

---

---

## Technical Notes

# Infineon TriCore Family On-Chip Emulation

## Contents

Contents.....	1
1 Introduction .....	2
2 Emulation options.....	3
2.1 Hardware Options.....	3
2.2 Initialization Sequence .....	4
2.3 Initialization After Download Sequence.....	5
2.4 JTAG Scan Speed.....	6
3 CPU Setup .....	7
3.1 General Options.....	7
3.2 Debugging Options.....	8
3.3 Advanced Options .....	9
3.4 Events .....	10
4 Internal FLASH Programming .....	10
5 Real-Time Memory Access .....	11
6 Access Breakpoints .....	11
7 Peripheral Controller Processor (PCP).....	14
7.1 winIDEA Workspaces for TriCore and PCP .....	14
7.2 Usage Notes.....	14
7.3 Reserved Resources.....	14
8 Trace.....	16
8.1 Background.....	16
8.1.1 Trace Memory (EMEM).....	18
8.1.2 Multi-Core Debug Solution (MCDS) .....	18
8.2 Trace Configuration.....	21
8.2.1 Record everything.....	21
8.2.2 Use Trigger/Qualifier .....	21
9 Profiler.....	30
10 Execution Coverage.....	34
11 Getting Started.....	36
12 Troubleshooting.....	36

# 1 Introduction

Devices based on Infineon Tricore V1.3.1 core architecture belonging to the AUDO FUTURE and AUDO MAX series are supported. The development tool connects to on-chip OCDS (On-Chip Debug Support) via four wire (or five) JTAG or two wire DAP debug interface. Major difference is that DAP interface requires less CPU pins while there is no difference in terms of debug functionalities between the two interfaces.

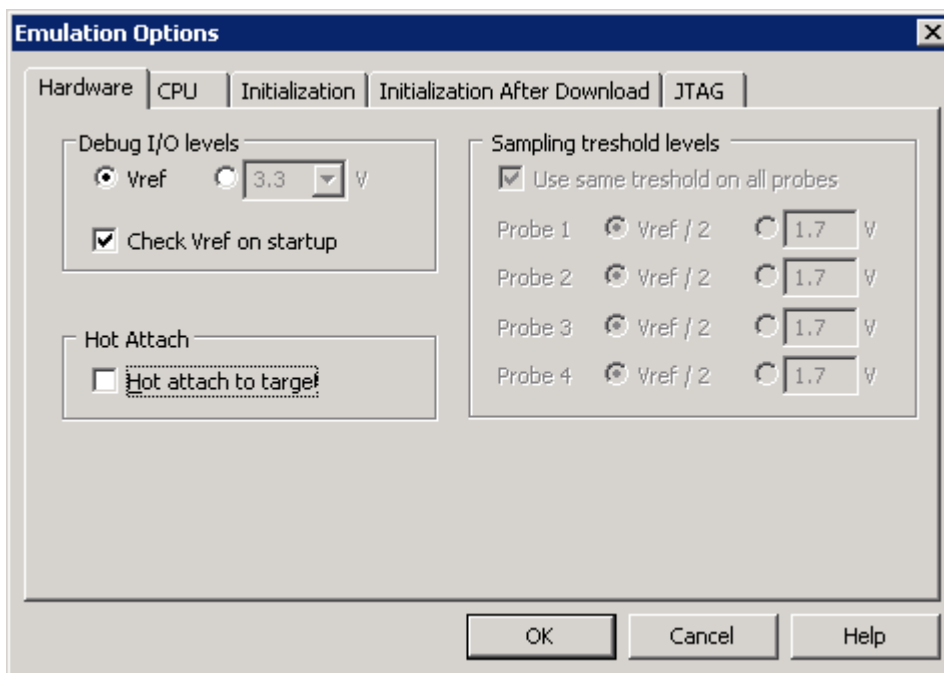
Due to the lack of the trace functionality on standard TriCore device, Infineon provides a dedicated pin compatible Emulation Device (ED), where trace, based on 256KB on-chip trace buffer, is available. This allows using the Emulation Device on a regular target during the development process when the trace functionality might be required.

## Debug Features

- DAP and JTAG debug interface
- 4 on-chip hardware execution breakpoints (TriCore V1.3.1)
- Unlimited software breakpoints
- Access breakpoints
- Real-time access
- Internal/external flash programming
- Trace, Profiler and Execution Coverage on Emulation Device (ED)

## 2 Emulation options

### 2.1 Hardware Options



*Emulation options, Hardware pane*

#### **Debug I/O levels**

iC3000 development system can be configured in a way that JTAG or DAP debug signals are driven at 3.3V, 5V or target voltage level (Vref). In case of iC5000 development system, arbitrary voltage can be set beside the listed 2.5, 3.3V and 5V by simply writing custom value e.g. 2.8V in the same field.

When 'Vref' Debug I/O level is selected, a voltage applied to the belonging reference voltage pin on the target debug connector is used as a reference voltage for voltage follower, which powers buffers, driving the debug JTAG/DAP signals. The user must ensure that the target power supply is connected to the Vref pin on the target JTAG/DAP connector and that it is switched on before the debug session is started. If these two conditions are not met, it is highly probably that the initial debug connection will fail already. However in some cases it may succeed but then the system will behave abnormal.

It is recommended to use 'Vref' setting for target Vref voltages 3.3V and below. When debug I/O levels should be 5V, it is recommended selecting internal '5.0V' source for Debug I/O levels.

#### **Check Vref on startup (iC5000 only)**

iC5000 development system can measure voltage on the Vref pin on the target debug connector. When 'Vref' Debug I/O level is selected, the debugger will pop up a warning message if no voltage is detected on the target debug connector.

#### **Hot Attach**

Option must be checked when Hot Attach is used. Refer to the 'Hot Attach' chapter for more details on Hot Attach use.

## 2.2 Initialization Sequence

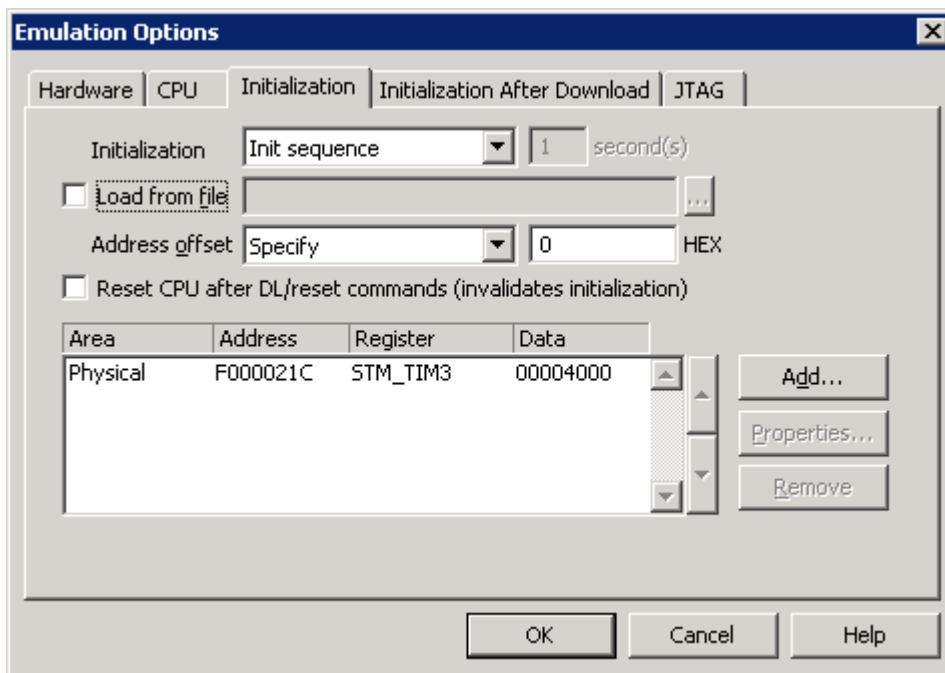
Before the flash programming or download can take place, the user must ensure that the memory is accessible. This is very important since there are many applications using memory resources (e.g. external RAM, external flash), which are not accessible after the CPU reset. In that case, the debugger must execute after the CPU reset a so called initialization sequence, which configures necessary CPU chip selects and then the download or flash programming can actually take place. The user must set up the initialization sequence based on his application.

Note: Keep the default 'Specify' and '0' setting for Address offset within the Initialization tab.

No initialization sequence is required for programming Tricore internal program flash.

The initialization sequence can be set up in two ways:

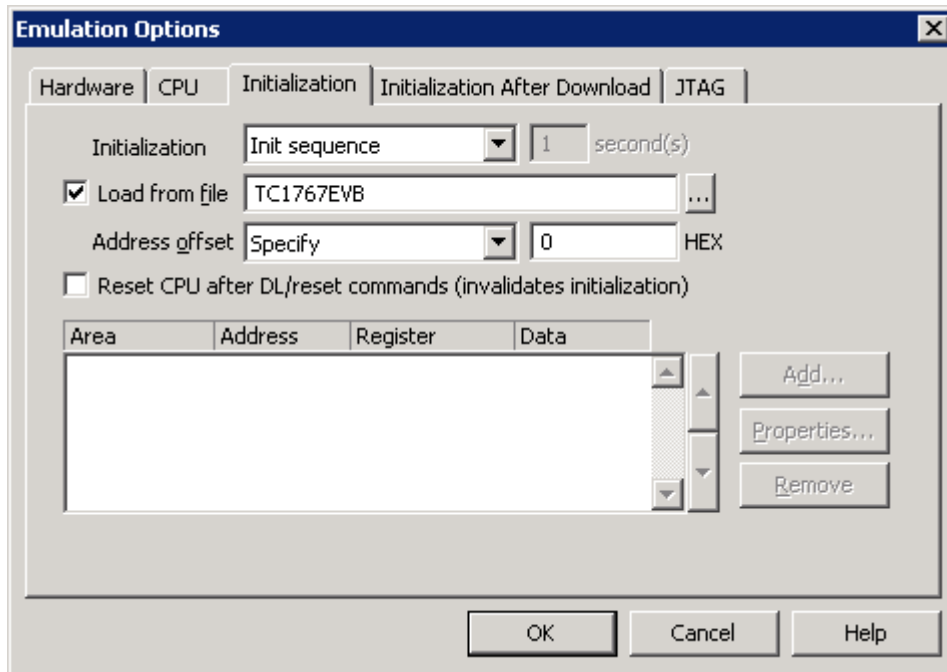
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



2. winIDEA accepts initialization sequence as a text file with .ini extension. The file must be written according to the syntax specified in the appendix in the hardware user's guide. Pressing F1 in the Initialization dialog opens a document describing .ini file syntax.

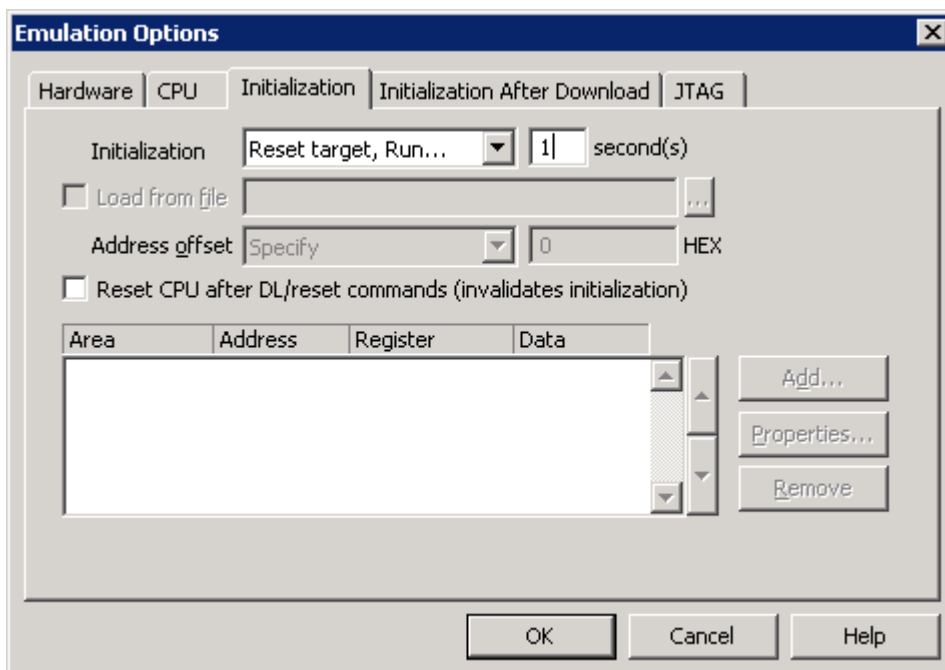
Excerpt from an example TC1767EVB.ini file for the Infineon TC1767:

```
S STM_TIM3 L 0x4000 //load TIM3 register
```



The advantage of the second method is that you can simply distribute your .ini file among different workspaces and users. Additionally, you can easily comment out some line while debugging the initialization sequence itself.

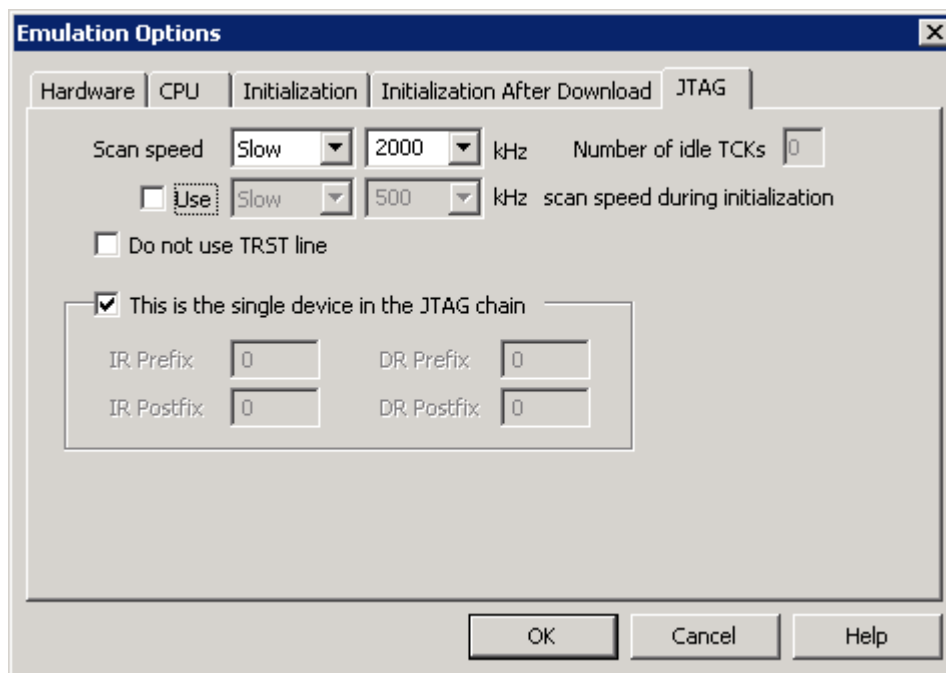
There is also a third method, which can be used too but it's not highly recommended for the start up. The user can initialize the CPU by executing part of the code in the target ROM for X seconds by using 'Reset and run for X sec' option.



### 2.3 Initialization After Download Sequence

There are rare cases when a different initialization of the CPU is required for debug download and after the debug download. When Initialization After Download is used, typically the 'Reset CPU after DL/reset commands (invalidates initialization)' option described in the previous chapter is checked too.

## 2.4 JTAG Scan Speed



*JTAG Scan Speed definition*

---

Note: These settings are not available when DAP debug interface is used.

---

### *Scan speed*

The JTAG chain scanning speed can be set to:

- Slow - long delays are introduced in the JTAG scanning to support the slowest devices. JTAG clock frequency varying from 1 kHz to 2000 kHz can be set.
- Fast – the JTAG chain is scanned with no delays.
- Other scan speed types (not supported for Tricore family) can be seen and are automatically forced to Slow.

Slow and Fast JTAG scanning is implemented by means of software toggling the necessary JTAG signals.

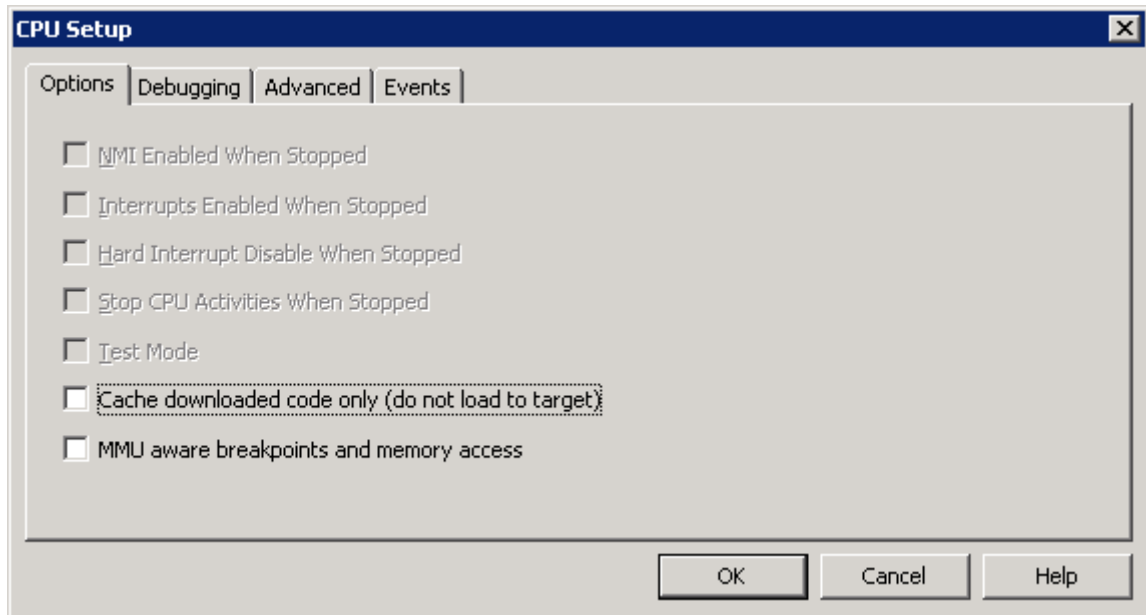
In general, Fast mode should be used as a default setting. If Fast mode fails or the debugging is unstable, try Slow mode at different scan frequencies until you find a working setting.

### *Use – Scan Speed during Initialization*

On some systems, slower scan speed must be used during initialization, during which the CPU clock is raised (PLL engaged) and then higher scan speeds can be used in operation. In such case, this option and the appropriate scan speed must be selected.

## 3 CPU Setup

### 3.1 General Options



*Tricore Family Debugging Options*

#### ***Cache downloaded code only (do not load to target)***

When this option is checked, the download files will not propagate to the target using standard debug download but the Target download files will.

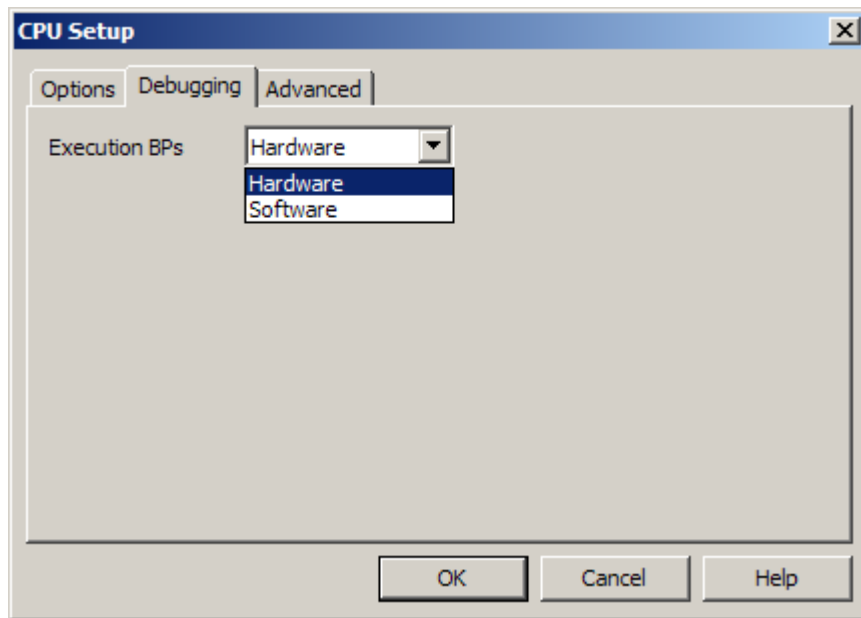
In cases, where the application is previously programmed in the target or it's programmed through the flash programming dialog, the user may uncheck 'Load code' in the 'Properties' dialog when specifying the debug download file(s). By doing so, the debugger loads only the necessary debug information for high level debugging while it doesn't load any code. However, debug functionalities like ETM and Nexus trace will not work then since an exact code image of the executed code is required as a prerequisite for the correct trace program flow reconstruction. This applies also for the call stack on some CPU platforms. In such applications, 'Load code' option should remain checked and 'Cache downloaded code only (do not load to target)' option checked instead. This will yield in debug information and code image loaded to the debugger but no memory writes will propagate to the target, which otherwise normally load the code to the target.

#### ***MMU aware breakpoints and memory access***

If checked, breakpoints are set with physical addresses, memory accesses are performed using virtual/physical mapping.

If the option is cleared, no distinction is made between physical and virtual addresses.

## 3.2 Debugging Options



*Tricore Family Debugging Options*

### ***Execution Breakpoints***

#### *Hardware Breakpoints*

Hardware breakpoints are breakpoints that are already provided by the CPU. The number of hardware breakpoints is limited to four. The advantage is that they function anywhere in the CPU space, which is not the case for software breakpoints, which normally cannot be used in the FLASH memory, non-writable memory (ROM) or self-modifying code. If the option 'Use hardware breakpoints' is selected, only hardware breakpoints are used for execution breakpoints.

Note that the debugger, when executing source step debug command, uses one breakpoint. Hence, when all available hardware breakpoints are used as execution breakpoints, the debugger may fail to execute debug step. The debugger offers 'Reserve one breakpoint for high-level debugging' option in the Debug/Debug Options/Debugging' tab to circumvent this. By default this option is checked and the user can uncheck it anytime.

---

Note: Memory protection unit must not be used while debugging the application. Hardware breakpoints and instruction step cannot be used in conjunction with Memory Protection Unit.

---

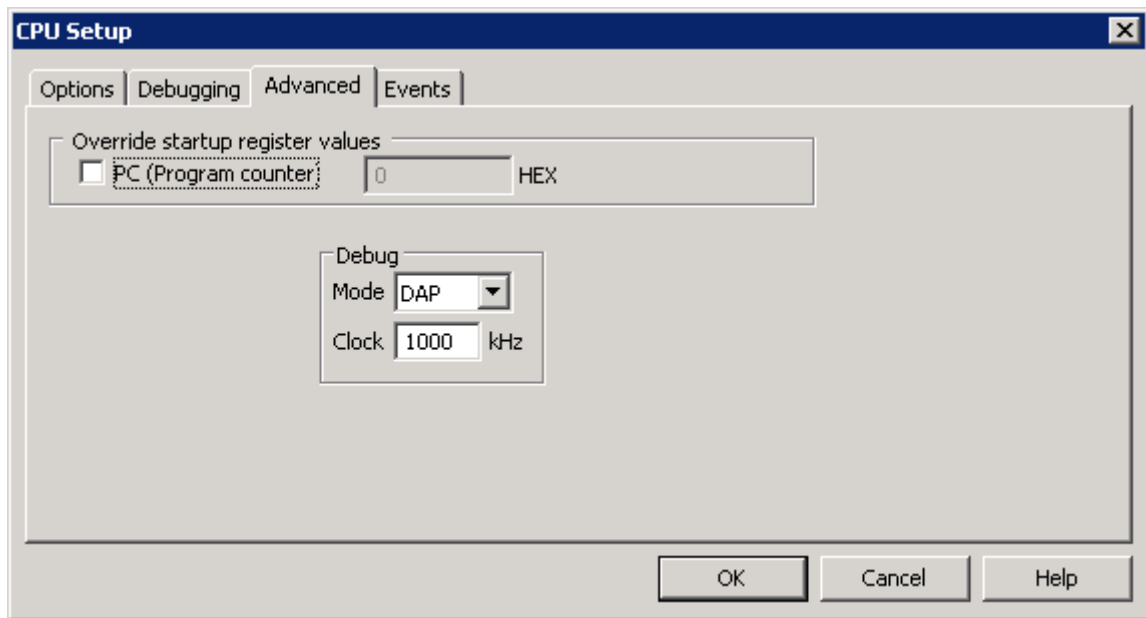
#### *Software Breakpoints*

Available hardware breakpoints often prove to be insufficient. Then the debugger can use unlimited software breakpoints to work around this limitation.

When a software breakpoint is being used, the program first attempts to modify the source code by placing a break instruction into the code. If setting software breakpoint fails, a hardware breakpoint is used instead.

The debugger provides software breakpoints in the internal program flash too but they are almost useless since only erasing a single erase unit can take up to 5s already.

### 3.3 Advanced Options



*Tricore Advanced Emulation Options*

#### ***Override startup register values***

This option overrides the default Instruction Pointer reset value with the value set.

#### ***Debug***

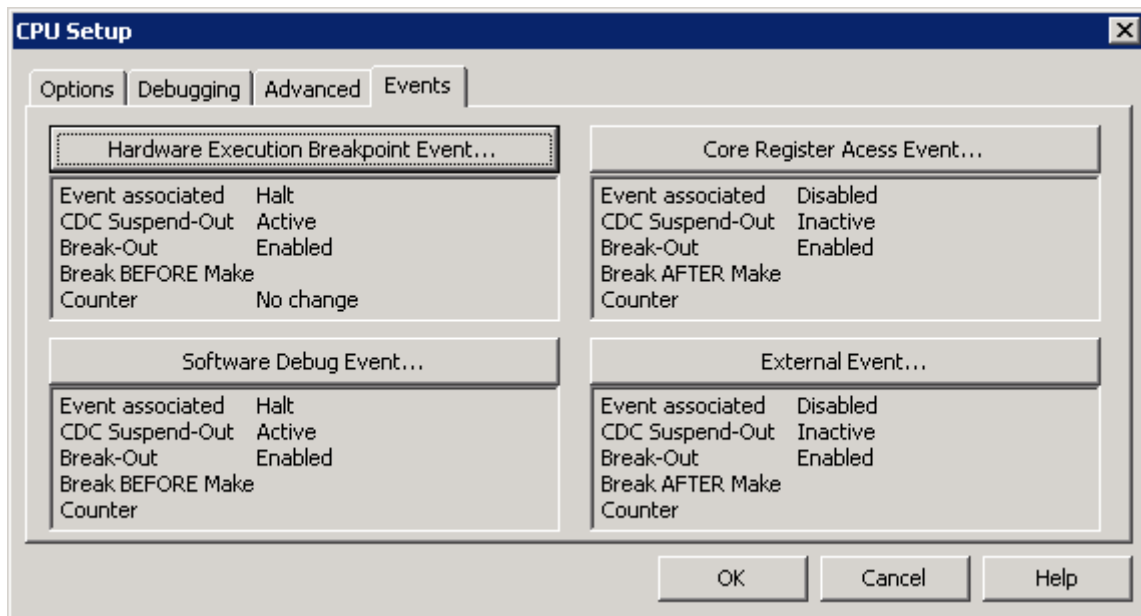
Tricore device can feature JTAG, DAP or both debug interfaces. JTAG or DAP debug mode must be selected depending on the debug interface used. Additionally, DAP clock must be selected when DAP debug interface is used. Typical DAP clock is in range between 2 and 4MHz.

---

Note: The debugger must be connected to the target DAP connector (10-pin 1.27 mm pitch) when DAP debug mode is selected and to the target JTAG connector (16-pin 2.54mm pitch) when JTAG debug mode is selected.

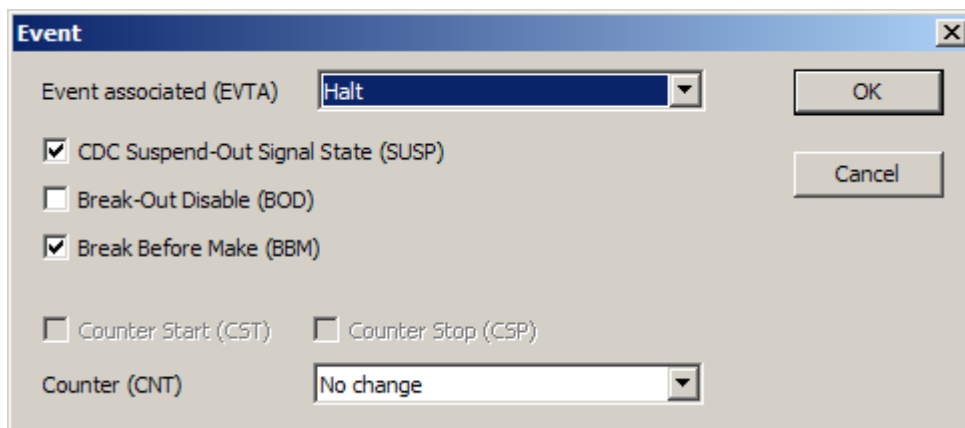
---

### 3.4 Events



It is possible to configure individual event actions for hardware execution breakpoint event (TROEVT, TR1EVT), core register access event (CREVT), software debug event (SWEVT) and external break input event (EXEVT).

Press the button to open the Event dialog.



Refer to Access Breakpoints chapter for available event actions and explanation.

## 4 Internal FLASH Programming

Internal CPU program flash is programmed through the standard debug download. The debugger recognizes which code from the download file fits in the internal flash and programs it during the debug download.

A standard FLASH setup dialog accessible from the FLASH menu is used only for programming external flash devices.

## 5 Real-Time Memory Access

Tricore debug module supports real-time memory access. Watch window's Rt.Watch panes can be configured to inspect memory with minimum intrusion while the application is running. Optionally, memory and SFR windows can be configured to use real-time access as well.

In general it is not recommended to use real-time access for Special Function Registers (SFRs) window. In reality, real-time access still means stealing some CPU cycles. As long as the number of real-time access requests stays low, this is negligible and doesn't affect the application. However, if you update all SFRs or memory window via real-time access, you may notice different application behavior due to stealing too many CPU cycles.

When a particular special function register needs to be updated in real-time, put it in the real-time watch window (don't forget to enable real-time access in the SFRs window but keep SFRs window closed or open but with SFRs collapsed). This allows observing a special function register in real-time with minimum intrusion on the application.

Using "alternative" monitor access to update a memory location or a memory mapped special function register while the application is running works like this: the application is stopped, the memory is read and then the application is resumed. Hence the impact on real time execution is severe and use monitor access for 'update while running' only if you are aware of the consequences and can work with them.

## 6 Access Breakpoints

---

Note: Memory protection unit must not be used while debugging the application. Hardware execution breakpoints and access breakpoints cannot be used in conjunction with Memory Protection Unit. Additionally, CLR and CU triggers of

---

The CDC (Core Debug Controller) allows code and data triggers to be combined to create a Debug Event. The combination is specified by the Trigger Event Register (TR0EVT and TR1EVT). The Trigger Event Unit can generate a number of Trigger Debug Events by combining four Debug Triggers for each Trigger Debug Event. The Debug Triggers are generated by the memory protection system. There are four possible Debug Triggers generated by the Memory Protection System:

### ***DLR***

Data read or write access to the lower bound or range of the Data RTEn, as enabled in the Data Protection Mode (DPM) register.

### ***DU***

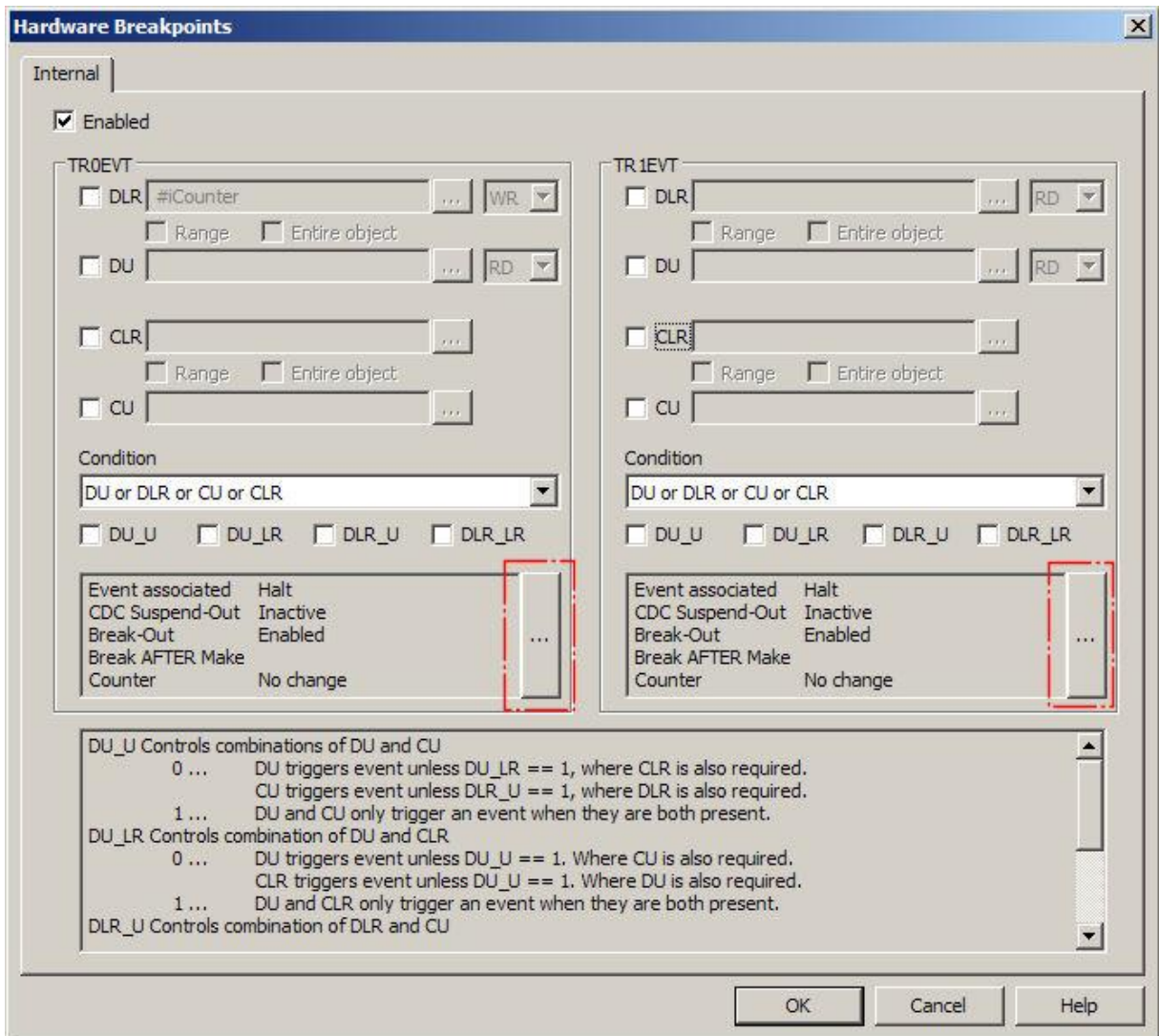
Data read or write access to the upper bound of the Data RTEn, as enabled in the Data Protection Mode (DPM) register.

### ***CLR***

Code execution from the lower bound address or address range of the Code RTEn, as enabled in the Code Protection Mode (CPM) register.

### ***CU***

Code execution from the upper bound address of the Code RTEn, as enabled in the Code Protection Mode (CPM) register.



*Tricore Access Breakpoints*

### **Condition**

Selects debug trigger combination which generates a Debug Event. 16 different combinations are possible.

### **DU\_U, DU\_LR, DLR\_U, DLR\_LR**

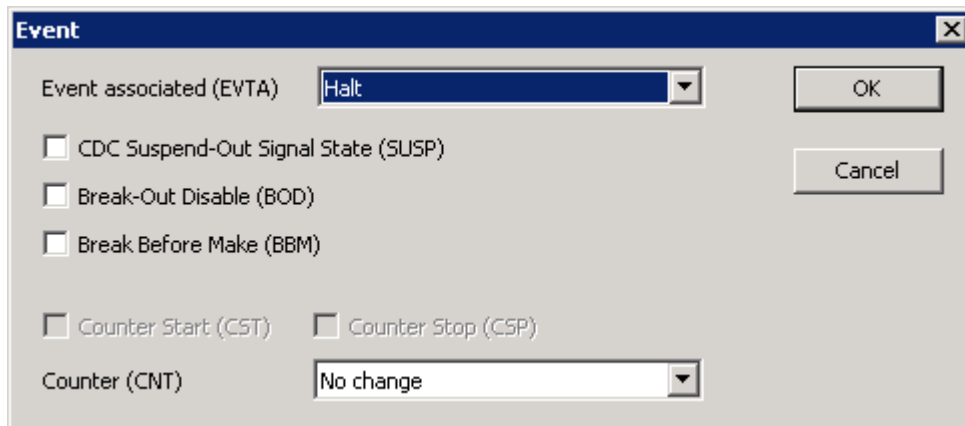
DU\_U controls combinations of DU and CU, DU\_LR controls combination of DU and CLR., DLR\_U controls combination of DLR and CU and DLR\_LR controls combination of DLR and CLR.

In principle these four options are unnecessary because matching setting of the debug trigger combination can be selected in the Condition combo box no matter how these options are set. They are available just not to restrict the user from programming these bit fields in the Trigger Event registers TR0EVT and TR1EVT.

Refer to the Target Specification document of specific Tricore microcontroller for more details on using these options.

### **Event Action**

Different actions can be configured for individual Trigger Event register. Default configuration stops the core only. Event dialog is open by pressing the button in the right bottom corner of the TR0EVT and TR1EVT configuration area (encircled in red in the Tricore Access Breakpoints dialog in the previous page).



### ***Event Associated (EVTA)***

#### ***Disabled***

The event is disabled and no actions occur: the suspend-out signal and performance counter control ignore the event.

#### ***None***

No action is implemented through the EVTA field of the event's register however the suspend-out signal and performance count still occur as normal for an event.

#### ***Halt***

The Debug Action Halt, causes the Halt mode to be entered where no more instructions are fetched or executed. While halted, the CPU does not respond to any interrupts.

#### ***Breakpoint Trap***

The Breakpoint Trap enters a Debug Monitor without using any user resource. Refer to TriCore Core Architecture manual for more details.

#### ***Breakpoint Interrupt***

One of the possible Debug Actions to be taken on a Debug Event, is to raise a Breakpoint Interrupt. The interrupt priority is programmable and is defined in the control register associated with the breakpoint interrupt. Refer to TriCore Core Architecture manual for more details.

### ***CDC Suspend-Out Signal State (SUSP)***

The suspend-out signal is asserted when a debug event occurs. It is up to the user then to configure according peripheral module to act upon asserted suspend-out signal.

### ***Break-Out Disable (BOD)***

When this option is checked, BRKOUT signal is not asserted. This takes priority over any assertion generated by the EVTA field.

### ***Break Before Make (BBM)***

When this option is checked, Halt mode performs a cancel of all instructions after and including the instruction that caused the breakpoint. If unchecked, it cancels all instructions after the instruction that caused the breakpoint.

### ***Counter (CNT)***

When the performance counter is operating in task mode, the counters are started and stopped by debug actions. All event registers allow the counters to either be started or stopped.

The trigger event registers also allow the mode to be toggled to active (start) or inactive (stop). This allows a single Range Table Entry (RTE) to be used to control the performance counter in certain applications.

## 7 Peripheral Controller Processor (PCP)

This chapter discusses debugging of the Infineon Peripheral Control Processor, PCP for short, within the iSYSTEM winIDEA environment. PCP features and debugging methods are device implementation dependent. Supported TriCore devices at the time of this writing (January 2010) are: TC1767, TC1767ED, TC1797 and TC1797ED.

### 7.1 winIDEA Workspaces for TriCore and PCP

Debugging session is started by opening a TriCore winIDEA workspace, e.g. Sample.xjrf. The download file contains both a TriCore run-time image as well as a PCP run-time code image. Set a breakpoint in the `main.c` module after the initialization of the PCP and other related special function registers, timers and I/O ports, for example.

At this point also the PCP run-time image has been copied into the PCP CMEM code memory by the compiler startup code. Now click on Debug/Core/PCP. This will open a new, secondary instance of winIDEA. It will automatically load a PCP workspace. The name of this workspace is defined by the TriCore workspace name of the primary winIDEA, to which a `_PCP` extension is appended. In our case this results to `Sample_PCP.xjrf`.

On Debug/Download winIDEA will load symbols for the PCP debug. It will also display a register context for the channel number 1 by default and disassemble a PCP code, if the PCP has been enabled in the `PCP_CS.EN` control/status register bit.

### 7.2 Usage Notes

Note that PCP instructions can disable debugging a PCP channel by clearing its saved context R7 register, the channel enable bit `R7.CEN`. A warning is displayed if run or step is attempted. To continue debugging, the bit needs to be re-set by hand.

The PCP debugging provides only software breakpoints. There are no hardware breakpoints.

Whenever TriCore CPU is reset and run again, the breakpoints set in the PCP winIDEA session will be overwritten. Open the Breakpoints dialog and click on the Reapply All.

Use F8 (Debug/Snapshot) to refresh windows' contents in the other winIDEA instance when monitoring shared TriCore and PCP memory resources.

Any access to an undefined memory space, for example access beyond the implemented CMEM or PRAM space, causes a bus error in TriCore. To recover, the CPU has to be reset.

When downloading code that has already been programmed into flash, the *Cache downloaded code only* option in the CPU Setup dialog can be used to bypass redundant and slow flash programming. Do, however, remember to uncheck this option whenever the project is recompiled.

In certain cases winIDEA will display the PCP status STOP, when in fact the PCP is running. This happens when a PCP channel program is interrupt-driven from the TriCore. For example, in the BlinkLED sample the interrupt is issued in the order of 500ms, while the PCP program takes only a couple of 100ns to execute. When the PCP execution stops, it waits for another invocation by the TriCore interrupt routine. Therefore, most of the time the PCP is in waiting, and this is perceived by a debugger as being stopped.

### 7.3 Reserved Resources

For the PCP run and step control winIDEA is using the `CPU_SBSRC`, the CPU Software Breakpoint Service Request Control Register, at address `F7E0.FFBCh`. User application should not use this register.

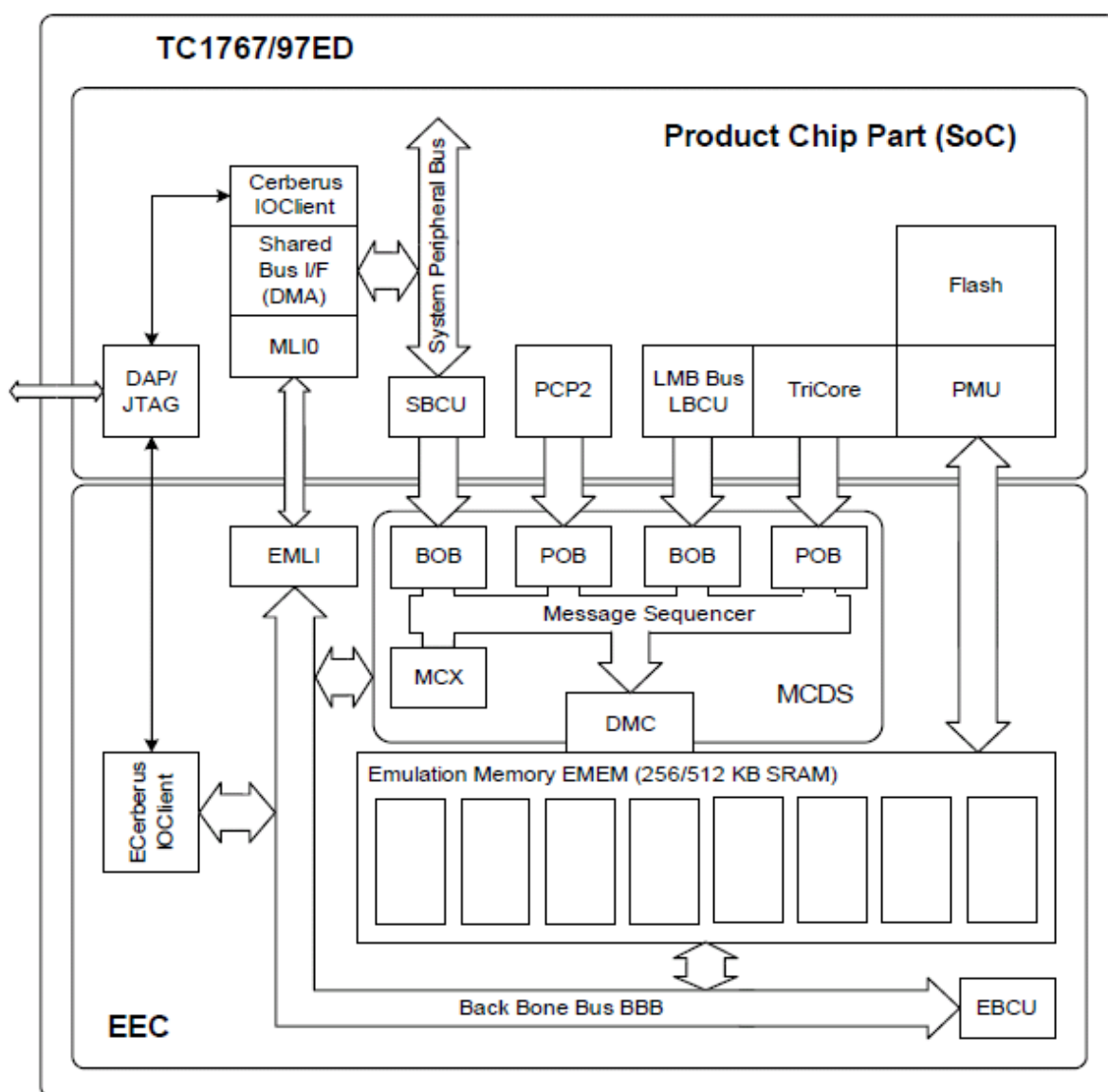
The current PCP channel being debugged is selected with the SRPN bit-field of the CPU\_SBSRC register. When needed, this scope can be changed either in the Disassembly Registers Window, where the channel service request field is marked with a CH, or directly in the SFR window where the register is located in the CDR Core Debug Register Group.

## 8 Trace

Per default, Tricore target processors don't provide debug trace functionality, which is often necessary during the development and test process. As an alternative, Infineon offers a dedicated pin compatible Emulation Device (ED), which features Nexus compliant on-chip trace (MCDS) in conjunction with the standard OCDS debug module, which is controlled by the external tool through the JTAG or DAP debug interface.

### 8.1 Background

Some technical background on the MCDS is fundamental in order to use highly capable but relatively complex Tricore trace. Majority of the text explaining the MCDS is taken from the Infineon documentation and its sole purpose is to get the user acquainted with the MCDS, which will then ease the trace use supported by iSYSTEM development tool. Contact Infineon representative when a detailed knowledge on the Emulation Device is required.



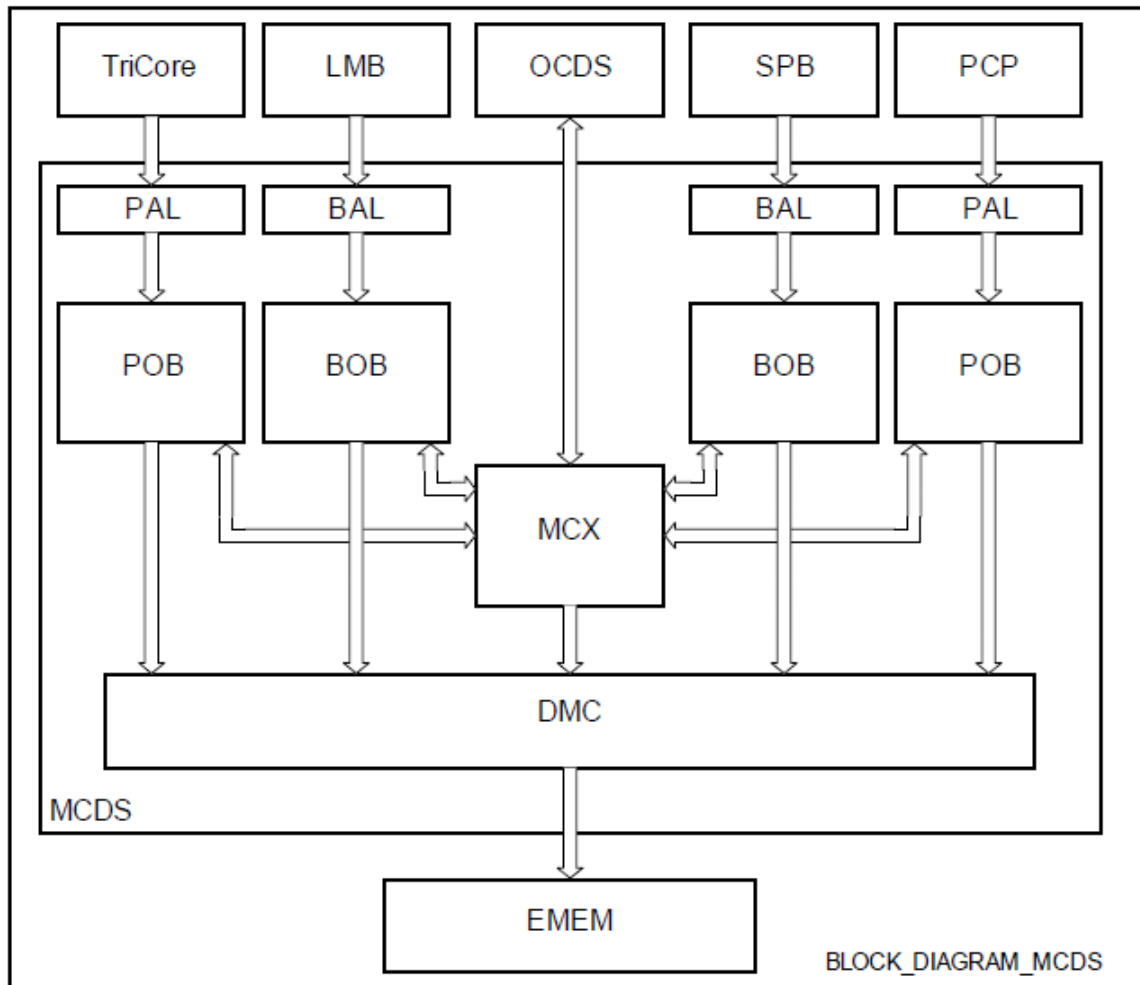
TC1767/TC1797ED block diagram

TC1767/TC1797ED block diagram shows the Product Chip Part, which is part of the standard product chip and EEC block, which exist on ED device only for the emulation and test needs. The product chip part is reduced to selected modules. The EEC part is implemented on additional silicon area located outside of product chip part and simply left away during manufacturing the product chips.

Most important blocks of the EEC part are MCDS and EMEM. A MLI bridge is used to link the FPI bus of product chip (System Peripheral Bus) and FPI bus of EEC (Back Bone Bus).

The Emulation Memory (EMEM) on the EEC is used for two conceptually different purposes: Calibration and Tracing. Tracing is the focus of this chapter.

Next picture shows the external connections of the MCDS module and its top level internal structure.



*MCDS Block diagram*

The prime target for the EEC is to cater for the real-time tracing of the TriCore cores. Nonetheless all event generating logic can be used for breakpoint generation via the central break switch as well. It should be noted that there is a certain latency using this route.

TriCore ED trace features:

- TriCore program trace
- TriCore load/store data trace (no register file trace)
- PCP ownership trace
- PCP program trace
- PCP data write to PRAM trace (no register file trace)
- Full visibility of internal peripheral bus (SPB)
- Full visibility of Local Memory Bus (LMB)

## 8.1.1 Trace Memory (EMEM)

TMEM is a 256kBytes on-chip trace buffer, which stores the trace messages coming out of the trace unit. Valid data in the buffer is uploaded by the external debugger, which then reconstructs the original program flow.

In case of program trace, TMEM stores only information of non-sequential instructions being executed. All sequential instructions between the non-sequential instructions are reconstructed by the debugger exploiting code image information extracted from the debug download file(s). The trace concept does not allow tracing of the self-modifying code. An example of self-modifying code would be a boot-loader, which copies some code in the internal RAM, where it's then executed. This code cannot be traced as long as its image is not part of the debug download file.

---

Note: Only the code included in the download files can be traced.

---

No considerable compression is possible for Data and Ownership trace due to the nature of the data, which needs to be captured. Depending on the traffic of the Data and Ownership messages, the trace buffer can run out relatively quickly. To get the most out of the trace memory, Infineon implemented complex qualification and trigger units, which allow to capture respectively filter only the information of interest. It's up to the user to set up an optimal filter in order to capture only the important information out of the vast amount of information, which PTU, DTU and OTU provide.

## 8.1.2 Multi-Core Debug Solution (MCDS)

Major four blocks of the MCDS are:

- **Processor Observation Block (POB)**

Each processor to be traced is paired with a dedicated Processor Observation Block (POB).

**Observation Block for TriCore**

- Direct access to the 4 IP/EA pretrigger comparators of OCDS L1
- 6 additional range comparators on the IP (this means 6 different ranges)
- 4 range comparators on the LMB write address
- 4 masked range comparators on the data written to the LMB
- Thread awareness: 2 comparators to restrict tracing to certain threads only
- Full non-cached data access visibility with trace at LMB and FPI bus
- Watch point traces based on all before mentioned comparators
- Debug status message based on execution mode of core
- Complete program trace
- Data trace for Write Back to LMB
- Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

**Observation Block for PCP**

- 4 range comparators on the IP
  - 4 range comparators on the PRAM write address
  - 2 masked range comparators on the data written to the PRAM
- Note: FPI accesses are handled by the SPB below*
- Watch point traces based on all before mentioned comparators
  - Debug status message based on execution mode of core
  - Complete Program Trace
  - Data trace for Write Back to PRAM
  - Ownership trace based on interrupt level of executed channel program
  - Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

- **Bus Observation Block (BOB)**

Each multi master on chip bus (LMB/FPI) has its own Bus Observation Block (BOB) to provide the required visibility.

- **Observation Block for System Peripheral Bus (SPB)**

- 4 range comparators on the FPI address
    - 4 masked range comparators on the FPI data bus
    - 4 masked range comparators on the FPI operation code/mastership

*Note: These features lock out the OCDS Level 1 features of the SBCU (breakpoints)*

- Watch point traces based on all before mentioned comparators
    - Ownership trace derived from bus arbiter
    - Data trace with or without address information for Read and/or Write access
    - Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

- **Observation Block for TriCore LMB Bus**

- Same features as SPB

- **Multi Core Cross-Connect (MCX)**

The main challenge with multi core system debugging is keeping a consistent view of the system, the components of which run independently and concurrently. This is achieved by time stamping the buffered trace messages. On the other hand, trace qualification is needed across core boundaries: It may be useful to trace a bus only if a certain processor core executes a specific subroutine. To keep the interfaces minimal, all such flow of information is routed through this central block, named Multi Core Cross-Connect for this reason.

- **Multi Core Cross-Connect**

- Gets 4 programmable pretrigger signals from each observation block
    - One dedicated global trace enable for each trace unit
    - 16 universal 16 bit counters, using programmable combinations of pretriggers as count and clear signals
    - Performance counter
    - Limit comparators based on these counters, generating further pretriggers
    - Global trace enables are sums of products (four multi-input ANDs ORed together)
    - Global break generation based on all available pretriggers - including the counters'
    - Bidirectional interface to the break switch on the product chip
    - Global time stamp messages, based on emulation or reference clock.

- **Debug Memory Controller (DMC)**

As explained elsewhere, there is little hope to transport all messages from target to host at the very instant of their creation. By integrating substantial amounts of FIFO-organized memory into the target's package it becomes possible to endure bursts of trace messages without loss of information, even if the physical interface is not extraordinary fast. To make the most of the memory an efficient Message Packer is provided to sort the trace messages from the different sources into the RAM. Additionally time markers can be inserted by the DMC which allow a reconstruction with accuracy down to the emulation clock cycle - if the memory required for the markers is sacrificed.

MCDS building blocks (generic modules) in a bottom up manner:

- Trigger Logic
  - Event Logic

When combining trigger results, two main cases are possible: Either the event is given by a number of triggers which have to match concurrently (e.g. address in range AND data equal to value) or something has to happen if at least one from a set of triggers (e.g. address lower than bottom OR higher than top of stack) matches.

As the second case very often requires ANDing triggers to formulate the elements of the OR set, it was

decided to relegate the OR function to the event's consumers, namely the Action Definitions. Number and kind of triggers connected to each event depend on the location of its implementation.

- Sequential Event Logic

In some applications an event is defined to have occurred when some triggers have happened in a certain or even arbitrary sequence. Key to the implementation for this problem is the concept of counting events and comparing the count values to given limits. The result of the comparison (cnt\_trig) is then treated as a trigger again.

- Performance Counter

The available counter structure can also be used for performance analysis.

- Action Logic

All activity inside MCDS is controlled by standardized registers called action definitions.

- Program Trace Unit (PTU)

The generic PTU is able to process the instruction pointer of an arbitrary processor core.

- Data Trace Unit (DTU)

The generic DTU is able to process transactions on an arbitrary bus system, consisting of address, data and control information

- Ownership trace unit (OTU)

Ownership is a NEXUS term; it translates to "Task ID" or "Process ID" for most practical purposes. The generic OTU is able to process the ownership information of an arbitrary processor core if implemented in hardware.

- Watch-point Trace Unit (WTU)

All comparators are implemented inside trace units (PTU, DTU, OTU). To keep the trace units simple, each of them is allowed to produce up to one trace message per clock cycle only. The watch-point trace messages are produced here to resolve the obvious problem of a watch-point and the "normal" trace being requested at the same time.

- Trace Qualifier Unit (TQU)

The TQU is the container for all Event and Action Logic. The calculation of the trace qualification signals needed by the trace units (DCU, PTU, DTU, OTU, WTU and TSU) is centralized in a unit of this kind.

- Time Stamp Unit (TSU)

This block delivers all time information, both for internal use (time tags) and messages (time stamps), to all the other parts of MCDS.

---

Note: A detail explanation of the Tricore trace is beyond the scope of this document. Contact your local Infineon representative if you need more details.

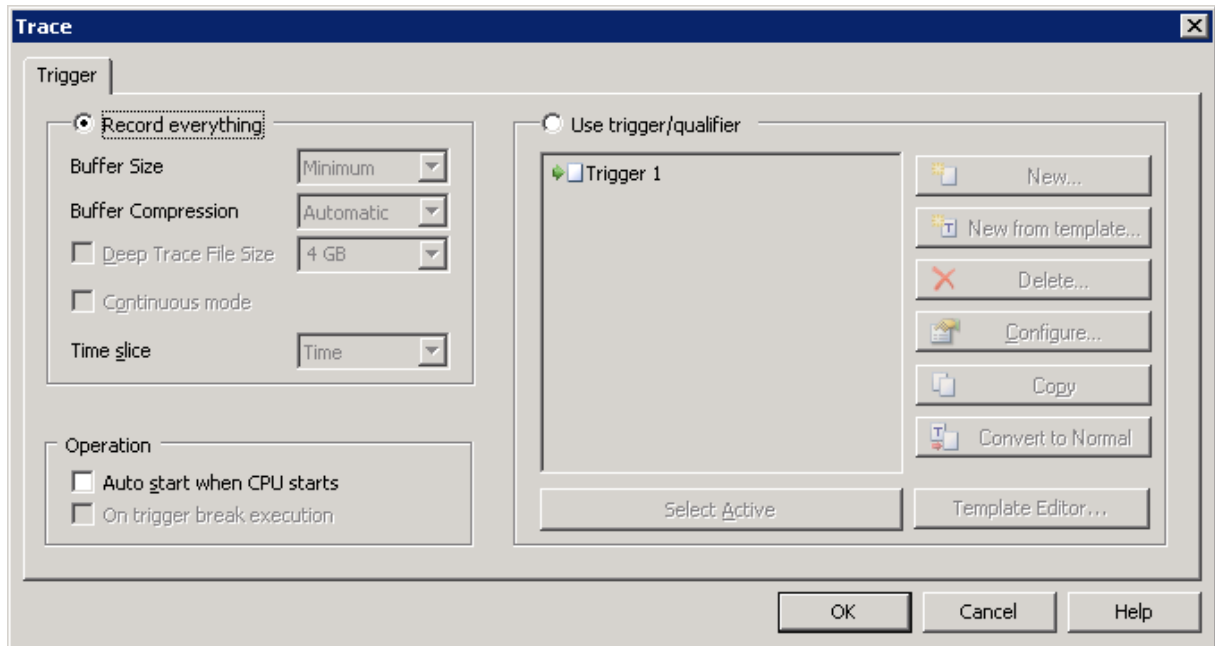
Due to the complexity of the TriCore trace, iSYSTEM tools offer a simple and intuitive Trace Wizard, which makes easy to set up most often trigger and qualifiers (trigger & qualifier on program counter and trigger & qualifier on data).

---

## 8.2 Trace Configuration

Trace window is open from the View menu. Pressing the 'Hardware Configuration' toolbar opens Trace dialog which allows configuring trace for 'Record everything' or 'Use trigger/qualifier' operation.

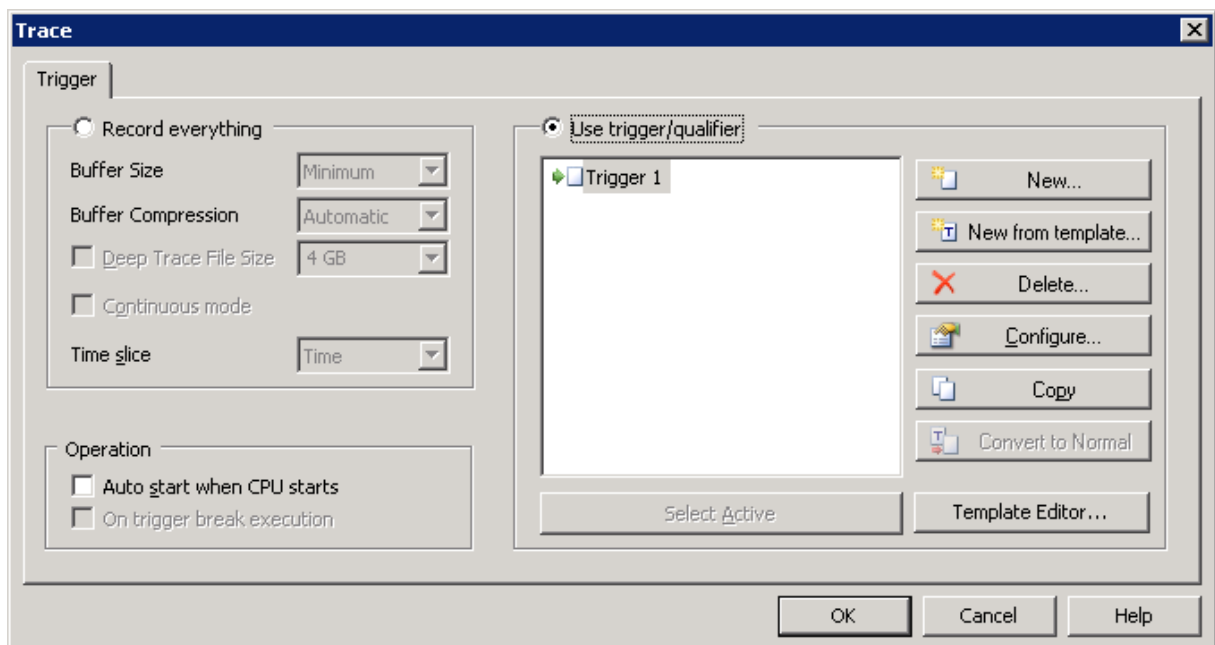
### 8.2.1 Record everything



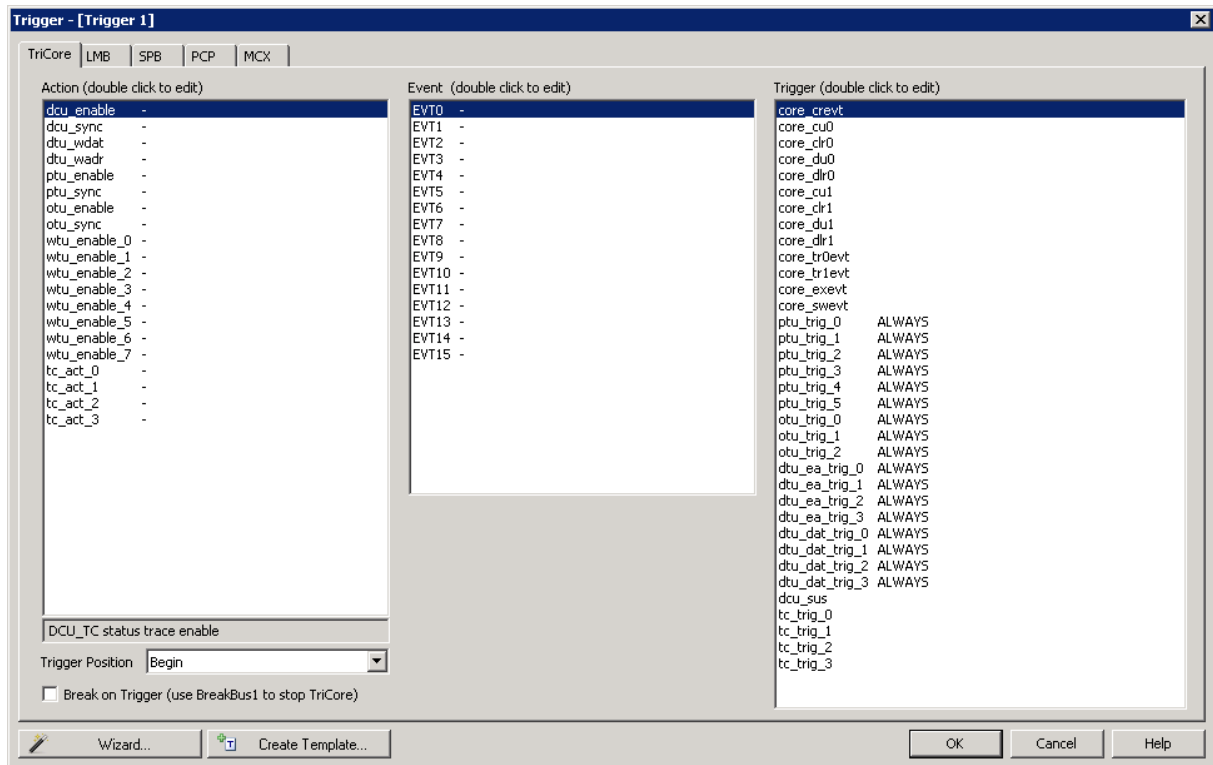
This configuration is used to record the contiguous program flow from the application start.

### 8.2.2 Use Trigger/Qualifier

This trace operation mode is used, when it's required to trace the application around a particular event or when only some parts of program or data have to be recorded. In practice it turns out to be the most important defining meaningful trace qualifiers.



Press the 'New...' button on the right to open the Trigger dialog.



*TriCore Trace Qualifier Unit configuration*

Tricore, LMB, SPB and PCP have each its own Trace Qualifier Unit (TQU) and hence its own pool of possible:

- triggers (right section in the dialog)

These are also referred as pretriggers. Pretrigger is a system state (status bit or comparator output).

- events (centre section)

An event is the combination of pretriggers at a certain point of time.

- and actions (left section)

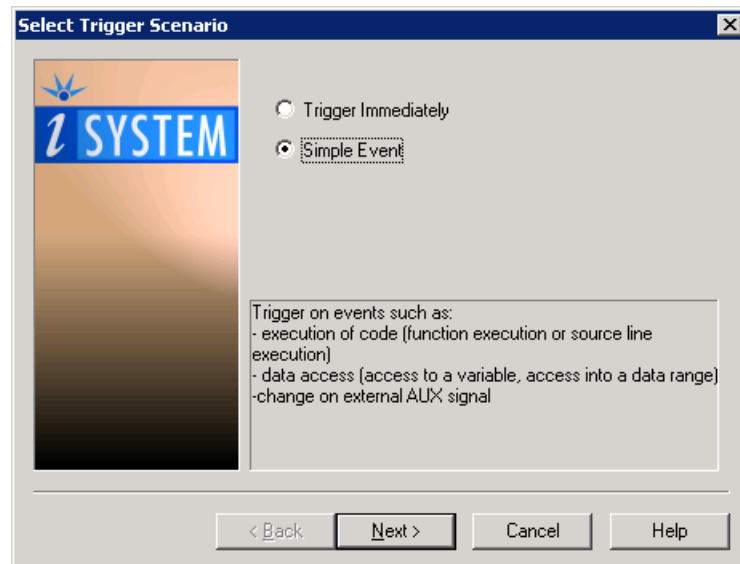
An action is something to be done when an event happens.

The MCX is not paired with a specific core or bus, but interfaces to the TQUs of all observation blocks (TriCore, LMB, SPB and PCP). A central trace qualification unit (TQU) contains the bulk of its functionality. Additionally the central time base (TSU), the usual watch-point message generation (WTU) and message sequencer (MSU) building blocks are implemented.

It is recommended to use iSYSTEM Trace Wizard when it's required to:

- set a simple trigger on an executed function
- set a simple trigger on a single data access
- trace only one program range
- trace only data access to a single address range

iSYSTEM Trace Wizard is invoked in the left bottom corner in the Trigger dialog and it configures all the necessary trace qualification units accordingly. When there is a demand for more complex trigger(s) and/or qualifier(s), the user should engage himself in the configuration of individual trace qualification unit.



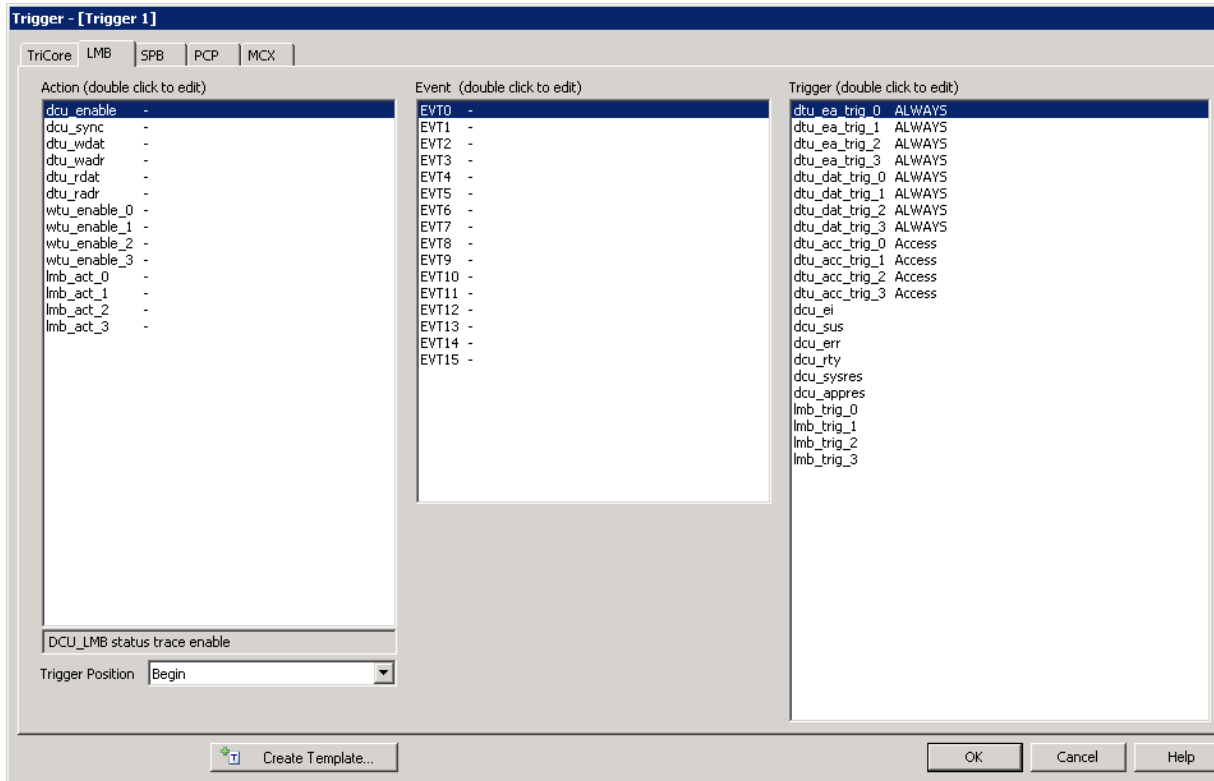
First page of the iSYSTEM *Trace Wizard*

## ***TriCore Trace Qualifier Unit (TQU\_TC)***

### **Feature Overview**

- Dedicated programmable trace enables for each Trace Unit of POB\_TC.
- Four dedicated actions for signalling to the TQU\_MCX.
- 13 core triggers from the OCDS-logic of the TriCore.
- Six range triggers from the PTU\_TC.
- Four range triggers on the write-back address seen by DTU\_TC.
- Four masked and signed data triggers from the DTU\_TC.
- Three triggers from the OTU\_TC.
- One trigger from the DCU\_TC.
- Four triggers from the TQU\_MCX.
- Five uncommitted trace enables from the TQU\_MCX.

## Local Memory Bus Trace Qualifier Unit (TQU\_LMB)

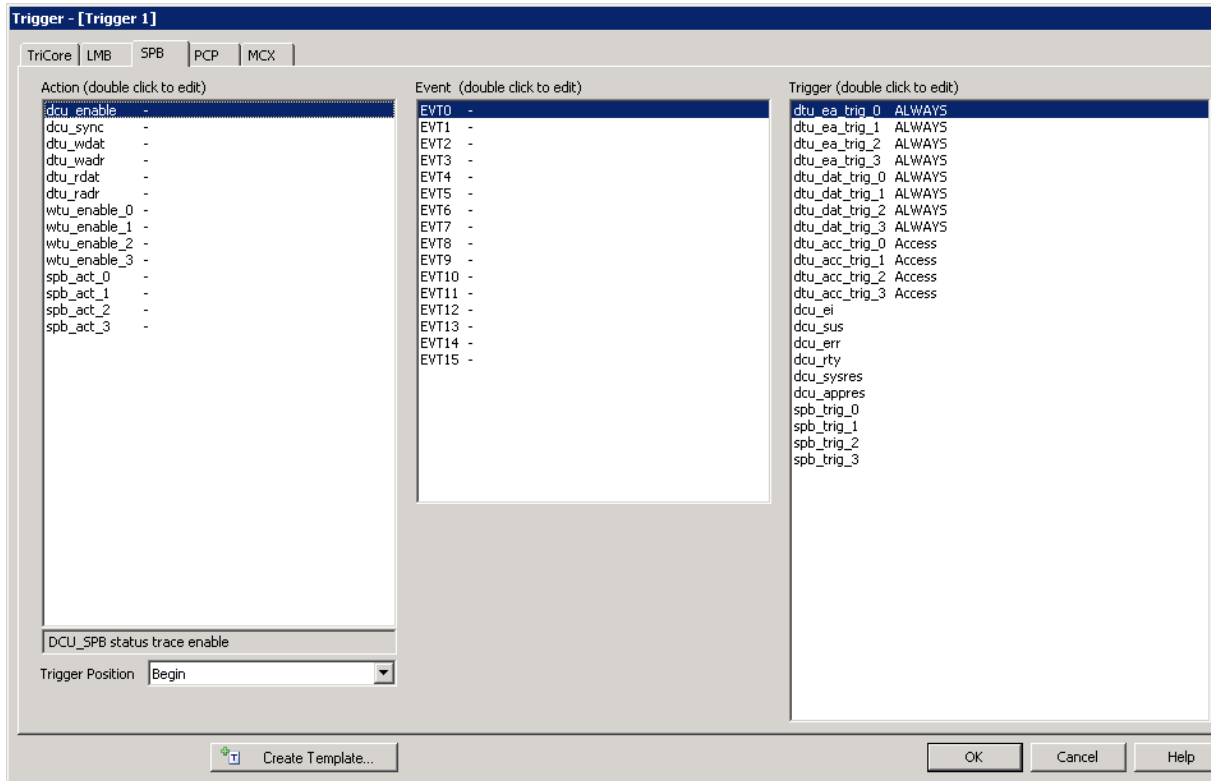


LMB Trace Qualifier Unit configuration

### Feature Overview

- Dedicated trace enables for each Trace Unit of BOB\_LMB.
- Four dedicated actions for signalling to the TQU\_MCX.
- Four range triggers on the transaction address seen by DTU\_LMB.
- Four masked and signed data triggers from the DTU\_LMB.
- Four range triggers on the transaction type and bus master seen by DTU\_LMB.
- Six triggers from the DCU\_LMB.
- Four triggers from the TQU\_MCX.
- Five uncommitted trace enables from the TQU\_MCX.

## System Peripheral Bus Trace Qualifier Unit (TQU\_SPB)



SPB Trace Qualifier Unit configuration

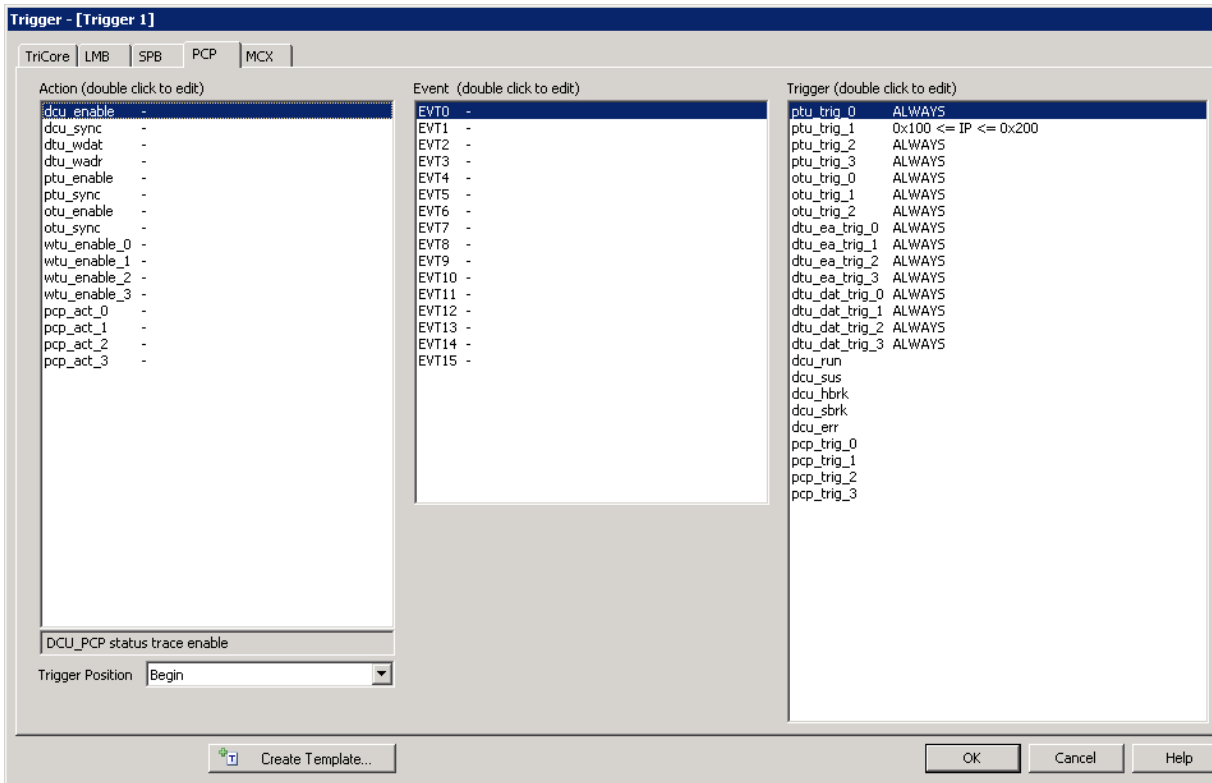
### Feature Overview

- Dedicated trace enables for each Trace Unit of BOB\_SPB.
- Four dedicated actions for signalling to the TQU\_MCX.
- Four range triggers on the transaction address seen by DTU\_SPB.
- Four masked and signed data triggers from the DTU\_SPB.
- Four range triggers on the transaction type and bus master seen by DTU\_SPB.
- Six triggers from the DCU\_SPB.
- Four triggers from the TQU\_MCX.
- Five uncommitted trace enables from the TQU\_MCX.

## PCP Trace Qualifier Unit (TQU\_PCP)

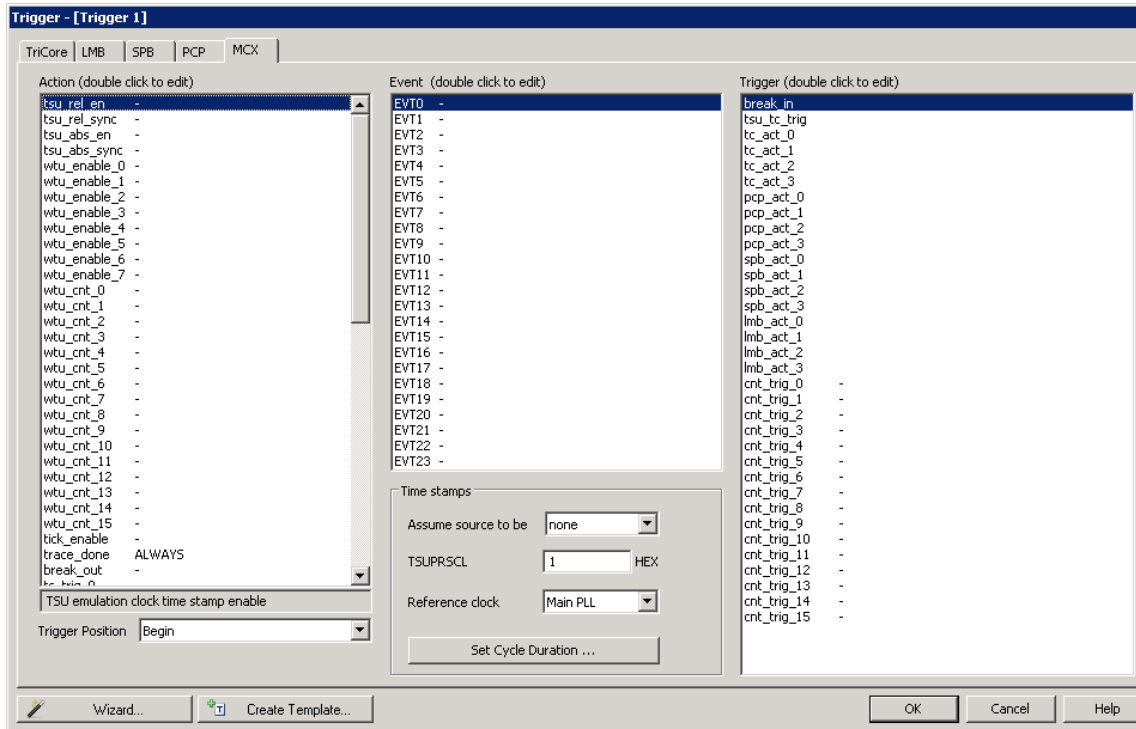
### Feature Overview

- Dedicated trace enables for each Trace Unit of POB\_PCP.
- Four dedicated actions for signalling to the TQU\_MCX.
- Four range triggers from the PTU\_PCP.
- Four range triggers from the PRAM write address seen by DTU\_PCP.
- Four masked and signed data triggers from the DTU\_PCP.
- Three triggers from the OTU\_PCP.
- Five triggers from the DCU\_PCP.
- Four triggers from the TQU\_MCX.
- Five uncommitted trace enables from the TQU\_MCX.



PCP Trace Qualifier Unit configuration

### Central Trace Qualifier Unit (TQU\_MCX)



MCX Trace Qualifier Unit configuration

Note that only triggers cnt\_trig0 to cnt\_trig15 are configurable on the right. Other triggers are actions already from the TriCore, LMB, SPB and PCP observation blocks. MCX Trace Qualifier Unit configuration dialog holds also the time stamp settings.

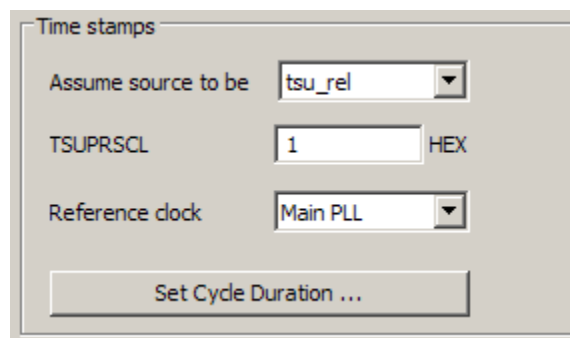
## Feature Overview

- Dedicated trace enables for each Trace Unit of MCX.
- 16 universal 16 bit counters, using programmable combinations of triggers as count and clear signals.
- Programmable limit comparator in each counter.
- Passing a limit is available as unique trigger for each counter.
- The counter values can be traced (see WTU\_MCX).
- Pre-scaled reference clock available as trigger.
- Four triggers from each observation block's TQU.
- Four triggers to each observation block's TQU.
- Five uncommitted trace enables to each observation block's TQU.
- Trigger from OCDS break switch.
- Trigger to OCDS break switch.
- 39 performance signals from TriCore, Flash, LMB, PCP, DMA, SPB.

## Time Stamp Configuration

By default, MCDS does not generate time stamp messages, which is somewhat distinct to Nexus trace implementations on other architectures. This means, in order to get the time stamp information as part of the trace record, the MCDS needs to be configured accordingly. Note that the trace time stamp information is not based directly on the time but is implemented on a CPU tick level. The trace user has to determine the CPU tick period in order to get the time information in the trace. This method of course relies on the assumption that the CPU clock doesn't change during the active trace session. Time cannot be measured properly when tracing program code changing the CPU clock.

As the time stamp mechanism is relatively complex too, the most convenient configuration will be described.



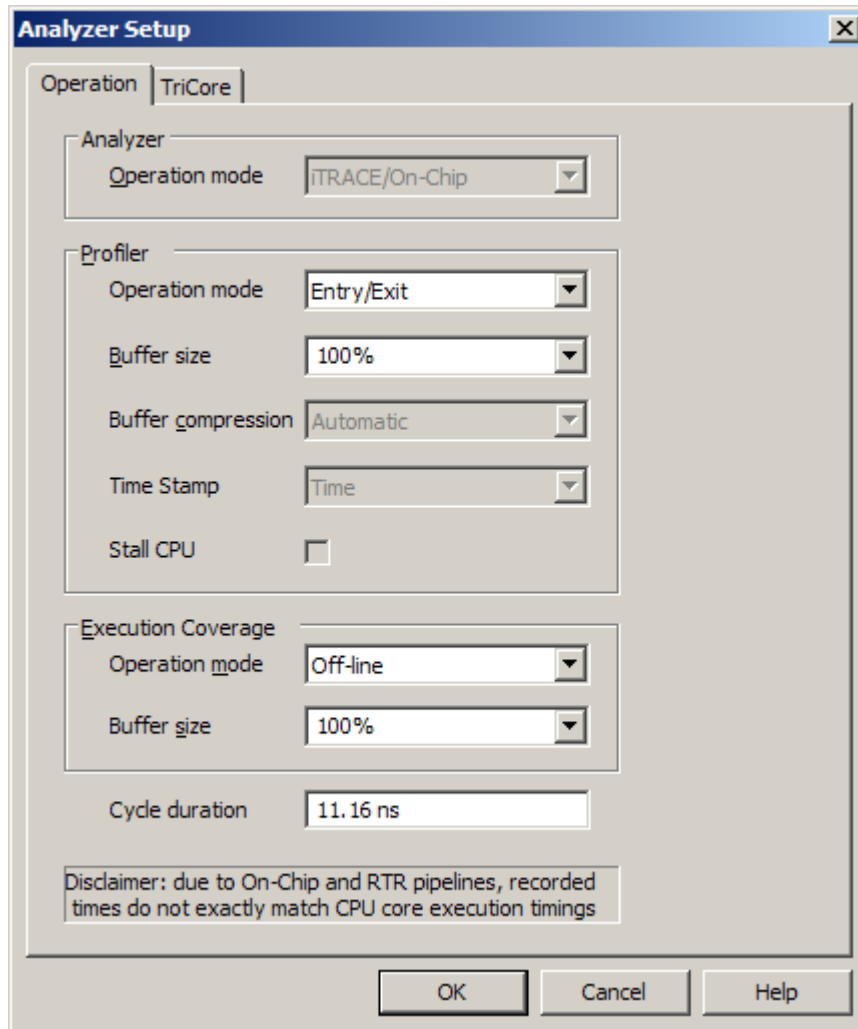
*Time Stamps area in the MCX Trace Qualifier Unit configuration dialog*

Set 'Assume source to be' to 'tsu\_rel', keep default TSUPRSCL set to 1 and Reference clock set to Main PLL. A value of 1 for TSUPRSCL activates the tsu\_tc\_trig output every second reference clock cycle. Increasing the value would yield higher time resolution.

Next open Measurement plug-in window from Plugins/Measurement menu. Run the application and press Refresh button in the Measurement plug-in window. Actual CPU clock will be displayed. Next calculate period for this clock. In this particular case, it's 11.16ns for 89.527MHz clock.

Item	Value	Description
CPU Clock	89.527 MHz	Measured CPU/System Clock

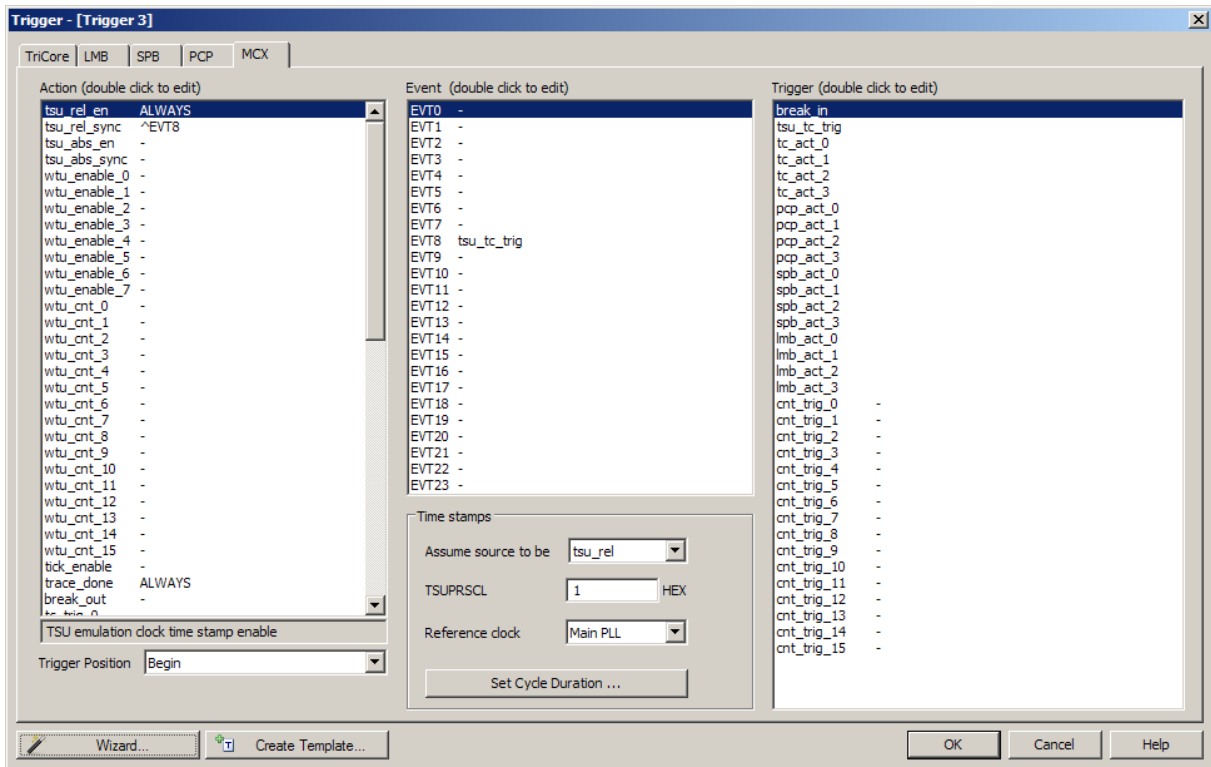
Next, press 'Set Cycle Duration ...' button in the Time Stamps area and set period at the bottom of the newly opened dialog.



*'Hardware/Analyzer Setup' dialog*

While this time stamp configuration is already operational for the trace ['Record everything'](#) operation mode, MCX need to be configured additionally to generate time stamps for the trace ['Trigger/Qualifier'](#) operation mode.

The easiest way is to use iSYSTEM trace wizard, which is started by pressing the 'Wizard...' button in the left bottom corner in the MCX Trace Qualifier Unit configuration dialog. Time stamps generation will be configured regardless of the specific trigger and/or qualifier setting. If trigger immediately and record program flow is configured via Wizard, MCX settings look like this:



Time stamps related settings are:

- event EVT8 is set to 'tsu\_tc\_trig' Trigger
- 'tsu\_rel\_en' Action is set on ALWAYS
- and 'tsu\_rel\_sync' is set to EVT8 Event edge

'trace\_done' Action set to ALWAYS yields recording everything from the trace start on.

Before the trace time stamp information is used, Cycle duration setting must be set in the 'Hardware/Analyzer Setup' dialog. The trace time stamp information is implemented on a CPU tick level only. Therefore, it's up to the user to find out the CPU cycle period of his target application and enter that value here. This type of time stamps doesn't provide accurate trace time information where the microcontroller frequency is not constant during the trace session.

## 9 Profiler

In general from the functional point of view, profiler can be used to profile functions and/or data.

- **Functions Profiler**

Functions profiler helps identifying performance bottlenecks. The user can find which functions are most time consuming or time critical and need to be optimized.

Its functionality is based on the trace, recording entries and exits from profiled functions. A profiler area can be any section of code with a single entry point and one or more exit points. Existing functions profiler concept does not support exiting two or more functions through the same exit point. Exit point can belong to one function only. In such cases, the application needs to be modified to comply with this rule or alternatively data profiler with code arming can be used in order to obtain functions profiler results.

The nature of the functions profiler requires quality high-level debug information containing addresses of function entry and exit points, or such areas must be setup manually. Profiler recordings are statistically processed and for each function the following information is calculated:

- total execution time
- minimum, maximum and average execution time
- number of executions/calls
- minimum, maximum and average period between calls

- **Data Profiler**

While functions profiler is based on analyzing code execution, data profiler performs time statistics on the profiled data objects, which are typically global variables. Typical use cases are task profiler and functions profiler based on code instrumentation.

When an operating system is used in the application, task profiler can be used to analyze task switching. When the task profiler is used in conjunction with the functions profiler, functions' execution can be analyzed for each task individually.

The development system features a so called off-line profiler. Off-line profiler is entirely based on the trace record. It first uses trace to record a complete program flow and then off-line, function entry and exit points are extracted by means of software, the statistic is run over the collected information and finally the results are displayed.

Refer to a separate document titled Profiler User's Guide for more details on profiler and its use.

---

Note: Execution Coverage is available on Tricore Emulation Device (ED) only. Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace. Single functionality can be used at the time only.

---

Be careful when including source lines in the offline profiler. A source line can often consists of a block of sequential instructions, which have all the same time stamp information due to the trace based on branch-trace concept. For instance, first instruction of the source line (entry) and last instruction (exit) will have the same time in such case and the profiler would display zero time spent in the source line although this is not the case in reality.

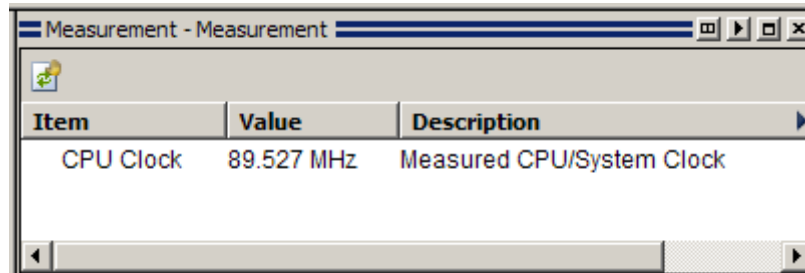
### ***Typical Use***

To use profiler, select working profiler buffer size in the 'Hardware/Analyzer Setup' dialog. Any value between 1% and 100% can be entered.

Before the profiler is used, Cycle duration setting must be set in the 'Hardware/Analyzer Setup' dialog. Note that Profiler relies on results captured by the trace, where the trace time stamp information is implemented on a CPU

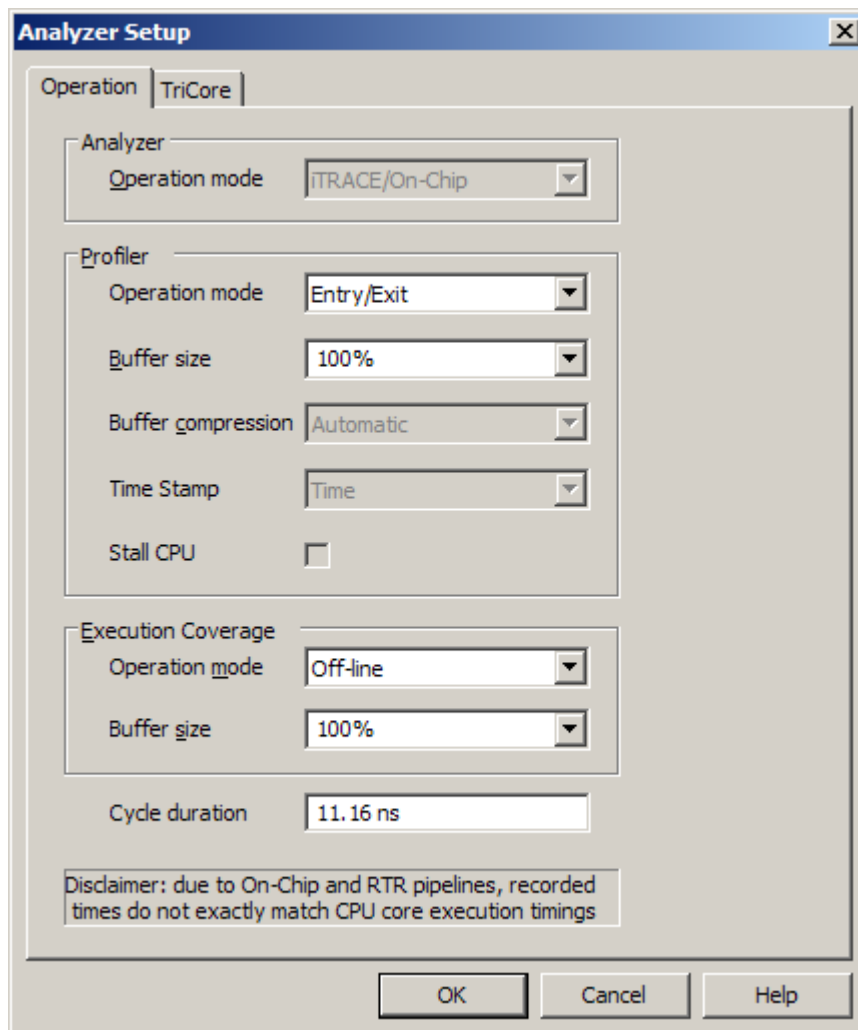
tick level only. Therefore, it's up to the user to find out the CPU cycle period of his target application and enter that value. This type of time stamps doesn't provide accurate trace time information where the microcontroller frequency is not constant during the trace session. Be careful not to profiler such applications.

Open Measurement plug-in window from Plugins/Measurement menu. Run the application and press Refresh button in the Measurement plug-in window. Actual CPU clock will be displayed. Next calculate period for this clock. In this particular case, it's 11,16ns for 89,525Hz clock.



Item	Value	Description
CPU Clock	89.527 MHz	Measured CPU/System Clock

Next, open 'Hardware/Analyzer Setup' dialog and set the period in the 'Cycle Duration' field at the bottom of the dialog. Hardware configuration is finished.



**Analyzer Setup**

Operation | TriCore

Analyzer  
Operation mode: iTRACE/On-Chip

Profiler  
Operation mode: Entry/Exit  
Buffer size: 100%  
Buffer compression: Automatic  
Time Stamp: Time  
Stall CPU:

Execution Coverage  
Operation mode: Off-line  
Buffer size: 100%

Cycle duration: 11.16 ns

Disclaimer: due to On-Chip and RTR pipelines, recorded times do not exactly match CPU core execution timings

OK Cancel Help

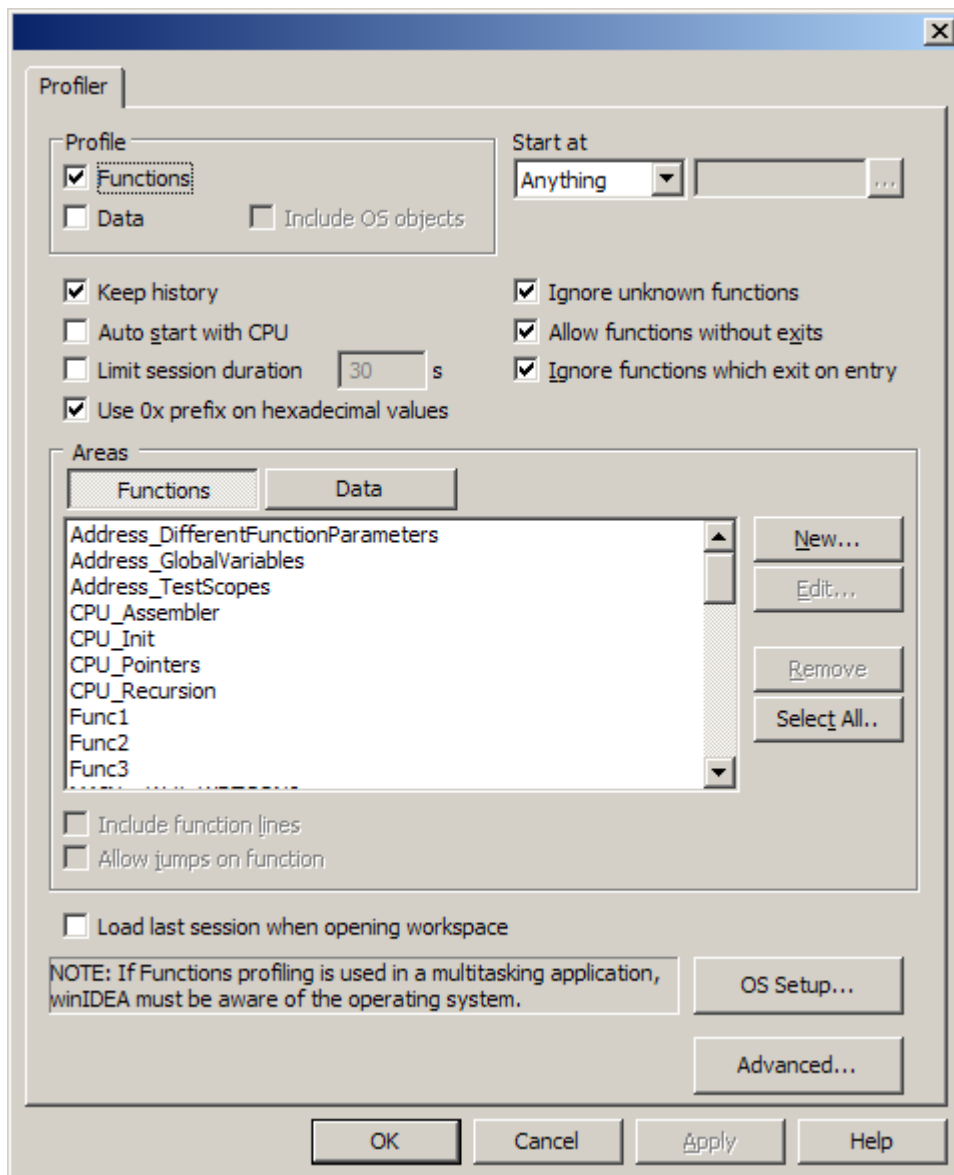
Next, select 'Profiler' window from the View menu and configure profiler settings (see next figure). Select 'Functions' option in the 'Profile' field when profiling functions.

In order to profile data information 'Data' should be checked in the Profile field. For instance, Data Profiler can be used as a Task Profiler, if the operating system writes a unique task ID to the trace. When using functions profiler in application with operating system, the task switches ABSOLUTELY & UNCONDITIONALLY MUST be profiled too!

Make sure that 'Keep history' option is checked if History view is going to be used during the results analysis. If the option is unchecked, all recorded profiler data are discarded after the statistic information is calculated and history view shows no results.

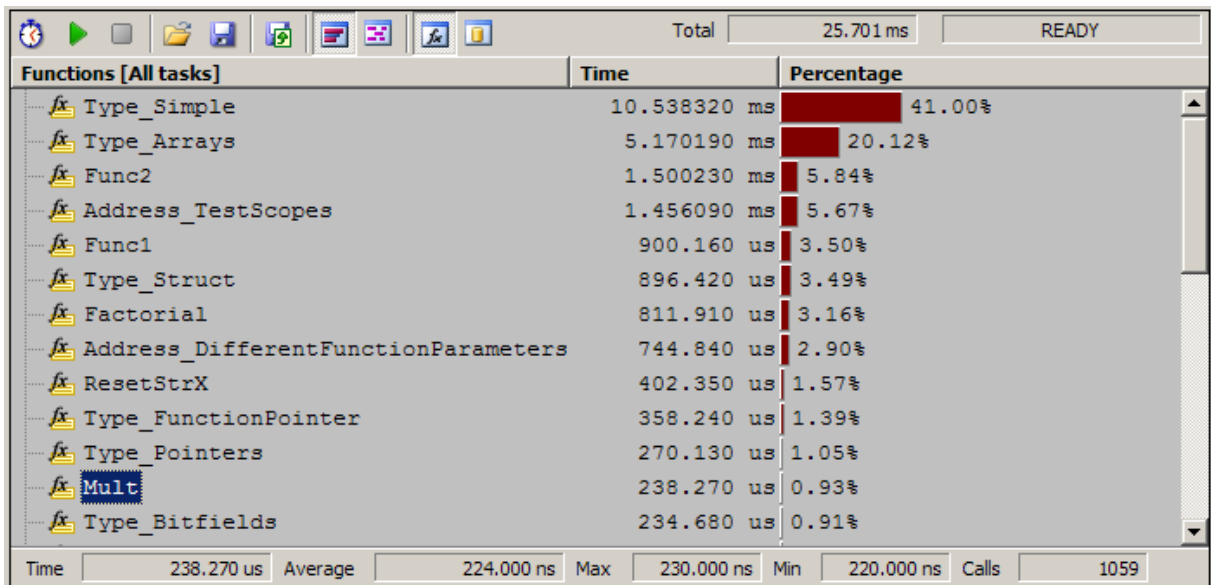
Finally, profiled functions are selected by pressing 'New...' button. It's recommended that 'All Functions' option is selected for the beginning.

The debugger extracts all the necessary information from the debug info, which is included in the download file and configure hardware accordingly.

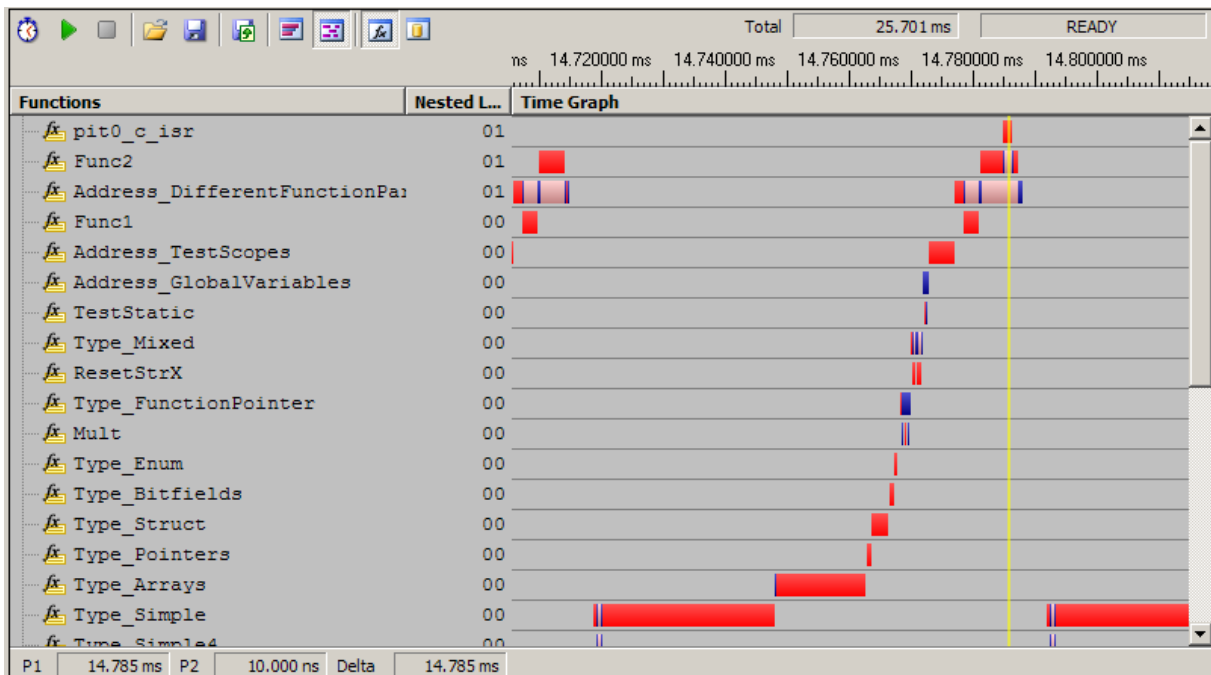


*Profiler configuration settings*

Profiler is configured. Reset the application, start Profiler and run the application. The Profiler will stop recording on a user demand or after the profiler buffer becomes full. While the buffer is uploaded, the recorded information is analyzed and profiler results displayed.



Statistics view



History view

## 10 Execution Coverage

Execution coverage records all addresses being executed, which allows the user to detect the code or memory areas not executed. It can be used to detect the so called “dead code”, the code that was never executed. Such code represents undesired overhead when assigning code memory resources.

The development system features a so called off-line execution coverage.

Off-line execution coverage is entirely based on the trace record. It first uses trace to record the executed code (capture time is limited by the 256kBytes on-chip trace buffer) and then offline executed instructions and source lines are extracted by means of software and finally the results displayed.

Off-line execution coverage tests the code for statement coverage metrics.

Refer to a separate Execution Coverage User’s Guide for more details on execution coverage configuration and use.

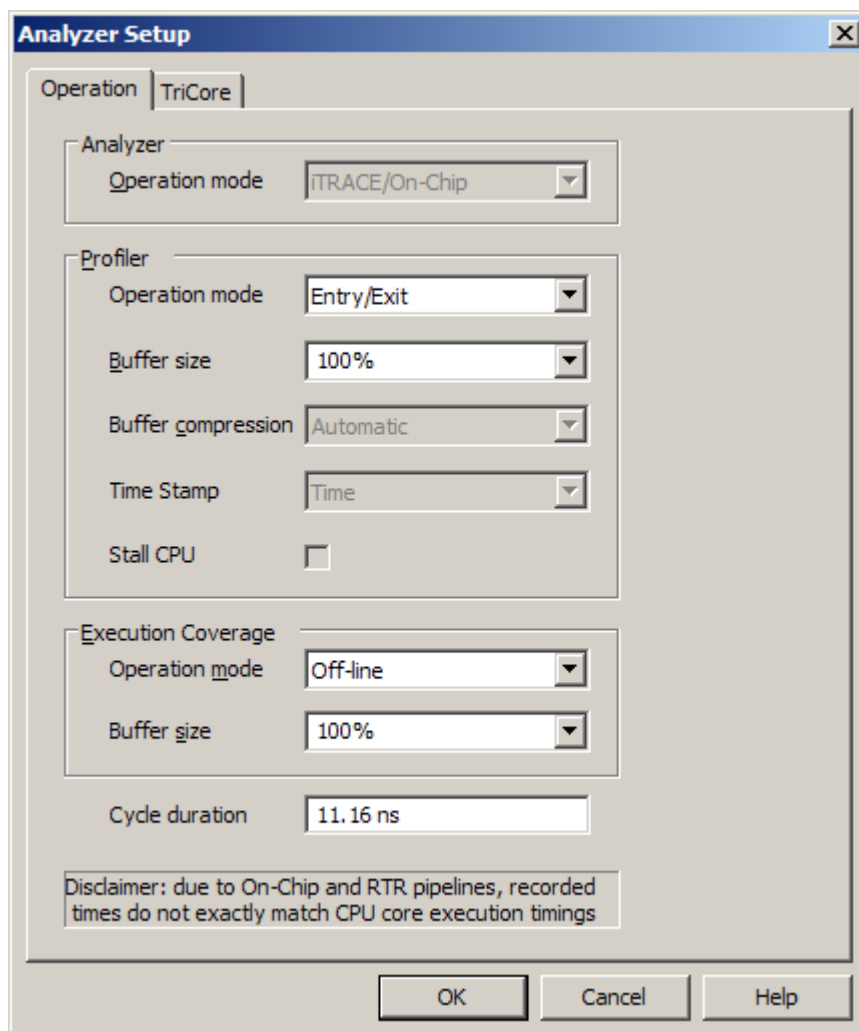
---

Note: Execution Coverage is available on Tricore Emulation Device (ED) only. Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace. Single functionality can be used at the time only.

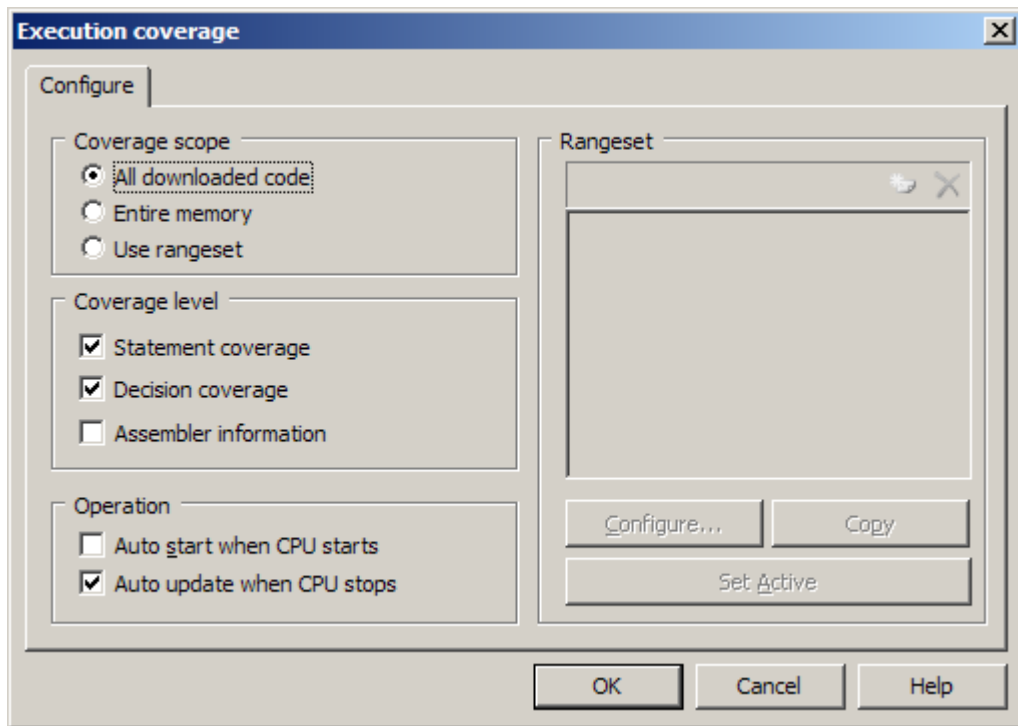
---

### *Typical Use*

No settings are required in the ‘Hardware/Analyzer Setup’ dialog.



Next, select 'Execution Coverage' window from the View menu and configure Execution Coverage settings. Normally, 'All Downloaded Code' option has to be checked only. The debugger extracts all the necessary information like addresses belonging to each C/C++ function from the debug info, which is included in the download file and configure hardware accordingly.



Execution Coverage is configured. Reset the application, start Execution Coverage and then run the application. The debugger uploads the results when the trace buffer becomes full or when requested by the user.

StatPane	Lines Bar	Lines	Sizes Bar	Sizes	Conditionals	Counts
Symbols		38/396 (10%)		0xAC/0xB5C (6...)	(7t,6f,4b)/72 (15%)	
..\common\		36/261 (14%)		0xA8/0x5B0 (12...)	(7t,6f,4b)/43 (24%)	
..\common\coverage		25/35 (71%)		0x74/0x90 (81%)	(6t,6f,3b)/20 (45%)	
..\common\debug.c		7/182 (4%)		0x1C/0x438 (3%)	(0t,0f,1b)/15 (7%)	
..\common\main.c		4/17 (24%)		0x18/0x60 (25%)	(1t,0f,0b)/5 (10%)	
main		4/17 (24%)		0x18/0x60 (25%)	(1t,0f,0b)/5 (10%)	
..\common\profilerC		0/27 (0%)		0x0/0x88 (0%)	(0t,0f,0b)/3 (0%)	
X:\TASKING\TriCore-VX_						
cstart.c		0/81 (0%)		0x0/0x2AC (0%)	(0t,0f,0b)/11 (0%)	
_START		0/3 (0%)		0x0/0xA (0%)		
__asm(")t_init_sp		0/1 (0%)		0x0/0x4 (0%)		
__asm(")wordt%0"		0/1 (0%)		0x0/0x4 (0%)		
}		0/1 (0%)		0x0/0x2 (0%)		

*Execution Coverage results*

## 11 Getting Started

- 1) Connect the system
- 2) Make sure that the target debug connector pinout matches with the one requested by a debug tool. If it doesn't, make some adaptation to comply with the standard connector otherwise the target or the debug tool may be damaged.
- 3) Power up the emulator and then power up the target.
- 4) Execute debug reset
- 5) The CPU should stop on reset location.
- 6) Open memory window at internal CPU RAM location(s) and check whether you are able to modify its content.
- 7) If you passed all 6 steps successfully, the debugger is operational. Now you may add the download file and load the code to the RAM
- 8) To program the flash or download the code to the RAM, which is not accessible after reset, make sure you use the initialization sequence to enable the access. First, the debugger executes reset, then the initialization sequence and finally the download or flash programming is carried out.

## 12 Troubleshooting

- Try 'Slow' JTAG Scan speed if the debugger cannot connect to the CPU.
- Make sure that the power supply is applied to the target JTAG connector when 'Vref' is selected for Debug I/O levels in the Hardware/Emulator Options/Hardware tab, otherwise emulation fails or may behave unpredictably.
- When flash programming fails, double check that proper target device is selected in winIDEA.
- When performing any kind of checksum, remove all software breakpoints since they may impact the checksum result.

---

Disclaimer: iSYSTEM assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information herein.

© iSYSTEM. All rights reserved.