 dtu1_acc trig_2 dtu1_acc trig_3 dtu2 ea trig_1 dtu2 ea trig_3 dtu2 dat trig_1 dtu2 dat trig_3 dtu2 acc trig_0 dtu2 acc trig_0 dtu2 acc trig_1 dtu2 acc trig_3 dtu3 acc trig_3 </td <td>[(OsCfg_Trace_OsCore_Core0_Dyn).Curre [(OsCfg_Trace_OsCore_Core0_Dyn).Curre ALWAYS ALWAYS ALWAYS ALWAYS ALWAYS ACcess Access</td>	[(OsCfg_Trace_OsCore_Core0_Dyn).Curre [(OsCfg_Trace_OsCore_Core0_Dyn).Curre ALWAYS ALWAYS ALWAYS ALWAYS ALWAYS ACcess Access

Figure 28: BOB_SRI Configuration to trace Data Write Accessed of Cores 0 and 1 to the corresponding OS Objects

Trigger:

• Two magnitude comparators (address comparators) generate a trigger when CPU0/1 access the corresponding CurrentTask (Running Task) and CurrentIsr (RunningISR2) variables of the OS.

Configuration of dtu1_ea_trig_0 (Running Task):



Configuration of dtu1_ea_trig_1 (Running ISR2):

Tri	gger		×
	X <= ADDR <= Y	\sim	OK
x	Core_Core0_Dyn).CurrentIsr		Cancel
	Entire Object		
Y			

DTU2 is configured accordingly.

Events:

- Trigger dtu1_ea_trig_0 is mapped to EVT0.
- Trigger dtu1_ea_trig_0 is mapped to EVT1.
- Trigger dtu2_ea_trig_0 is mapped to EVT14.
- Trigger dtu2_ea_trig_0 is mapped to EVT15.

Actions:

- EVT0 or EVT1 both cause capturing (by DTU1) the Write Access Data (dtu1_wdat) and Write Access Address (dtu1_waddr).
- EVT14 or EVT15 both cause capturing (by DTU2) the Write Access Data (dtu2_wdat) and Write Access Address (dtu2_waddr).

action (double click to edit)	Event (double click to edit)	Trigger (double click to edit)	
Advanced Advanced	Evro Evro <t< th=""><th>Ingel (dobie cit to early break: ffo_trg_0 ffo_trg_1 tsu_tc_trig tcx_act_0 tcx_act_1 tcx_act_2 tcx_act_1 tcy_act_1 tcy_act_3 spb_act_1 spb_act_1 spb_act_1 spb_act_1 spb_act_1 sri_act_1 sri_act_2 sri_act_3 sri_act_5 sri_act_5 sri_act_7 cnt_trig_0 cnt_trig_1 cnt_trig_4 cnt_trig_5 cnt_trig_6 cnt_trig_9 cnt_trig_10 cnt_trig_11 cnt_trig_12</th><th></th></t<>	Ingel (dobie cit to early break: ffo_trg_0 ffo_trg_1 tsu_tc_trig tcx_act_0 tcx_act_1 tcx_act_2 tcx_act_1 tcy_act_1 tcy_act_3 spb_act_1 spb_act_1 spb_act_1 spb_act_1 spb_act_1 sri_act_1 sri_act_2 sri_act_3 sri_act_5 sri_act_5 sri_act_7 cnt_trig_0 cnt_trig_1 cnt_trig_4 cnt_trig_5 cnt_trig_6 cnt_trig_9 cnt_trig_10 cnt_trig_11 cnt_trig_12	

Figure 29 shows the MCX configuration required to use TICK time stamping.

Figure 29: MCX Configuration to generate TICKS for Time Stamping and Upload-While-Sampling

As a final stage of the trace configuration the Recorder, i.e. iC5700, properties need to be set. In this use-case we immediately start trace recording and use the Upload-While-Sampling feature to stream trace messages via the DAP interface.

Figure 30 depicts the corresponding Recorder settings.

ardware P	rofiler Covera	ige			
-Analysis an	nd Configuration	1			
Profile	r				
Cover	age				
🗹 Manua	al Trigger/Recor	der configuration	Configure	Preset >	
	FNet Operati	ion			
Options Reacti	ivate session af	fter CPU stop			
Options Reacti	ivate session af	fter CPU stop	Value		
Options Reacti Property	ivate session af	fter CPU stop	Value		
Options Reaction Property Record Start	ivate session af	fter CPU stop	Value Trigger Immediately		^
Options Reacti Property Record Start Recordi	ivate session af der Ing Size Limit	fter CPU stop	Value Trigger Immediately 1 GB		^
Options Reaction Property Records Start Records Trigger	ivate session af der ing Size Limit Position	fter CPU stop	Value Trigger Immediately 1 GB Begin		^
Options Reaction Property Records Start Records Trigger Timer In	ivate session af der ing Size Limit Position iterpolation	fter CPU stop	Value Trigger Immediately 1 GB Begin		
Options Reaction Property Records Start Records Trigger Timer In Generat	ivate session af der ing Size Limit Position iterpolation te time synchror	fter CPU stop	Value Trigger Immediately 1 GB Begin		

Figure 30: Analyzer Configuration for DAP Upload-While-Sampling (UWS) and immediate Recording

2.1.4 winIDEA Profiler Configuration

In order to make winIDEA and the winIDEA Profiler aware of the AUTOSAR OS running on the target the so-called ORTI file, generated by the AUTOSAR generation tool, needs to be imported into winIDEA. This is done via the menu "Debug – Operating System...".

When importing the AUTOSAR ORTI file via the "New..." button, the OS type "OSEK AUTOSAR" has to be selected (see Figure 31). Afterwards you can give the OS awareness some descriptive name. In our example shown in Figure 32, the ORTI file has the name "Os_Trace.ORT". As the AUTOSAR OS used in this example is a Vector Microsar OS, we name the OS-awareness the name "Microsar ORTI".

Operating system	×
Operating System	
	New
	OSLinux
	L4ReOS
	PikeOS
	freeRTOS RTOS
	OSEK AUTOSAR
	NOV TOO W

Figure 31: Creating of a new OSEK AUTOSAR OS Awareness

Operating system		×	Edi	t options	
Operating System		_		Property	Value
[OSEK AUTOSAR] Microsar ORTI	New			Configuration	
			RTOS description file type	ORTI	
	Configure Rename			RTOS description file location	Os_Trace.ORT

Figure 32: Selection of the AUTOSAR ORTI File (in this example the file "Os_Trace.ORT")

A "Download" or "Symbol Download" operation also reads in the ORTI file.

Subsequently, the winIDEA Profiler can be configured to utilize the information given by the ORTI file. As shown in Figure 33, the profiling of "OS objects" needs to be enabled.

Analyzer Configuration - [OS_ORTI]	× *************
Hardware Profiler Coverage	RTOS Profiler Options ×
Code Advanced	Operating System Microsar ORTI ~
☐ OS objects OS Setup	Objects to profile
AUX	✓ Core 2: Tasks ✓ Core 0: ISRs2 ✓ Core 1: ISRs2 ✓ Core 2: ISRs2
Code Areas Enter filter string(s)	Object Info:
	Name: RUNNINGTASK[0] Definiton: RUNNINGTASK[0] Description: Core 0: Tasks Signaling: OsCfg_Trace_OsCore_Core0_Dyn.CurrentTa

Figure 33: "OS Setup..." Configuration of the Analyzer – Profiler Dialog

Pushing the "OS Setup..." button opens the "RTOS Profiler Options" dialog. The name of the Operating Systems corresponds to the name that was given via the "Debug – Operating System..." dialog, when selecting the ORTI file.

All OS objects described by the ORTI are listed under "Objects to profile". One or multiple objects can be selected for profiling. The sub-window "Object Info:" provides more detailed info of a selected object, such as the Name given in the ORTI file and the type of signaling.

In the example shown in Figure 33 the ORTI object RUNNINGTASK[0], i.e. the Running Task on CPU0, is signaled via the global variable Os_Cfg_Trace_OsCore0_Dyn.CurrentTask. As described in the section 2.1.2 these are the global variables which need to be observed via Data Trace.

2.1.5 winIDEA Profiler View

Figure 34 shows the resulting profiler timeline.



Figure 34: Sample winIDEA Profiler Timeline, showing Running Tasks and Running ISR2s of all three Cores of a Multi-Core AUTOSAR OS running on an Infineon AURIX TC277

2.2 Multi-Core OS & Program Trace via AGBT

This section demonstrates the code/function profiling of an AUTOSAR OS based multi-core application. How many cores can be profiled in parallel depends on the number of available POBs, i.e. either two or three cores. The objective is to perform timing analysis of the complete software running on the cores. The OS running tasks and running ISR2s are also traced. This is needed when nested function profiling is done on systems using a pre-emptive operation system (such as the AUTOSAR OS).

Nested function profiling is provided when the winIDEA Profiler operates in either "Range" or "Entry/Exit" mode.

The AGBT interface is designed to provide sufficient trace bandwidth for such a use-case (see also section 1.8)

2.2.1 What needs to be traced?

This use-case requires tracing of the program flow as well as the OS task and ISR2 context. Program flow trace can only be done by means of Processor Observation Blocks (POB).

In the sample use-case discussed in the following section, we assume that the multi-core AUTOSAR application is running on three TriCore cores of a TC277TF device. On this device the MCDS offers two POBs, i.e. only on two out of three cores the program flow can be traced. In our particular use-case here we decide to focus on CPU0 and CPU2, thus we trace the program flow of CPU0 and CPU2 via POBs. The data accesses of CPU2 to the OS objects are also traced by the connected POB. The data accesses of the other two cores (CPU0 and CPU1 can be traced by means of one BOB, by connecting it to the DSPR2 slave of the SRI. Thus, overall we can trace:

- The program flow of CPU0 and CPU2
- The OS tasks and ISR2 of all three cores

Figure 35 depicts the overall setup.



Figure 35: Sample AURIX-internal Data Access Paths & POB/BOB Connectivity for Multi-Core Program Flow and OS Trace

2.2.2 winIDEA Configuration

Infineon AGBT Active Probe Detection

After the communication to the iC570 has been established, it is recommended to perform a detection of the connected Active Probe. This can be done via the menu "Hardware – Emulation Options – Probe". Select Active Probe and then click the "Refresh" button. Select the detected AGBT Active Probe.

Emulation Options

CPU Setup

Probe	Hardware	CPU		
OD	тм			
Ac	ctive Probe	AP_AGBT_MST	~	Refresh

Figure 36: AGBT Active Probe Detection

DAP Width & Frequency Configuration / AGBT Flush & Buffer Configuration

Although an AGBT Active Probe is used, the bi-directional debug communication is still performed via the DAP interface. Therefore, also the DAP interface needs to be configured. DAP mode should be set to "DAP Wide", the DAP clock speed is not that critical in this case as the high-bandwidth trace data streaming is routed through the AGBT interface.

These settings can be configured via the "Hardware – CPU Options... - SoC" dialog as shown in Figure 37.

	CPU1			CPU2			
Reset	Debugging	Analyzer	Aurora	SoC	C Advanced	SoC	
Debug o	hannel						
Mode	DAP Wide		~	Clock	30000	kHz	
	PDVIN drive	LITCH					
	DRATH GUN	HIGH	~				
🗌 Qui	ck LBIST detect	ion					
HSI	M Enabled						
⊡ Flu	sh trace when A	AGBT is stoppe	d				
			_				
Trace B	uffer						
Region	TCM	1 ~					
Use all available memory for trace buffer							
Fir	rst tile 0	Number o	f tiles 16	i			

Figure 37: DAP Mode/Frequency Selection & AGBT Flush/Buffer Selection

The option "Flush trace when AGBT is stopped should be enabled. However, in case an AGBT overflow occurs at the end of the trace recording, this option should be disabled. Potentially, the flush operation ³³ of ⁵¹ Application Note

which is performed when trace is stopped may actually cause an AGBT overflow. Thus, disabling the AGBT buffer flushing may eliminate the overflow.

Which type of Trace Buffer can be used depends on the individual device. In general, either EMEM TCM tiles or the EMEM XTM can be used as AGBT FIFO buffer.

Trace Port Selection

As the AGBT interface is used here, the Analyzer must run in Operation mode "AURORA Trace Port".

CPU Setup							×
	CPU1			CPU2			
Reset	Debugging	Analyzer	Aurora	SoC Advanced	SoC	SoC Events	CPU0
Operat Cod Cycle o SoC Init	tion mode	Aurora Trace Po download file s 12.5 ns	ort	ad at run-time			

Figure 38: Analyzer Trace Port Selection for AGBT

The Cycle duration does **not** directly represent the CPU clock cycle duration, but the MCDS clock cycle duration. The CPU and also the MCDS clock can be obtained via the TriCore Plugin, which can be opened via the menu "View – TriCore".

Push the "Refresh" button while the CPU is running. The plugin will display the current clock settings for CPU and MCDS in Hz. In the sample shown in Figure 39, the MCDS clock is equal to the CPU clock, running at 80MHz.

Especially for higher CPU clock frequencies, the MCDS clock is typically half the CPU clock.

[TriCore] - TriCore	
2 🛃 🖆 👙	
Group Refresh ⊞™MCXCounters	Count
CDCCounters	
⊡ Clocks	
STMClock	79982kHz
CPUClock	79982kHz
MCDSClock	79982kHz
SRIClock	159964kHz

Figure 39: Sample TriCore Plugin View (MCDS and CPU running at 80MHz, i.e. Cycle Duration is 12.5ns)

In addition, the AGBT interface to be configured. The parameters are:

- **Number of lanes**: How many AURORA Lines are used for the AGBT interface. Most AURIX devices support only 1 AURORA lane.
- **Baudrate**: Most AURIX devices support a bitrate of either 1.25Gbps or 2.5Gbps. If possible, select the higher bitrate to allow for a maximum trace bandwidth.
- **Generate clock**: This parameter is ignored for AURIX devices.

ii.

	CPU Setup	•					
		CPU1			CPU2		
l	Reset	Debugging	Analyzer	Aurora	SoC Advanced	SoC	
	Number	of lanes 1		\sim			
l	Baudrat	e 2	.5 Gbps	Y			
	Gene	erate clock 2	.25 Gbps .5 Gbps	45			

Figure 40: AGBT Interface Configuration (Number of Lanes, Baudrate)

2.2.3 winIDEA Trace Analyzer Configuration

You can either create a new trace configuration for this trace use-case, or you can derive it from an already existing configuration, e.g. from OS profiling configuration described in section 2.1.

\geq			
	Default		
~	OS_ORTI h		
*	Create New Configuration		
$\mathbf{\lambda}$	Edit Analyzer Configuration List		
Ana	alyzer Configuration List		×
-	<u>N</u> ew <u>E</u> dit <u>♠ R</u> ename <u>⇒ S</u> et Active <u></u>	Сору 🗙 !	<u>D</u> elete
	🕘 👎 Default		
⇒	🕒 OS_ORTI		
Ente	er Item Name	×	
Nam	PT_CPU_0_2_OS_ORTI		

Figure 41: Derive a new winIDEA Analyzer Trace Configuration from an existing Configuration

A trace configuration for OS task and ISR2, as a full program trace profiling of two cores (CPU0 and CPU2), could, for instance, look like depicted in Figure 41. The configuration should have a descriptive name, enabled **Profiler** Analysis and enabled **Manual** Hardware Trigger Configuration.

OK

Cancel

Figure 42 shows the MCDS configuration corresponding to the POB/BOB connectivity described in Figure 35.

Trigger - [Advanced Coverage	Trigger]				
MCDS TriCore X TriCore Y	SRI SPB	MCX	I/O Module	iNET	
Trigger					
Trigger Position Er	nd v	1			
MUX					
Which SRI slave is seen by	SRI1	CPU2 (PS	PR,DSPR)		\sim
Which SRI slave is seen by	SRI2	CPU2 (PS	PR,DSPR)		\sim
Which processor core is se	en by POB X	CPU0			\sim
Which processor core is se	en by POB Y	CPU2			\sim
Which processor core is se	en by POB Z				\sim
Time stamps					
Assume source to be tic	k v	/			
TSUPRSCL 1	HE	EX			
Reference clock Ma	ain PLL 🚿	/			
Note: configure cycle dura Hardware/CPU Setup/Debi	tion in ugging				
Options					
Enable trace during CPU	No U reset trig	te: enablin Iger (MCX/	ng this option, /trace_done is	, will disable th s set to NEVER	e !)
Continuous mode	No and	te: enablin d force MC	ng this option, X/trace_done	, will disable UV e to NEVER	NS

Figure 42: POB/BOB MUX and Timestamping Configuration

Trigger:

The "Trigger Position" setting is not relevant in this case, as we will use DAP Upload-While-Sampling.

MUX:

The BOB_SRI and POB X and Y MUXes are set in the following way:

- BOB_SRI1: CPU2 (PSPR, DSPR...), i.e. connected to the SRI interface of the CPU2 local memory.
- BOB_SRI2: CPU2 (PSPR, DSPR...), i.e. connected to the SRI interface of the CPU2 local memory.
- POB X: CPU0, i.e. connected to CPU0
- POB Y: CPU2, i.e. connected to CPU2

Time stamps:

TICK time stamping is used ("Assume source to be tick").

In this case, the value entered for TSUPRSCL and also the "Reference clock" selection is irrelevant.

Options: No additional options need to be enabled.

Figure 43 shows the required POB X configuration (POB X is connected to CPU0).

Action (double dick to edit) Event (double dick to edit) Trigger (double dick to edit) dou_shot - dou_wata - du_wata - ettis - e	DS TriCore X	TriCore Y SRI SPB	MCX I/O Module iNET		
dou_enable - dou_wata - eVT2 - eVT3 - eVT4 - eVT5 - eVT6 - eVT7 - eVT7 - eVT6 - eVT7 - eVT7 - eVT7 - eVT7 - eVT8 - eVT10 - eVT11 - eVT11 </td <td>Action (double d</td> <td>lick to edit)</td> <td>Event (double click to edit)</td> <td>Trigger (double click to edit)</td> <td></td>	Action (double d	lick to edit)	Event (double click to edit)	Trigger (double click to edit)	
PTU_TC program trace enable dtu_dat_trig_1 ALWAYS dtu_dat_trig_2 ALWAYS dtu_dat_trig_3 ALWAYS dtu_dat_trig_3 ALWAYS	dcu_enable dcu_sync dtu_wdat dtu_wdat dtu_wadr dtu_radr ptu_nesting ptu_sync otu_enable_ otu_sync wtu_enable_1 wtu_enable_3 wtu_enable_4 wtu_enable_5 wtu_enable_6 wtu_enable_7 tc_act_0 tc_act_1 tc_act_1 tc_act_3 tsu_rel	ALWAYS 	EVT0 - EVT1 - EVT2 - EVT3 - EVT4 - EVT5 - EVT6 - EVT7 - EVT8 - EVT9 - EVT10 - EVT10 - EVT11 - EVT12 - EVT12 - EVT14 - EVT15 -	core crevt core_trig0 core_trig1 core_trig2 core_trig3 core_trig4 core_trig5 core_trig6 core_trig7 core_trig7 core_trig7 core_trig7 core_trig1 core_trig1 core_trig1 core_trig2 core_trig1 core_trig2 core_trig1 core_trig2 co	Â
	Advanc	ed		dtu_dat_trig_2 ALWAYS dtu_dat_trig_3 ALWAYS dtu_acc_wr	

Figure 43: POB_X Configuration to trace the Program Flow of Core 0

Trigger: not used Event: not used Action:

• ptu_enable: ALWAYS (unconditional program flow trace)

How to handle Program Flow Trace Overflows?

For program trace on CPU2, a first attempt also used unconditional program trace, i.e. ptu_enable = ALWAYS. However, this trace configuration generated immediate AGBT trace buffer overflows. Further analysis reveals that the root cause for the overflow is the execution of a short software loop, while the OS idle task is running on CPU2. This software loop is implemented in the OS function "Os_Hal_CoreFreeze".

Such short software loops are generally rather "trace unfriendly" as they generate program trace message at a high rate and therefore tend to overflow trace buffer/interfaces.

As shown in Figure 44 the "Os_Hal_CoreFreeze" function implements a simple wait loop by jumping to itself. Each jump generates a program trace message (IPI_C message).

Trace	2					
ø	💥 🌱 🕐	₽ ₽	Os_Hal_Co	reFreeze		
	Number	Address	Data	Content	Time	SourceID
	27.7	8000'0E6A	0000 003C	"Os_Hal_Core.c"::143 Os_Hal_CoreFreeze 3C00 j Os_Hal_CoreFreeze (80000E6A) Instruction	512 ns	PTU_TCX
	27.8	0060'0002	00000000	IPI_C (coreID=PTU_TCX, bytes=0x0)	512 ns	PTU_TCX
	27.9	0000'0000	00000000	tick ()	525 ns	PTU_TCX
	28.0	8000'0E6A	0000 003C	"Os_Hal_Core.c"::143 Os_Hal_CoreFreeze 3C00 j Os_Hal_CoreFreeze (80000E6A) Instruction	525 ns	PTU_TCX
	28.1			IPI_C (coreID=PTU_TCX, bytes=0x0)	525 ns	PTU_TCX
	28.2			tick ()	537 ns	PTU_TCX

Figure 44: Program trace recording of the software loop implemented in "Os_Hal_CoreFreeze".

In order to avoid these overflows, we can either add NOP instructions within the loop (to reduce the trace message generation rate) or we exclude this software loop of the Idle task from the program flow trace. Modification of the source code is often not possible, so we go for the option excluding the loop from program trace. In other words, the POB connected to CPU2 is configured in a way that it generates program trace messages for all code areas, except when the core executes code of the function body of the function "Os Hal CoreFreeze".

A magnitude (i.e. address range) compactor of the POB is used to generate a trigger when the CPU instruction pointer (address of executed instruction) falls into the address range of "Os_Hal_CoreFreeze".

ptu_trig_0 [Os_Hal_CoreFreeze]	
Trigger	×
X <= IP <= Y ~	ОК
X Os_Hal_CoreFreeze	Cancel
Entire Object	
Υ	

Figure 45: PTU address range trigger covering the "Os_Hal_CoreFreeze" object.

The trigger (ptu_trig_0) is mapped to event EVT0 and inverted (NOT). Thus, the Event is active while the CPU executes instruction outside of the "Os_Hal_CoreFreeze" function body.

Event (double click to edit)						
EVT0 !ptu_trig_0	с с					
Event						
⊡ Enabled						
NOT Trigger						
core_trig0						
core_trig1						
☑ ☑ ptu_trig_0	[Os_Hal_CoreFreeze]					

Figure 46: ptu_trig_0 is mapped to EVT0 and inverted (NOT)

Finally, the event EVTO is mapped to the Program Trace enable action (ptu_enable). While (Level = State) the event EVTO is active (Qualifier = Active), program trace is enabled.

Action	Action (double click to edit)							
dcu_enable - dcu_sync - dtu_wdat EVT11 EVT10 dtu_wadr EVT11 EVT10 dtu_radr - ptu_enable EVT0								
Act	Action							
	Qualifier Level Event							
IF	Active	\sim	State	\sim	EVT0	\sim		
OR	NEVER	\sim	Edge	\sim	EVT0	\sim		
OR	NEVER	\sim	Edge	\sim	EVT0	\sim		
			- 1					

Figure 47: Enabling Program Trace while EVTO is active.

Figure 48 shows the required POB `	Y configuration (POB Y	' is connected to CPU2).
------------------------------------	------------------------	--------------------------

ion (double dick to edit) Event (double dick to edit) u_sync - u_wdat EVT11 EVT10 u_wdat EVT11 EVT10 u_radie EVT0 u_radie EVT1 u_radie EVT0 u_sync - u_uenable_1 - u_uenable_3 - u_uenable_4 - u_uenable_5 - u_uenable_6 - u_uenable_7 - act_1 - act_2 - j_rel -	Trigger (double dick to edit) core_trig0 core_trig1 core_trig2 core_trig2 core_trig3 core_trig5 core_trig6 core_trig7 core_troevt core_sevt core_sevt core_sevt core_a fine ptu_trig_1 ALWAYS ptu_trig_3 ALWAYS ptu_trig_3 ALWAYS ptu_trig_1 ALWAYS ptu_trig_1 ALWAYS ptu_trig_2 ALWAYS ptu_trig_1 ALWAYS otu_trig_1 ALWAYS otu_trig_1 ALWAYS otu_trig_1 ALWAYS otu_trig_2 ALWAYS otu_trig_1 ALWAYS otu_trig_2 ALWAYS otu_trig_1 ALWAYS otu_trig_1 ALWAYS otu_trig_2 ALWAYS otu_trig_2 ALWAYS dtu_ea_trig_3 [(OsCfg_Trace_OsCore_Core2_Dyn).Curr dtu_ea_trig_6 ALWAYS dtu_ea_trig_6 ALWAYS dtu_ea_trig_0 ALWAYS dtu_ea_trig_0 ALWAYS dtu_ea_trig_0 ALWAYS dtu_ea_trig_6 ALWAYS dtu_ea_trig_1 ALWAYS dtu_dat_trig_1 ALWAYS dtu_dat_trig_1 ALWAYS

Figure 48: POB_Y Configuration to trace the Program Flow and Data Write Accesses to the OS Objects of Core 2

Trigger:

- Two magnitude comparators (address range comparators) of the Data Trace Unit (DTU) of the POB generate a trigger when CPU2 access the corresponding CurrentTask (Running Task) and CurrentIsr (RunningISR2) variables of the OS.
- A magnitude comparator (address range comparator) of the Program Trace Unit (PTU) of the POB generates a trigger when CPU2 executes an instruction located in the address range covered by the function "Os_Hal_CoreFreeze".

Events:

- Trigger dtu_ea_trig_2 is mapped to EVT10.
- Trigger dtu_ea_trig_3 is mapped to EVT11.
- Trigger ptu_trig_0 is inverted (!) and mapped to EVT0.

Actions:

- EVT10 or EVT11 both cause capturing (by DTU) the Write Access Data (dtu_wdat) and Write Access Address (dtu_waddr).
- EVTO is mapped to Action ptu_enable, i.e. program trace is enabled while EVTO is true, i.e. the CPU executed instruction located outside of the function body of function "Os_Hal_CoreFreeze".

Figure 49 shows the required BOB_SRI configuration. In this configuration the Data Trace Unit 1 (DTU1) of the BOB_SRI is used to trace the CurrentTask and CurrentIsr object of CPU0, DTU2 is used to trace the CurrentTask and CurrentIsr object of CPU1.

Trigger - [Advanced Coverage Trigger]		×
MCDS TriCore X TriCore Y SRI SPB MCX I/	O Module iNET	
Action (double click to edit)	Event (double dick to edit)	Trigger (double click to edit)
dcu_enable - dcu_sync - dtu1_wdat EVT0 EVT1 dtu1_wdat EVT0 EVT1 dtu1_rdat - dtu1_radr - dtu2_wdat EVT15 EVT14 dtu2_wdat EVT15 EVT14 dtu2_radr - wtu_enable_1 - wtu_enable_2 - wtu_enable_3 - wtu_enable_6 - wtu_enable_7 - sri_act_0 - sri_act_1 - sri_act_4 - sri_act_5 - sri_act_6 - sri_act_7 - tsu_rel -	EVT0 dtu1 ea trig 0 EVT1 dtu1_ea_trig_1 EVT2 - EVT3 - EVT3 - EVT5 - EVT5 - EVT5 - EVT7 - EVT8 - EVT9 - EVT10 - EVT10 - EVT11 - EVT12 - EVT13 - EVT14 dtu2_ea_trig_0 EVT15 dtu2_ea_trig_1	dtu1 es fine dtu2 es fine dtu1 es_tig_0 [(OsCfg_Trace_OsCore_Core0_Dyn).CurrentTidu1 es_tig_1 dtu1 es_tig_1 1[(OsCfg_Trace_OsCore_Core0_Dyn).CurrentTidu1 es_tig_2 dtu1 es_tig_2 ALWAYS dtu1 es_tig_2 ALWAYS dtu1 dat_tig_1 ALWAYS dtu1_dat_tig_3 ALWAYS dtu1_dat_tig_3 ALWAYS dtu1_acc_tig_1 Access dtu1_acc_tig_1 Access dtu1_acc_tig_1 Access dtu1_acc_tig_2 Access dtu2_es_tig_1 I(OsCfg_Trace_OsCore_Core1_Dyn).CurrentTidu2 es_tig_1 dtu2_es_tig_1 ALWAYS dtu2_dat_tig_1 ALWAYS dtu2_dat_t
Wizard Treate Template		OK Cancel Help

Figure 49: BOB_SRI Configuration to trace Data Write Accessed of Cores 0 and 1 to the corresponding OS Objects

Trigger:

• Two magnitude comparators (address comparators) generate a trigger when CPU0/1 access the corresponding CurrentTask (Running Task) and CurrentIsr (RunningISR2) variables of the OS.

Configuration of dtu1_ea_trig_0 (Running Task):



Configuration of dtu1_ea_trig_1 (Running ISR2):

Tri	igger		×
	X <= ADDR <= Y	\sim	ОК
x	Core_Core0_Dyn).CurrentIsr		Cancel
	Entire Object		
Y			

DTU2 is configured accordingly.

Events:

- Trigger dtu1_ea_trig_0 is mapped to EVT0.
- Trigger dtu1_ea_trig_0 is mapped to EVT1.
- Trigger dtu2_ea_trig_0 is mapped to EVT14.
- Trigger dtu2_ea_trig_0 is mapped to EVT15.

Actions:

- EVT0 or EVT1 both cause capturing (by DTU1) the Write Access Data (dtu1_wdat) and Write Access Address (dtu1_waddr).
- EVT14 or EVT15 both cause capturing (by DTU2) the Write Access Data (dtu2_wdat) and Write Access Address (dtu2_waddr).

DS TriCore X TriCore Y SRI SPB	MCX INET		
ction (double click to edit)	Event (double click to edit)	Trigger (double click to edit)	
su_rel_en - su_rel_sync - su_abs_en - su_abs_en - su_abs_en - su_abs_en - su_abs_en - su_abs_en - su_abs_en - vtu_enable_1 - vtu_enable_2 - vtu_enable_3 - vtu_enable_5 - vtu_enable_6 - vtu_enable_7 - superserver superser	 ▲ ▲ EVT0 = EVT1 = EVT3 = EVT5 = EVT6 = EVT7 = EVT9 = EVT10 = EVT10 = EVT11 = EVT12 = EVT14 = EVT15 = EVT15 = EVT16 = EVT17 = EVT16 = EVT17 = EVT18 = EVT19 = EVT20 = EVT20 = EVT20 = EVT21 = EVT20 = EVT21 = EVT22 = EVT23 = 	break in fifo_trg_0 fifo_trg_1 ts_trg ts_trg tx_act_0 tx_act_1 tx_act_2 tx_act_1 tx_act_1 tx_act_2 tx_act_1 txy_act_1 txy_act_2 ty_act_1 ty_act_1 ty_act_2 spb_act_0 spb_act_1 spb_act_2 spb_act_1 sr_act_1 sr_act_2 sr_act_3 sr_act_4 sr_act_5 sr_act_6 sr_act_6 sr_act_6 sr_act_7 cnt_trig_1 cnt_trig_5 cnt_trig_5 cnt_trig_6 cnt_trig_10 cnt_trig_10 cnt_trig_11 cnt_trig_12	

Figure 50 shows the MCX configuration required to use TICK time stamping.

Figure 50: MCX Configuration to generate TICKS for Time Stamping and Upload-While-Sampling

As a final stage of the trace configuration the Recorder, i.e. iC5700, properties need to be set. In this use-case we immediately start trace recording.

Figure 51 depicts the corresponding Recorder settings.

Analyzer C	nalyzer Configuration - [PT_CPU_0_2_OS_ORTI]								
Hardware	Profiler	Coverage							
Analysis	and Confi	iguration							
🗹 Pro	filer								
Cov	verage								
🗹 Mar	nual Trigge	er/Recorder configuration	Configure	Preset >					
	FNet	Operation							
Options Rea	Options Reactivate session after CPU stop								
Property			Value						
E Rec	order								
_			Triane and the second second						
Start			Ingger immediately						
Start Rece	ording Size	Limit	1 GB						
Start Reco Trigg	ording Size jer Position	: Limit	1 GB Begin						
Start Reco Trigg Time	ording Size jer Position r Interpolat	Limit 1 tion	1 GB Begin						

Figure 51: Analyzer Configuration for immediate Trace Recording

2.2.4 winIDEA Profiler Configuration

In order to make winIDEA and the winIDEA Profiler aware of the AUTOSAR OS running on the target the so-called ORTI file, generated by the AUTOSAR generation tool, needs to be imported into winIDEA. This is done via the menu "Debug – Operating System...".

When importing the AUTOSAR ORTI file via the "New..." button, the OS type "OSEK AUTOSAR" has to be selected (see Figure 52). Afterwards you can give the OS awareness some descriptive name. In our example shown in Figure 53, the ORTI file has the name "Os_Trace.ORT". As the AUTOSAR OS used in this example is a Vector Microsar OS, we name the OS-awareness the name "Microsar ORTI".



Figure 52: Creating of a new OSEK AUTOSAR OS Awarenss

Operating system		×	E	dit options	
Operating System				Property	Value
OSEK AUTOSAR] Microsar ORTI	New			Configuration	
	Configure		RTOS description file type	ORTI	
			RTOS description file location	Os_Trace.ORT	
	comgarem				
	Rename				

Figure 53: Selection of the AUTOSAR ORTI File (in this example the file "Os_Trace.ORT")

A "Download" or "Symbol Download" operation also reads in the ORTI file.

Subsequently, the winIDEA Profiler can be configured to utilize the information given by the ORTI file. As shown in Figure 53, the profiling of "OS objects" needs to be enabled.

Analyzer Configuration - [PT_CPU_0_2_C	DS_ORTI] ×
Hardware Profiler Coverage	
Profile	RTOS Profiler Options
Advanced	Operating System
☑ OS objects OS Setup	Objects to profile
	TASKs.Core0
✓ Network Passpoint	
Enter filter string(s)	✓ISR2s.Core1 ✓ISR2s.Core2 Core 0: SERVICETRACE[0] ✓
	Object Info:
	Name: RUNNINGTASK[0] Definiton: RUNNINGTASK[0] Description: TASKs.Core0 Signaling: (OsCfg_Trace_OsCore_Core0_Dyn).Current'
Data Areas	

Data Areas

Figure 54: "OS Setup..." Configuration of the Analyzer – Profiler Dialog

Pushing the "OS Setup..." button opens the "RTOS Profiler Options" dialog. The name of the Operating Systems corresponds to the name that was given via the "Debug – Operating System..." dialog, when selecting the ORTI file.

All OS objects described by the ORTI are listed under "Objects to profile". One or multiple objects can be selected for profiling. The sub-window "Object Info:" provides more detailed info of a selected object, such as the Name given in the ORTI file and the type of signaling.

In the example shown in Figure 54 the ORTI object RUNNINGTASK[0], i.e. the Running Task on CPU0, is signaled via the global variable Os_Cfg_Trace_OsCore0_Dyn.CurrentTask. As described in the section 2.1.2 these are the global variables which need to be observed via Data Trace.

In addition to OS objects, also the entire code shall be profiled. This requires the following configurations.

The "Profile - Code" option needs to be enabled and also a Code Profiler Operation mode needs to be selected.

In general, the following function profiling modes are supported:

Operating Mode	Concepts	Description
Entry/Exit	Function Nesting	Profiling in Entry/Exit mode also recognizes function nesting.
	Operating System	If a preemptive Operating System (OS) is used, it is required to also signal and profile the context switches of the OS. The profiler basically maintains a function call stack for each recognized OS context.
	Compiler Optimization	Entry/Exit mode profiling does not cater for function exit optimizations, i.e. function exit optimizations may yield this profiling mode unusable.
Flat	Function	Profiling in Flat mode does not recognize function nesting. It
	Nesting	assume a valid entry/exit sequence for each function.
	Operating	In Flat mode it is not necessary to also profile the context switches
	System	of an OS.
	Compiler Optimization	Flat mode profiling is not affects by compiler optimizations.
Range	Function Nesting	Profiling in Range mode also recognizes function nesting.
	Operating System	If a preemptive Operating System (OS) is used, it is required to also signal and profile the context switches of the OS. The profiler basically maintains a function call stack for each recognized OS context.
	Compiler Optimization	Range mode profiling performs an in-depth analysis of the code trying to recognize and compensate compiler optimizations, such a function tail optimizations.

Analyzer Configuration - [PT_CPU_0_2_OS_0	ORTI]	×
Hardware Profiler Coverage		
Profile Code Advanced Data OS objects OS Setup AUX Network Passpoint Code Areas Enter filter string(s)	Operation mode Trigger at (Default) Flat Range Analyze only events after trigger point Limit session duration 1000 ms Ignore unknown functions / variables	t
Data Areas	New > Edit Remove	

Figure 55: Profiler Configuration of OS Profiling and Code (Function) Profiling

2.2.5 winIDEA Profiler View

Figure 56 and Figure 57 show the resulting profiler timeline. The "Code" section shows the profiler timeline of functions, the "Data" section displays the timeline of the OS tasks and ISR2s.



Figure 56: Sample Timeline of a dual-core OS (task and ISR2) and Code (Function) Profile



Figure 57: Zoomed-In section of the sample Profiler Timeline shown in Figure 56 (Profiler Operating Mode = Range)

2.3 Function Specific Program Trace

Full program flow trace might not be feasible in situations where an AGBT trace interface is not available. In these situations, Function-Specific Program Trace can give insight into the behavior of a specific function. We can avoid overflowing the trace buffer by cleverly configuring the trace buffer while still getting sufficient Program Trace to resolve an issue.

Let's say we want to record the function *cO_RunA* and all its subfunctions. First, we pick a POB for our analysis, in this case, POB X. We map POB X to Core 0 because that's where our function executes. Then, we configure two events, EVT0 and EVT1, for the entry and exit point of the function. The events create a cross-trigger action that the MCX receives (see 1 in Figure 59).

In the MCX menu, the events from POB X increment and clear a counter. The same counter maps back to tcx_trig_0 via EVT7. Finally, if we go back to Figure 58, we use that trigger to enable the program trace.

To summarize, we increment a counter at the beginning of our function of interest and decrement it at the exit. Whenever the counter is active, meaning non-zero, it enables Program Trace. That has the effect that we record the function, as well as all its subfunctions.



Figure 58: We use function entry and exit triggers to increment and clear an MCX counter. We can then use the MCX counter to enable Program Trace.



Figure 59: We use the entry and exit event from POB X to increment and clear an MCX counter. We then provide the state of the counter back to POB X to enable the Program Trace.

3 Technical support

3.1 Online resources



3.2 Contact

Please visit <u>https://www.isystem.com/contact.html</u> for contact details.

iSYSTEM has made every effort to ensure the accuracy and reliability of the information provided in this document at the time of publishing. Whilst iSYSTEM reserves the right to make changes to its products and/or the specifications detailed herein, it does not make any representations or commitments to update this document.

© iSYSTEM. All rights reserved.