
Technical Notes

Freescal MPC5xxx & ST SPC56 Nexus Class 3+

Contents

Contents.....	1
1 Introduction	2
2 Nexus Trace Configuration	4
2.1 e200 Nexus Trace Configuration	4
2.1.1 Record everything.....	4
2.1.2 Trace Trigger	4
2.2 eDMA Nexus Trace Configuration	9
2.3 FlexRay Nexus Trace Configuration.....	10
2.4 eTPU Nexus Trace Configuration.....	11
3 e200 Nexus Trace Examples	14
3.1 Nexus	14
3.2 Nexus RTR.....	18
3.3 Troubleshooting	31
4 Profiler.....	32
4.1 Typical Use	32
4.2 Troubleshooting	37
5 Execution Coverage.....	37
5.1 Typical Use	38

1 Introduction

This document covers Freescale MPC5xxx and ST SPC56 microcontrollers featuring Nexus Class 3+ interface. Refer to microcontroller reference manual to identify Nexus Class level of a particular microcontroller or contact iSYSTEM technical support for this information.

According to the Nexus standard, these devices contain multiple Nexus clients that communicate over a single IEEE-ISTO 5001-2003 Nexus class 3(+) combined JTAG IEEE 1149.1 auxiliary out interface. Combined, all of the Nexus clients are referred to as the Nexus development interface (NDI). Class 3+ Nexus allows for program, data and ownership trace of the microcontroller execution without access to the external data and address buses.

Communication to the NDI is handled via the auxiliary port and the JTAG port.

The Nexus trace is based on messages and has its limitations comparing to the in-circuit emulator where the complete CPU address, data and control bus is available to the emulator in order to implement exact and advanced trace features.

Due to the limited debug features of the Nexus trace, iSYSTEM has introduced a proprietary Nexus Real-Time Reconstruction (Nexus RTR), which restores the original e200 core execution bus, which is otherwise embedded deeply in the CPU silicon, in the development system. With this technology, advanced trace functions, extended profiler and infinite real-time execution coverage become available.

Nexus trace supports:

- program, data and ownership trace for the e200 core
- program, data and ownership trace for the eTPU (on CPUs where it's available)
- tracing data accesses for the eDMA module (on CPUs where it's available)
- tracing data accesses for the FlexRay (on CPUs where it's available)

Program Trace

Using a branch-trace mechanism, the program trace feature collects the information to trace program execution. For example, the branch-trace mechanism takes into account how many sequential instructions the processor has executed since the last taken branch or exception. Then the debugging tool can interpolate the instruction trace for sequential instructions from a local image of program memory contents. In this way, the debugging tool can reconstruct the full program flow. Self modifying code cannot be traced due to this concept.

Nexus trace implements internal FIFO buffer, which keeps the data in the pipe when the Nexus port bandwidth requirements are greater than capabilities. FIFO is heavily used when the application sequentially accesses data, which yields heavy trace port traffic through a narrow Nexus port.

Note that only transmitted addresses (messages) contain relatively (time of message, not of execution) valid time stamp information. All CPU cycles being reconstructed by the debugger relying on code image and inserted between the recorded addresses, do not contain valid time information. Any interpolation with the recorded addresses containing valid time stamp would be misleading for the user. Thereby, more frames displayed in the trace window contain the same time stamp value.

Data Trace

Data trace is used to track real-time data accesses to device specific internal peripheral and memory locations by specifying a start and stop address with read or write access (the MPC55xx supports two such qualifier areas).

Transmitted information about the memory access cannot be compressed fundamentally since each memory access is distinctive and not predictable. Errors in the trace window appear when the CPU executes too many

data accesses in a short period. These yield numerous Nexus messages, which cannot be sent out through the narrow Nexus port to the external development system on time and an internal data message FIFO overflow is reported in the trace window. Consequentially, it's highly recommended to configure on-chip message control (qualifier) to restrict data trace recording only to data areas of interest to minimize possible overflows.

Program and data trace messages are not ordered in time. Since the data trace has precedence over the program trace, a number of data messages is recorded before the actual instruction (block of instructions between two branches, or sync) is recorded that caused the data accesses. No reordering is done by the debugger since it would be highly speculative and cannot be guaranteed to be valid, unless the messages would contain a time-stamp. Unfortunately, this is not realized in the MPC5500 Nexus implementation.

Ownership Trace

Ownership trace is based on ownership trace messaging (OTM). OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. In practice, an operating system writes to the process ID register (PID0), which yields an ownership trace message for every write. Then it's up to the data profiler to record these messages and display the task activities (task profiler).

Nexus Class 3+ Trace Features (iC5000 & iTRACE GT):

- External trace buffer
- Program, Data and OTM Trace for e200 core
- Program, Data and OTM Trace for eTPU1 and eTPU2
- Data trace for eDMA
- Data trace for FlexRay (MPC5567)
- Advanced external trigger and qualifier
- Time Stamps
- AUX inputs
- Profiler
- Execution Coverage

2 Nexus Trace Configuration

Default winIDEA instance allows debugging and tracing the primary e200 core. In case of a second core, another winIDEA instance is open from the Debug/Core in order to debug and trace the 2nd e200 core.

Analyzer window is open from the View menu.

Refer to a separate document titled Analyzer User's Manual for more details on general handling & configuring the analyzer window and its use. Only MPC5xxx Nexus L3+ specifics are explained in this document.

A detailed and exhaustive explanation on how the Nexus trace works and the meaning and purpose of all Nexus options, which are found in the Nexus configuration dialogs within winIDEA, can be found in the individual Core Reference Manual. Identify the core inside of your microcontroller and then refer to the belonging Core Reference Manual, which can be typically found at and downloaded from the semiconductor vendor web site. Some information may also be found in the Microcontroller Reference Manual of a specific microcontroller.

2.1 e200 Nexus Trace Configuration

2.1.1 Record everything

This configuration is used to record the contiguous program flow either from the application start or up to the moment when the application stops.

The trace can start recording on the initial program start from the reset or after resuming the program from the breakpoint location. The trace records and displays program flow from the start until the trace buffer fulfills.

This is the default mode when a new analyzer .trd file is created.

Buffer Size

This setting defines a maximum analyzer file size. The analyzer stops collecting Nexus trace information when this limit is exceeded.

Note that this setting is not correlated to the physical trace buffer of the HW debug tool by any means. The actual analyzer physical buffer size is limited by the debug tool. For instance, if the debug tool is capable of recording 512KB of the Nexus trace information only, limiting analyzer file size to 1MB poses no restriction at all. However, if the user finds just a small portion of the analyzer record (e.g. 16kB) being of interest and requires a swift analyzer window handling, it makes sense limiting the analyzer files size to 16kB. In this case, just a belonging portion of the complete analyzer physical buffer is required and used.

2.1.2 Trace Trigger

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.

Create a new Trace Trigger in the Analyzer window.

Trigger

Note: There is a single Trigger dialog which covers all different devices, which also feature different set of on-chip debug resources. Based on the selected CPU, only supported settings in the dialog are enabled and others are disabled.

The same on-chip debug resources are shared among e200 hardware execution breakpoints, e200 access breakpoints and e200 on-chip trace trigger. Consequentially, debug resources used by one debug functionality

are not available for the other two debug functionalities. In practice this would mean that no trace trigger can be set for instance on instruction address, when four execution breakpoints are set already.

Trigger

Trace can trigger immediately after the trace is started or can trigger on one or more watchpoints (debug events), which occur while the target application is running. Trigger watchpoints can be IAC1-IAC8, DAC1-DAC2, CNT1-CNT2 and are described next.

Instruction

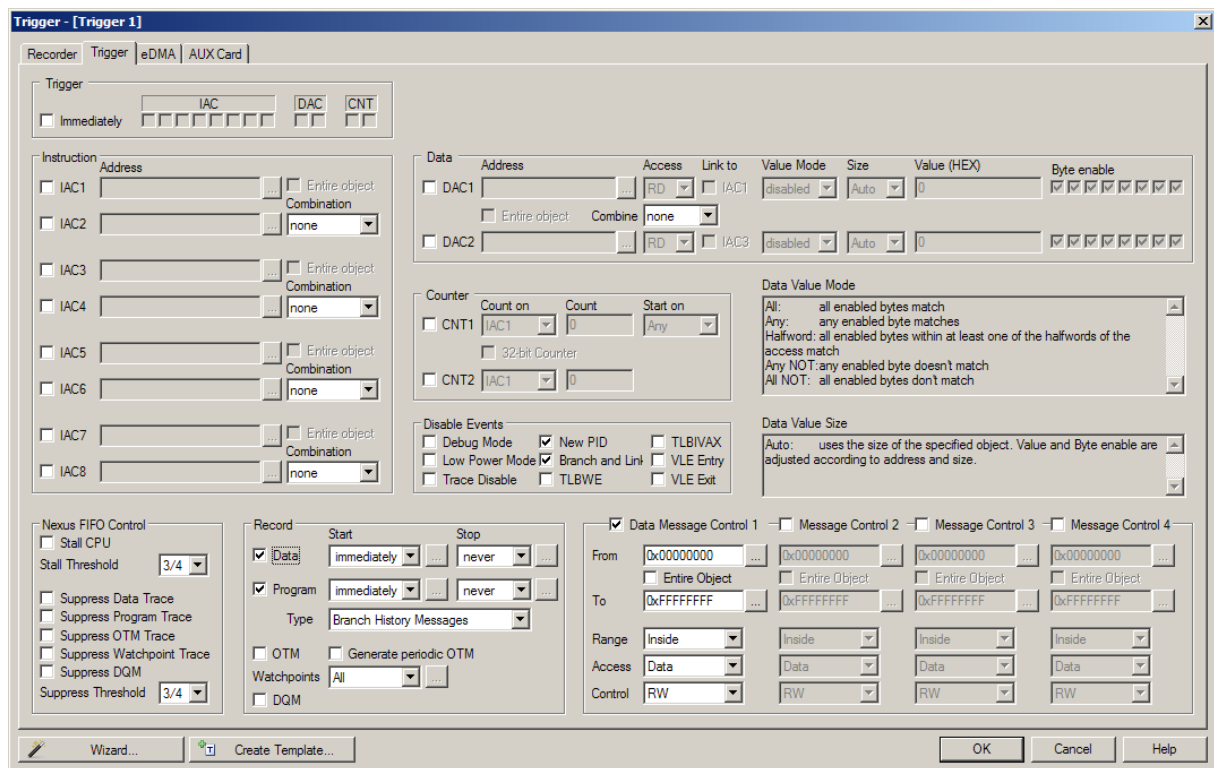
Up to 8 watchpoints (IAC1-IAC8) can be configured to trigger on executed instruction address (program counter match). Eight address matches, four address in/out range matches or four address matches where address can be additionally masked, can be configured.

Devices based on e200z4 and e200z7 cores feature eight instruction watchpoints (IAC1-IAC8) while majority of devices feature four instruction watchpoints (IAC1-IAC4).

Data

Two watchpoints (DAC1, DAC2) can be configured to trigger on accessed data address. Besides the access type, two address matches, one data address in/out range match or one address match where address can be additionally masked, can be configured.

When 'Link to' option is checked, configured data access is further conditional on instruction defined by IAC1/IAC3 watchpoint. In practice, the user can restrict trigger on data access caused by an explicit instruction.



Trigger tab in the Trace Trigger Configuration dialog

Counter

Debug module features two 16-bit counters CNT1 and CNT2 which can be configured to operate independently or can be concatenated into a single 32-bit counter. Each counter can be configured to count down when one or more count-enabled events occur (IAC1-IAC4, DAC1-DAC2). When the count value reaches zero, a debug

event is generated. First counter (CNT1) can have additionally a start condition which can be IAC1, IAC3, DAC1 or CNT2 event.

Disable Events

Messages generated by certain events (Debug Mode, Low Power Mode, Trace Disable, New PID, Branch and Link, TLBWE, TLBIVAX, VLE Entry, VLE Exit) can be disabled in order to minimize total amount of generated Nexus messages.

Note: These options are available on e200z4 and e200z7 cores and per default they are unchecked.

The on-chip Nexus module has a limited amount of nexus messages, which can be broadcasted over the Nexus port in certain time frame without a loss of Nexus information. An overrun Nexus message is reported when the maximum bandwidth is exceeded.

Refer to the belonging Core Reference Manual for more details on these options.

Program Trace

Program trace is enabled by default. Most often setting for the Start is 'immediately' and for the 'End' is 'never'. However, user can select any of the previously described watchpoints to act as Start or End condition on match.

There are two types of messages, which can be used for the Nexus program trace protocol. 'Individual Branch Messages' yield more information about program execution than the 'Branch History Messages' setting. Major advantage of the 'Individual Branch Messages' setting is more accurate time information but it requires more Nexus port bandwidth, which means that the Nexus trace is more subject to the overflows, which are depicted in the trace window when they occur. In case of overflows, program reconstruction in the trace window resumes with next valid Nexus trace message.

Data Trace

Enable data trace when it's required to record data accesses besides the program.

Default setting for the Start is 'immediately' and for the 'End' is 'never'. However, mind that the trace may start displaying errors due to the data trace enabled. Depending on the application, Nexus trace can output a huge amount of access addresses and access data which in worst case yield internal data message FIFO overflows due to a limited Nexus port bandwidth. Then also program path reconstruction fails.

To stay away from the possible overflows, the user should use watchpoints, which can be used as data trace Start or End condition on match and/or Message Control, which allows defining two data windows. This minimizes the number of data messages to be sent through the Nexus port.

Message Control 1-4 define four independent data address ranges. Valid condition can be either address in range or outside of range. Nexus trace can record Instruction access data and Data access data (default). Tracing can be further restricted to Read/Write, Read or Write accesses only.

Note: Devices based on e200z4 and e200z7 cores feature four Message Controls (1-4) while majority of devices feature two Message Controls (1-2)

OTM Trace

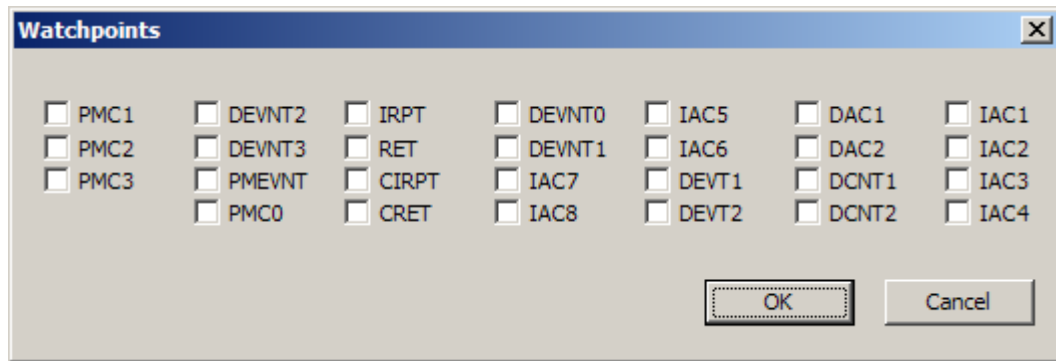
Enable OTM check box, when 8-bit writes to the process ID register should be recorded.

Generate periodic OTM

Periodically, once every 256 messages, the most recent state of the PID0 register is messaged out when this option is checked.

Watchpoints

Per default all watchpoints are generated and recorded. If there are custom requirements, the user can configure either 'no watchpoints are generated' or selects specific watchpoints to be generated and recorded.



DQM Trace

Data acquisition trace provides a convenient and flexible mechanism for the debugger to observe the architectural state of the core through software instrumentation.

For DQM, a dedicated 32-bit SPR has been allocated (DDAM). It is expected that the general case is to instrument the software and use `mtspr` operations to generate Data Acquisition Messages.

Nexus FIFO Control

The Overrun Control register (OVCR) controls the Nexus behavior as the message queue fills. The Nexus block may be programmed to:

- Allow the queue to overflow, drain the contents, queue an overrun error message and resume tracing.
- Stall the processor when the queue utilization reaches the selected threshold.
- Suppress selected message types when the queue utilization reaches the selected threshold

By default, Nexus block is configured for the first type of operation (stalling & suppression disabled), where an overrun condition is possible but the program execution time is not affected by any means.

Stall CPU

In this mode, processor instruction issue is stalled when the queue utilization reaches the selected threshold. The processor is stalled long enough to drop one threshold level below the level which triggered the stall. $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ Stall Threshold can be selected.

Message Suppression

In this mode, the message queue will disable selected messages types when the queue initialization reaches the selected threshold. This allows lower bandwidth tracing to continue (e.g. program trace) and possibly avoid an overrun condition. Once triggered, message suppression will remain in effect until queue utilization drops to the threshold below the level selected to trigger suppression.

Data Trace, Program Trace, OTM Trace, Watchpoint Trace and DQM Trace messages can be individually suppressed by checking the individual option. $\frac{1}{4}$, $\frac{1}{2}$, or $\frac{3}{4}$ Suppress Threshold can be selected.

2.2 eDMA Nexus Trace Configuration

eDMA trace is configured in the eDMA pane in the Trace dialog.

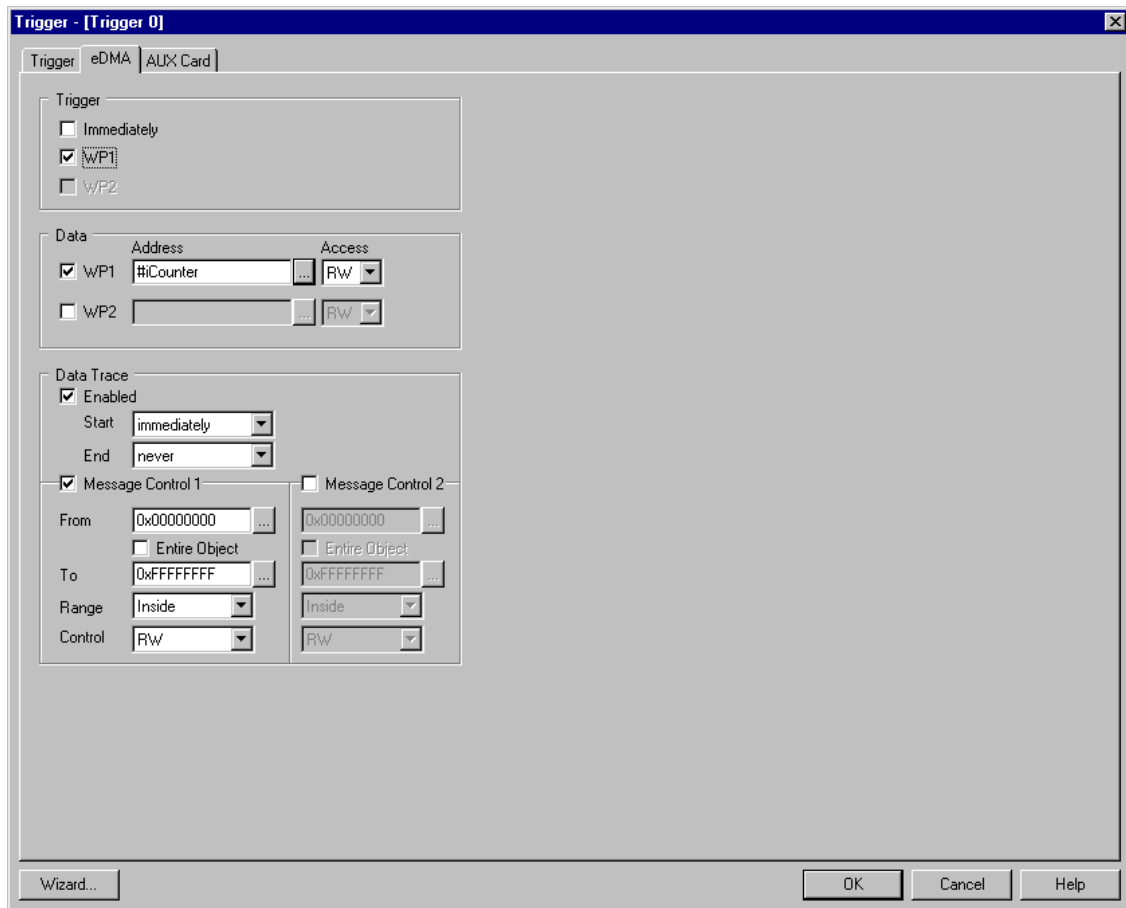
Note: eDMA trace is not available on all microcontrollers. eDMA trace is available only when the 'Nexus' trace type is selected in the 'Hardware/Analyzer Setup' dialog. eDMA trace cannot be used in conjunction with iSYSTEM Nexus RTR technology.

Trigger

Trigger options specify how the eDMA trace is triggered. The trace can be triggered immediately or by one of the two eDMA watchpoints.

Data

Two eDMA watchpoints (WP1, WP2) can be defined and can be used either for the trigger or data trace start/end. Address and access type can be defined for each watchpoint.



eDMA Trace Configuration dialog

Data trace

First, eDMA trace must be globally enabled by checking the 'Enabled' option.

Next, the user must set the data trace start and end condition. Trace can be started immediately or by means of one of the watchpoints. The same goes for the trace stopping – the trace can be set to never stop or when one of the watchpoints is reached.

Two eDMA data address ranges (Message Control 1 & 2) can be defined in order to optimize the amount of the eDMA trace messages on the Nexus port. The user should use them in order to avoid or to keep at minimum possible internal message FIFO overflows due to limited Nexus port bandwidth. eDMA data accesses that fall within the Message Control 1 and/or Message Control 2 window, are output only.

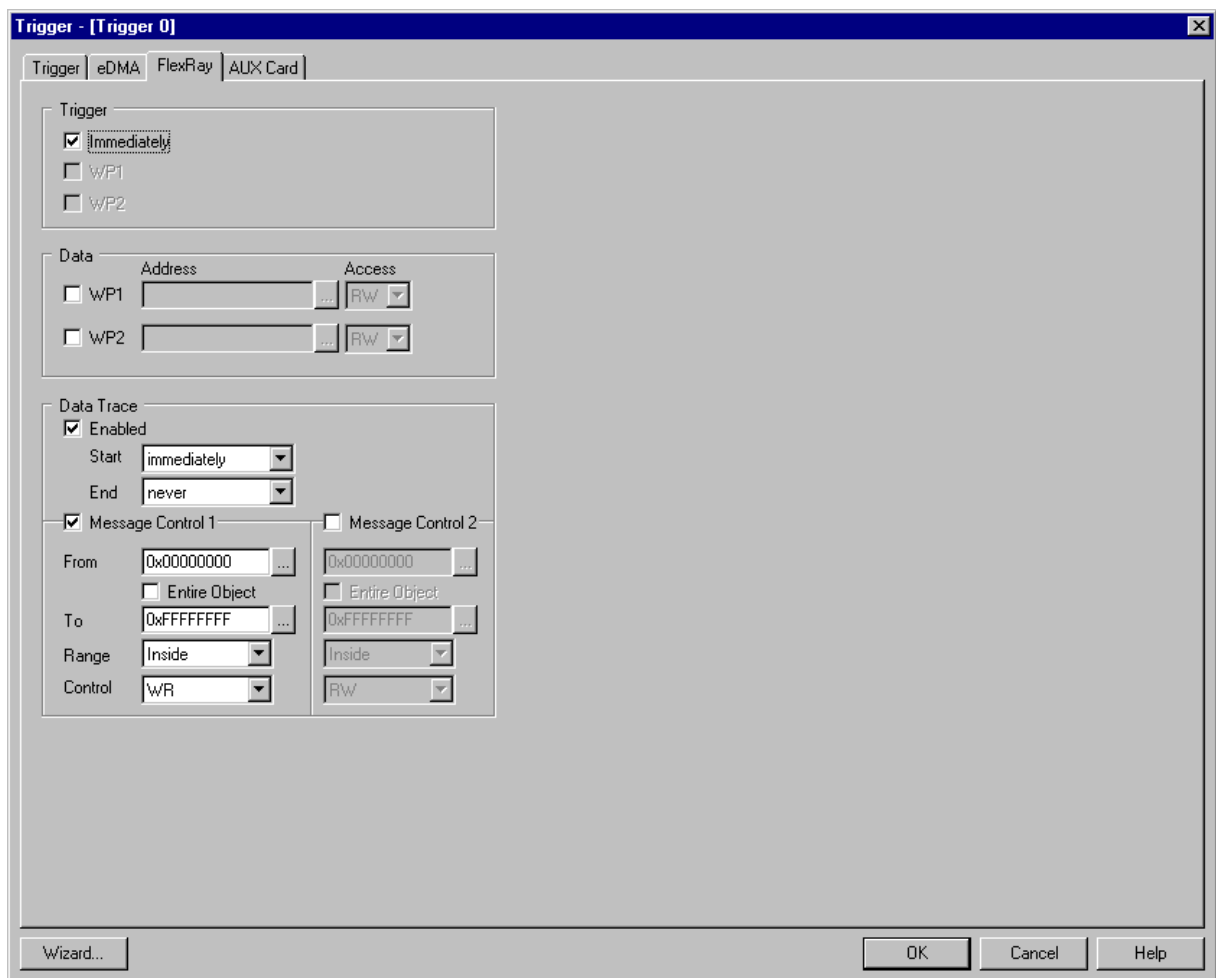
2.3 FlexRay Nexus Trace Configuration

FlexRay Trace is configured in the FlexRay pane in the Trace dialog. FlexRay trace is supported only on MPC5500 devices featuring the FlexRay module (e.g. MPC5567).

Note: FlexRay trace is available only when the 'Nexus' trace type is selected in the 'Hardware/Analyzer Setup' dialog. FlexRay trace cannot be used in conjunction with iSYSTEM Nexus RTR technology.

Trigger

Trigger options specify how the FlexRay trace is triggered. The trace can be triggered immediately or by one of the two FlexRay watchpoints.



FlexRay Trace Configuration dialog

Data

Two FlexRay watchpoints (WP1, WP2) can be defined and can be used either for the trigger or data trace start/end condition. Address and access type can be defined for each watchpoint.

Data trace

First, FlexRay trace must be globally enabled by checking the 'Enabled' option.

Next, the user must set the data trace Start and End condition. Trace can be started immediately or by means of one of the watchpoints. The same goes for the trace stopping – the trace can be set to never stop or when one of the watchpoints is reached.

Two FlexRay data address ranges (Message Control 1 & 2) can be defined in order to optimize the amount of the FlexRay trace messages on the Nexus port. The user should use them in order to avoid or to keep at minimum possible internal message FIFO overflows due to limited Nexus port bandwidth. Only FlexRay data accesses that fall within the Message Control 1 and/or Message Control 2 window are output only.

2.4 eTPU Nexus Trace Configuration

eTPU Trace is based on messages and features program trace, data trace and ownership trace.

Program trace is based on branch trace messaging, which displays program flow discontinuities (start, jump, return, etc.) allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced only.

Data trace allows tracing reads and writes to selected shared parameter RAM (SPRAM) address ranges.

Ownership trace provides visibility of which channel is being serviced. An ownership trace message is transmitted to indicate when a new channel service request is scheduled, allowing the development tools to trace task flow. A special OTM is sent when the engine enters in idle, meaning that all requests were serviced and no new requests are yet scheduled.

The eTPU1 and eTPU2 module are debugged each in a separate winIDEA session. Refer to Freescale MPC5500 Family On-Chip Emulation technical notes document for more details on the eTPU debugging. eTPU Nexus Trace is open from View/Trace, assuming that the eTPU winIDEA debug session is set up and active.

Both eTPU engines have their own Nexus register sets that allows trace to be set up independently for each of them. The only exception to this is the data trace address range registers that are shared. Refer to the Nexus Dual eTPU Development Interface chapter in the eTPU Reference Manual for more details on the eTPU trigger and qualifier settings and the eTPU trace in general.

Note: The same on-chip debug resources are shared among eTPU hardware execution breakpoints, eTPU access breakpoints and eTPU on-chip trace trigger. Consequentially, debug resources used by one debug functionality are not available for the other two debug functionalities. In practice this would mean that no trace trigger can be set for instance on instruction address, when hardware execution breakpoints are set already, etc.

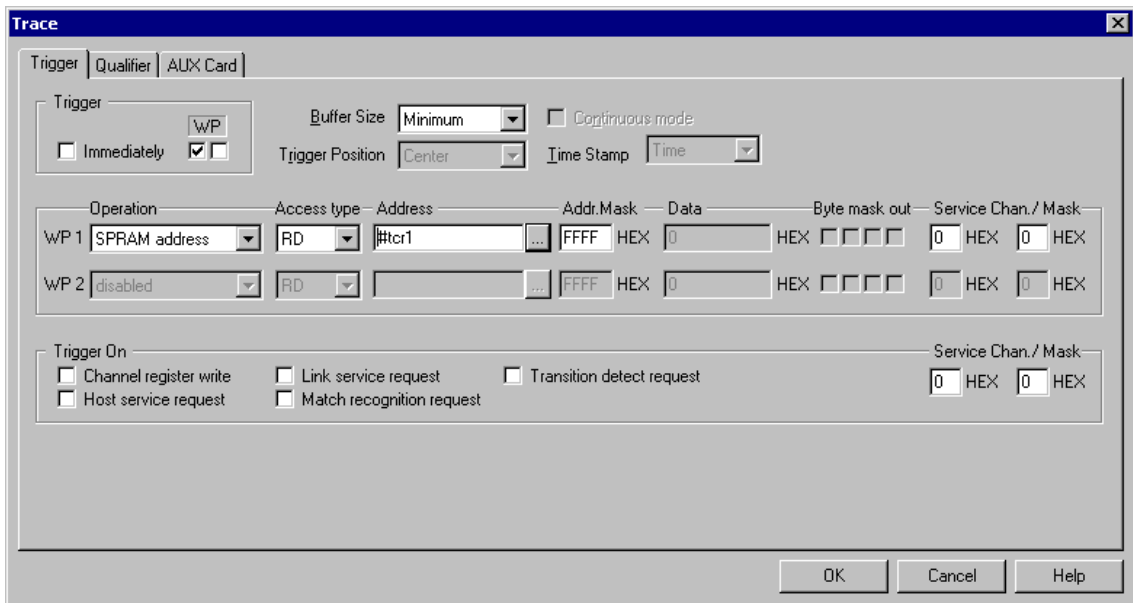
Trigger

Trigger options specify how the eTPU trace is triggered. The trace can trigger immediately or on:

- eTPU watchpoint 1 (WP1) occurrence
- eTPU watchpoint 2 (WP2) occurrence
- channel register write occurrence
- host service request occurrence
- on link register occurrence
- on match recognition request

- on transition detect request

All seven trigger conditions can be limited to a specific serviced channel. If serviced channel information is to be ignored, 0 should be written for the service channel mask.

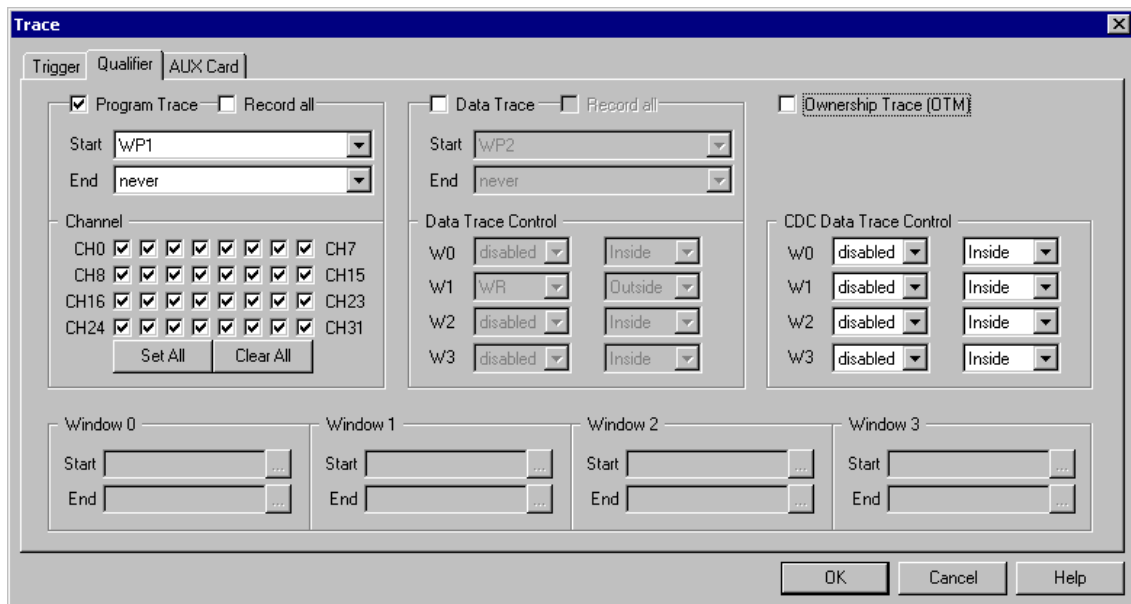


eTPU Trigger Configuration

Watchpoints

Two watchpoints (WP1, WP2) can be defined and used for the trigger and the qualifier. The 'Operation' combo box selects whether address and/or data matching is done and if matching is done on data fetches or instruction fetches. Access Type can be read, write or read/write. Address can be masked. Address Mask 0xFFFF considers all the bits in the address while 0x0 masks all address bits. When Data is used for the watchpoint, individual byte within 32-bit value can be masked (Byte mask out). Lastly, 5-bit service channel must be specified for the watchpoint, which can be also masked (when a mask bit is 0 that bit is not compared for masking).

Qualifier



eTPU Qualifier Configuration

Qualifier should be used with sense in order to prevent or at least minimize the eTPU trace overflows on the Nexus port. Depending on the application and the eTPU trace settings, the on-chip eTPU trace module can generate more messages than it is capable to send out externally over the Nexus port without loss. In general, the user should strive after the settings, which generates minimum traffic on the Nexus port while still displaying the relevant information on eTPU activities.

Program Trace

By default, program trace is configured to trace all the program activity. However, the program trace information can be limited by defining start and end condition or by focusing program trace on one or more active eTPU channels only. See an explanation for the available Start and End events in the Trigger section.

Data Trace

Four data trace windows with programmable address ranges and access attributes are provided. Data trace windowing reduces the requirement on the Nexus port bandwidth by constraining the number of trace locations. The four trace window address ranges are shared among the dual engines and the eTPU coherent dual-parameter controller (CDC). Besides the four data trace windows, number of trace locations can be additionally limited through the data trace Start and End condition.

Ownership Trace

Check the option when ownership trace messages need to be traced.

3 e200 Nexus Trace Examples

3.1 Nexus

Following examples show some of the capabilities of the Nexus trace port.

Select 'Nexus' in the 'Hardware/Analyzer Setup' dialog.

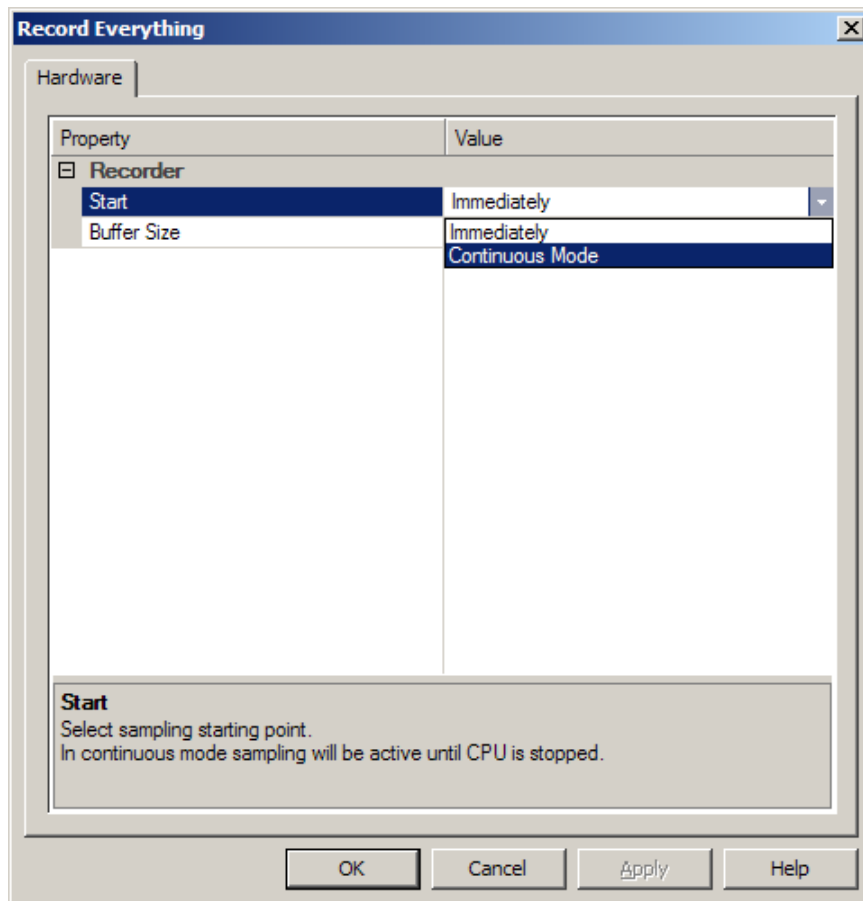
Default trace configuration is used to record the continuous program flow either from the program start on or up to the moment, when the program stops.

The trace can start recording on the initial program start from the reset or after resuming the program from the breakpoint location. The trace records and displays program flow from the start until the trace buffer fulfills.

As an alternative, the trace can stop recording on a program stop. 'Continuous mode' allows roll over of the trace buffer, which results in the trace recording up to the moment when the application stops. In practice, the trace displays the program flow just before the program stops, for instance, due to a breakpoint hit or due to a stop debug command issued by the user.

Example: The application behavior needs to be analyzed without any intrusion on the CPU execution. The trace should display program execution just before the CPU is stopped by debug stop command.

- Use 'Record everything' operation type in the 'Analyzer' window and make sure that 'Continuous mode' is configured to ensure that the trace buffer rolls over while recording the running program. The trace will stop as soon as the CPU is stopped. Note that this 'Record everything' operation type always apply for the e200 trace.



- Define reasonable buffer size depending on the required depth of the trace record. Have in mind that a smaller buffer uploads faster. You can start with e.g. 128kB.

With these settings, the trace records program execution as long as it's running. As soon as the program is stopped, the trace stops recording and displays the results.

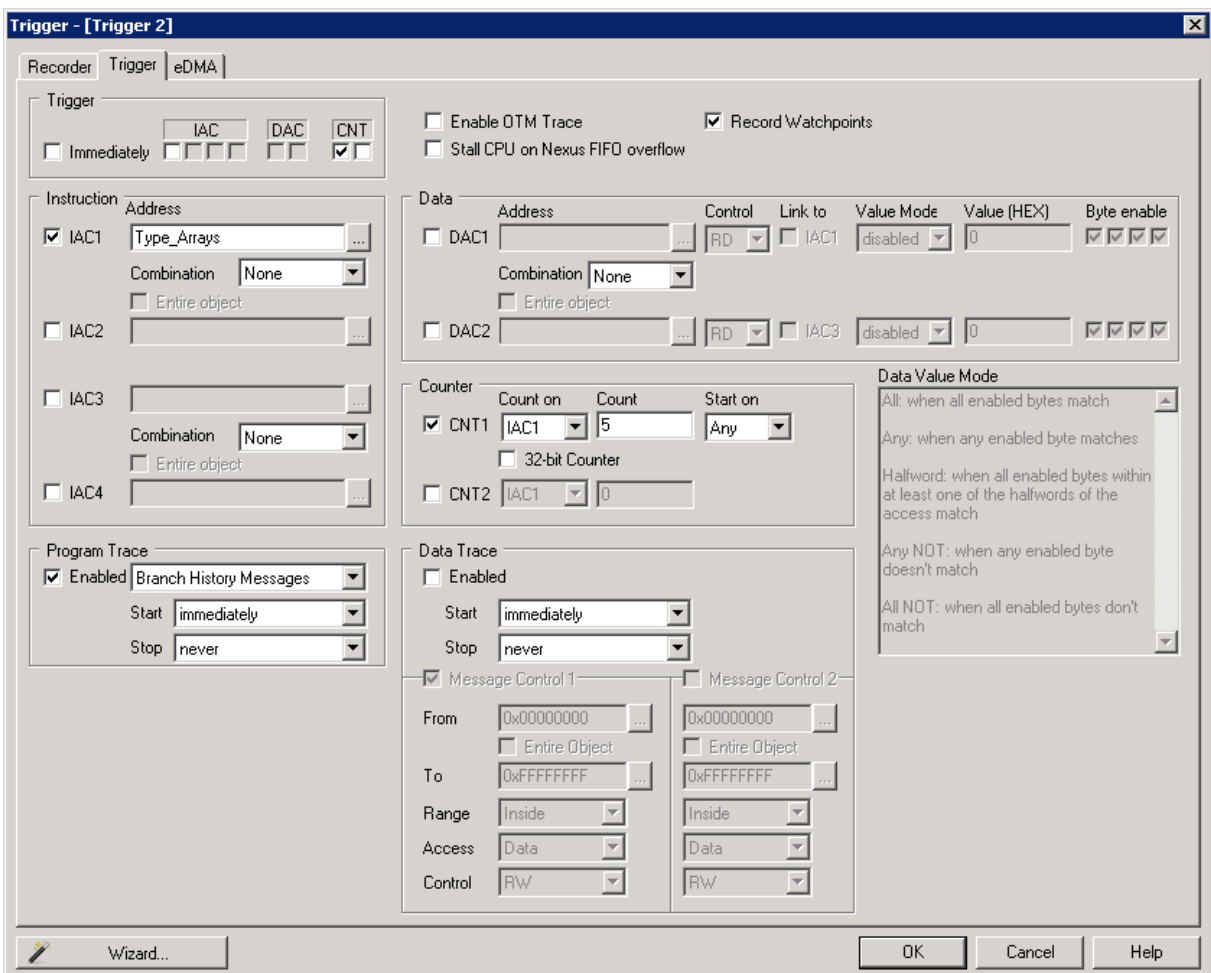
Following examples describe configuring trace to trigger on a specific function being executed or to record specific variable data accesses.

The 'On trigger break execution' option in the 'Trace Configuration' dialog should be checked when it's required to stop the program on a trigger event.

Example: Trace starts recording after `Type_Struct` function is called for the fifth time.

- Create new Trace Trigger in the Analyzer window.
- Enable Instruction IAC1 watchpoint and specify `Type_Struct` for the Address.
- Enable CNT1 counter, select IAC1 watchpoint for 'Count on' event, set 5 for 'Counter' and keep 'Start on' set to 'Any'.
- Enable Data Trace and keep default configured Message Control 1
- Set Trigger on CNT1 debug event.

The trace is configured. Following picture depicts current trace settings. Initialize the complete system, start the trace and run the program.



Let's inspect the results. Trigger point can be found around frame 0 and marked as 'Watchpoint' in the Content bus.

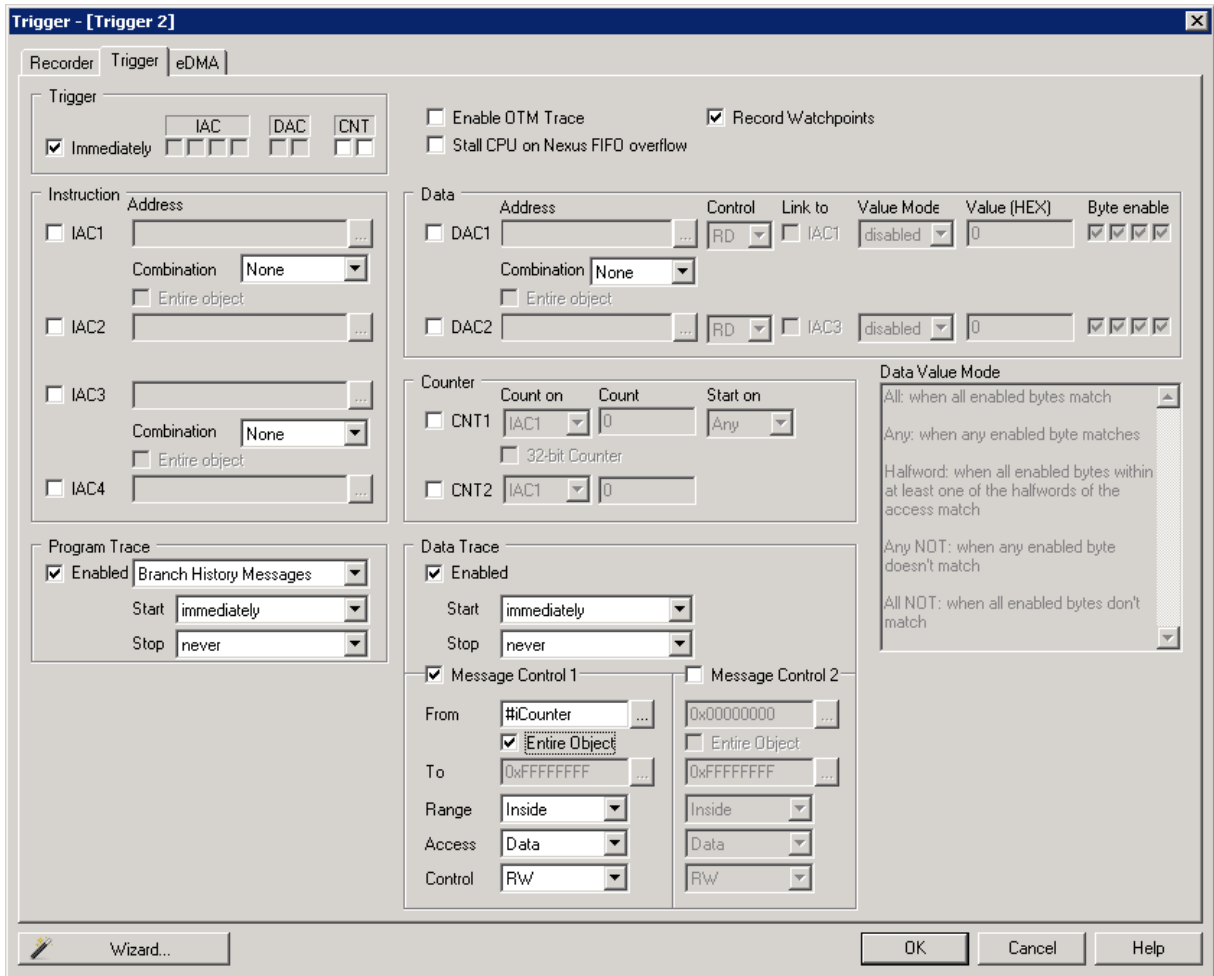
Don't forget that recorded instruction and belonging data access are not recorded in expected time order. This applies for all data access records. Belonging instruction is recorded and visible much later (e.g. 25 frames) after the data access was recorded. . This happens due to the Nexus concept, which immediately broadcasts data messages, while instruction messages are usually sent in blocks.

	Number	Address	Data	Content	Time
	-7	000006F0	7C0803A6	7C0803A6 mtlr Instruction r0	-5.325 us
	-6	000006F0	7C0803A6	7C0803A6 mtlr Instruction r0	-5.325 us
	-5	000006F0	7C0803A6	7C0803A6 mtlr Instruction r0	-5.325 us
	-4	000006F0	7C0803A6	7C0803A6 mtlr Instruction r0	-5.325 us
	-3	00000354	480003AD	Type_Struct(); 480003AD bl Instruction Type_Struct (0700)	-4.988 us
	-2	00000000	41000000	Watchpoint	-4.650 us
	-1	40005FA0	40005FEO	Write	-4.325 us
T	0	40005FDC	40005FEO	Write	0 ns
	1	40005FE4	00000358	Write	1.000 us
	2	00000700	9421FFC0	{ Type_Struct 9421FFC0 stwu Instruction r1,-40(r1)	1.338 us
	3	00000704	7C0802A6	7C0802A6 mflr Instruction r0	1.338 us
	4	00000708	93E1003C	93E1003C stw Instruction r31,3C(r1)	1.338 us
	5	0000070C	90010044	90010044 stw Instruction r0,44(r1)	1.338 us

Example: Trace monitors the value of `iCounter` variable while the application is running.

- Create new Trace Trigger in the Analyzer window.
- Set 'Immediately' for the Trigger
- Enable Data Trace and configure Message Control 1 for `iCounter` data accesses only. Select `iCounter` address and check 'Entire Object' range option. The debugger will determine and configure range end address based on the size of the variable.

The trace is configured. Following picture depicts current trace settings.



Initialize the complete system, start the trace and run the program. The trace records all writes to `iCounter` variable.

	Number	Address	Data	Content	Time
	270	40002014	0000003A	iCounter iCounter+0 iCounter+0 iCounter+0 Read	2.132503463 s
1	271	40002014	0000003B	iCounter iCounter+0 iCounter+0 iCounter+0 Write	2.132504450 s
	272	40002014	0000003B	iCounter iCounter+0 iCounter+0 iCounter+0 Read	2.132614113 s
2	273	40002014	0000003C	iCounter iCounter+0 iCounter+0 iCounter+0 Write	2.132615438 s
	274	40002014	0000003C	iCounter iCounter+0 iCounter+0 iCounter+0 Read	2.137054900 s

Time difference between two consecutive data write accesses can be measured using markers. In this particular case, the time difference is 110,988 us.

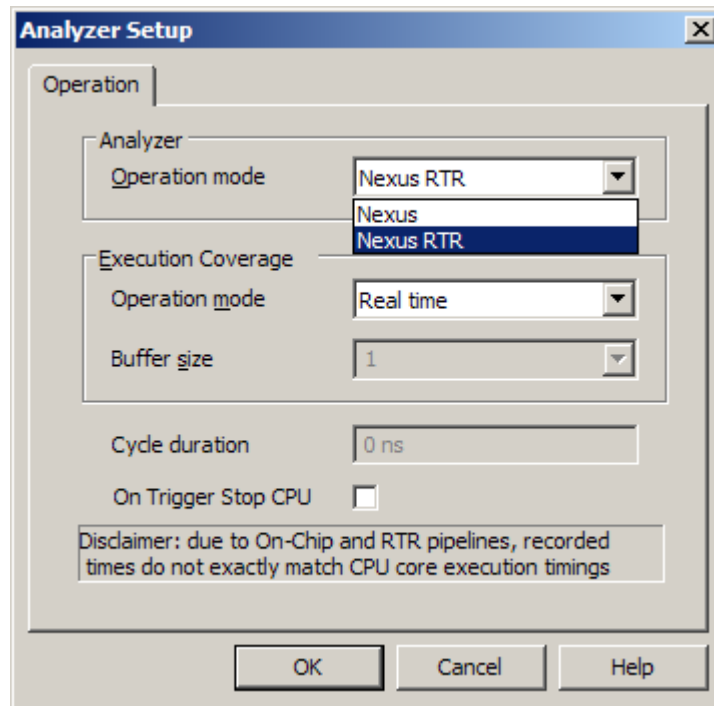
3.2 Nexus RTR

iTRACE GT development system offers some advanced trace features, which are based on iSYSTEM Nexus RTR technology. Nexus RTR is restricted to the e200 program execution bus and default Power ISA instruction set (e.g. MPC555x). **VLE instruction set is not supported, which means none of the Freescale MPC56xx or ST SPC56 devices are supported.**

- 3-Level Trigger
- Unlimited Qualifier
- Watchdog Trigger
- Duration Tracker

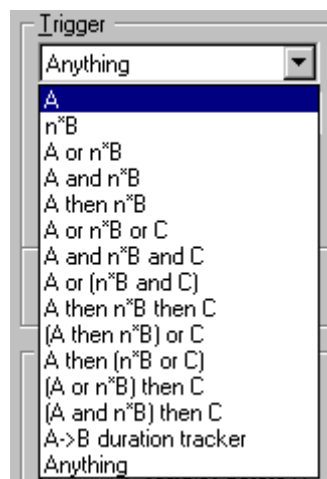
‘Nexus RTR’ must be selected in the Hardware/Analyzer Setup dialog to use these extra features.

Note: Nexus RTR is implemented on iTRACE GT only for 12-bit MDO (Nexus port) implementation only.



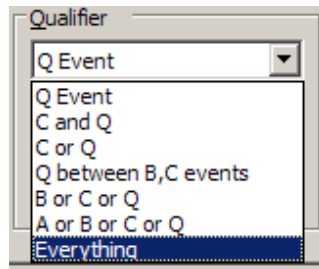
3-Level Trigger

On-chip Nexus resources don't support two or more level triggers, which might be a showstopper sometimes. The iSYSTEM development system offers 3-level trigger applicable to the instruction bus. Events A, B and C can be logically combined in numerous ways, including counter n for B event. All three events can be one or more instruction address matches or ranges. A 2-level trigger example can be found in next Qualifier chapter.

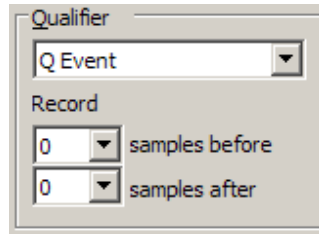


Qualifier

Filter is equivalent term to the Qualifier. To make the most of the trace buffer limited in depth, a qualifier (filter) can be used, which allows the trace to record only CPU events matching the qualifier condition(s) and thus saving memory space for important information only. Typically, 'Q Event' selection is used when using qualifier and can be configured for one or more instruction address matches or ranges.



A so called Pre/Post Qualifier is available besides the prime qualifier. Pre Qualifier can record up to 8 CPU cycles before the qualifier event and Post Qualifier up to 8 CPU cycles after the qualifier event.

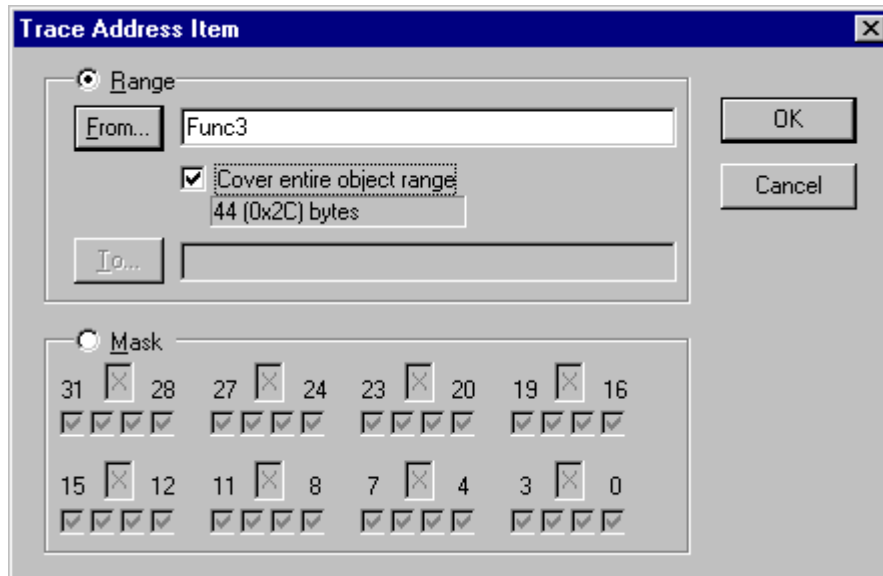


Thereby, the qualifier can be configured in a standard way and then additionally up to 8 CPU cycles can be recorded before and/or after the qualifier. For instance, this allows recording of a function or just its entry point and few instructions recorded before make possible to determine, which code (e.g. function) actually called the inspected function.

Next example demonstrates 2-Level Trigger, Qualifier and Pre Qualifier use.

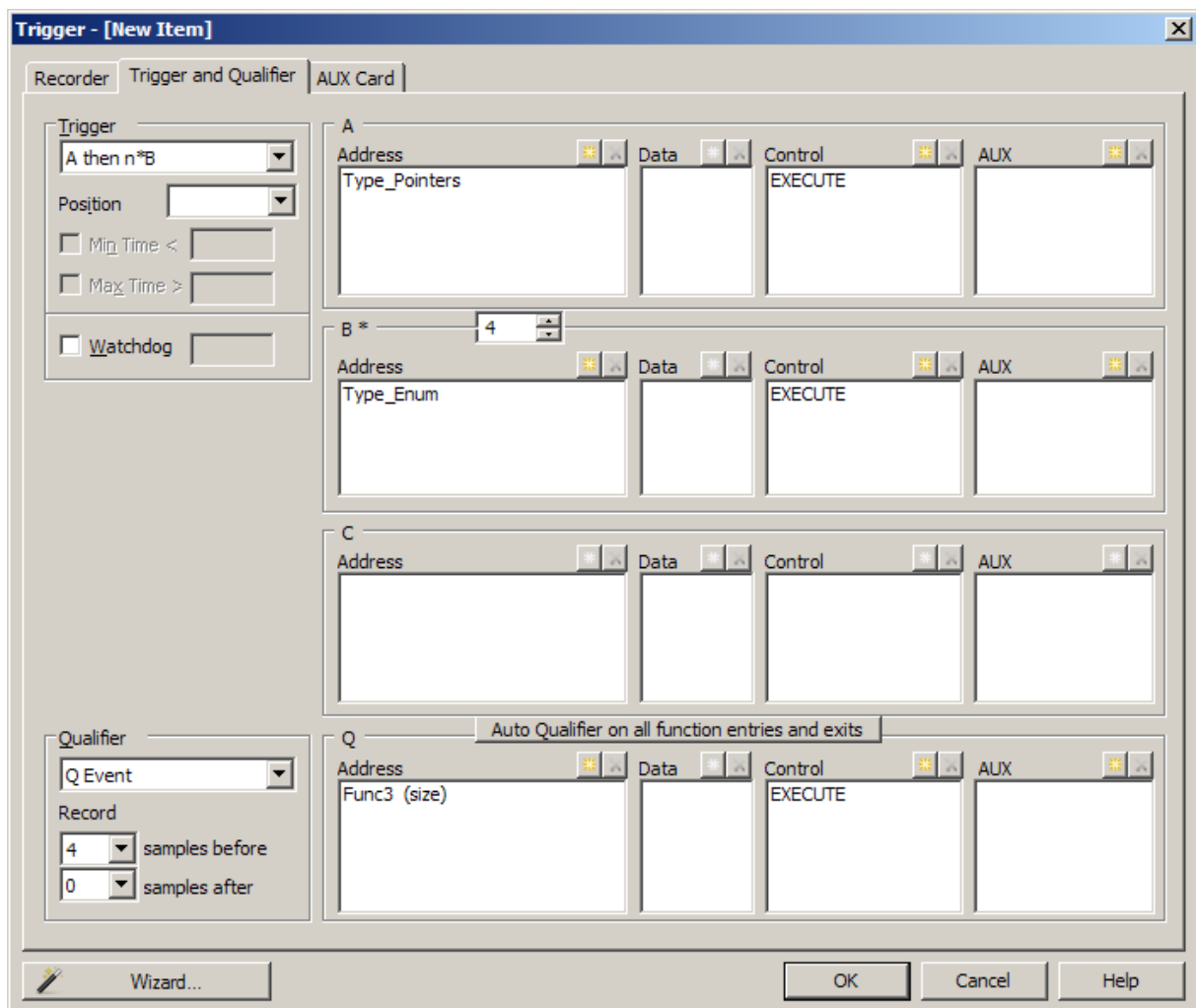
Example: Let's record `Func3` execution after the `Type_Pointers` function is executed and then 4-times `Type_Enum` function is called.

- Create new Trace Trigger in the Analyzer window and open 'Trigger and Qualifier Configuration' dialog.
- Select 'A then n*B' for the trigger condition, specify `Type_Pointers` for the event A address, `Type_Enum` for the event B address and set B counter to 4. Don't forget to set Control bus to 'Executed' for both, A and B events.
- Next select 'Q Event' for the Qualifier. Specify `Func3` for the Q event address and don't forget to 'Check entire object range' option. By doing so, the debugger will extract the size of the `Func3` and configure address range end address accordingly.



- Finally, configure 'Record 4 samples before Qualifier' in the Qualifier filed.

Following picture depicts current trace settings.



Initialize the complete system, start the trace and run the program. Following picture shows the trace record. Green colored line depicts Func3 entry point and yellow colored line Func3 exit point.

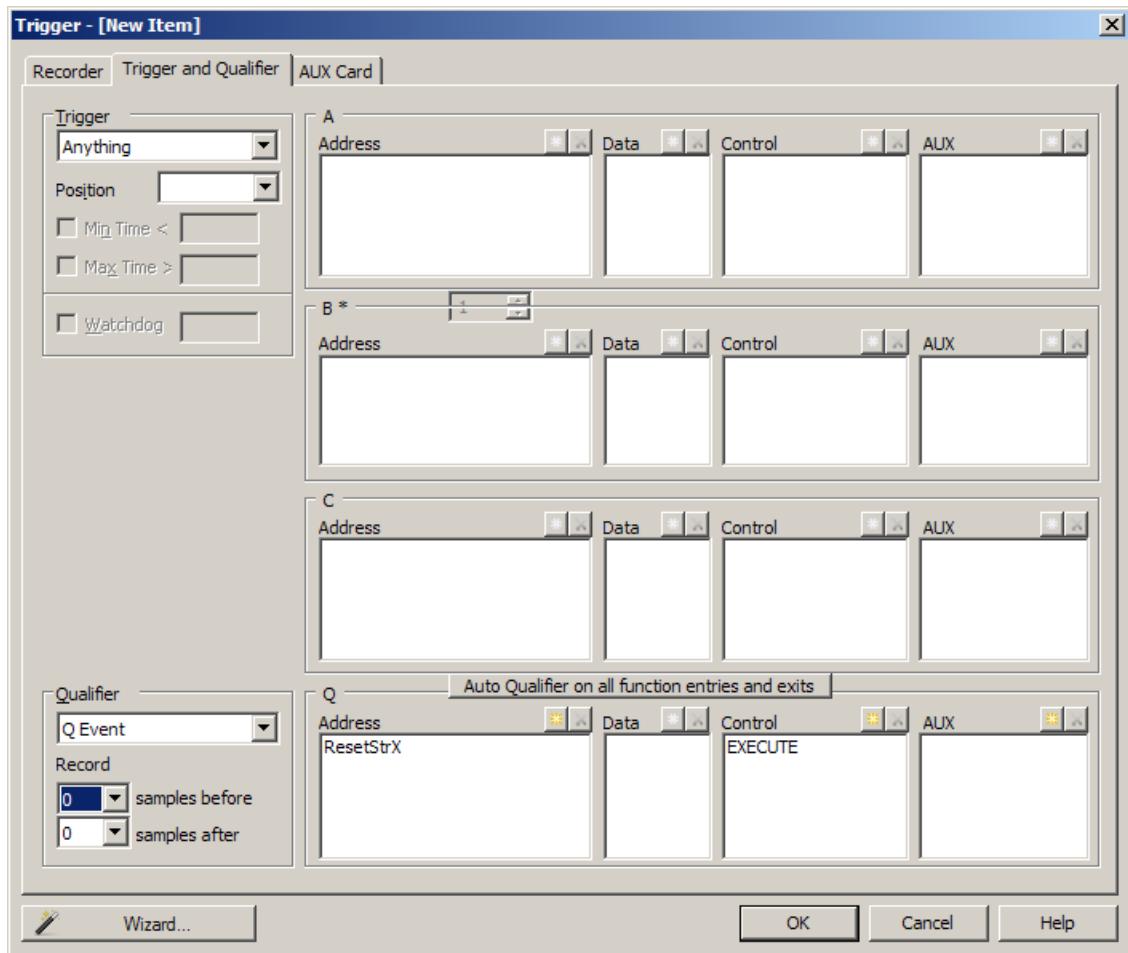
The user is able to determine which code actually called each Func3 function by clicking on any of four lines before Func3 entry point.

Number	Address	Data	Content	Time
45	00000F48	381F000C	pY=ay; 381F000C la r0,0C(r31) Executed	18.894037 ms
46	00000F4C	901F0010	901F0010 stw r0,10(r31) Executed	18.894037 ms
47	00000F50	807F0010	Func3(pY); 807F0010 lwz r3,10(r31) Executed	18.894050 ms
48	00000F54	4BFFFE29	4BFFFE29 bl Func3 (0D7C) Executed	18.894050 ms
49	00000D7C	9421FFE0	{ Func3 9421FFE0 stw r1,-20(r1) Executed	18.910012 ms
50	00000D80	93E1001C	93E1001C stw r31,1C(r1) Executed	18.910025 ms
51	00000D84	7C3F0B78	7C3F0B78 mr r31,r1 Executed	18.910025 ms
52	00000D88	907F0008	907F0008 stw r3,08(r31) Executed	18.910037 ms
53	00000D8C	813F0008	*pY=0; 813F0008 lwz r9,08(r31) Executed	18.910037 ms
54	00000D90	38000000	38000000 li r0,00 Executed	18.910050 ms
55	00000D94	90090000	90090000 stw r0,00(r9) Executed	18.910050 ms
56	00000D98	81610000	} 81610000 lwz r11,00(r1) Executed	18.910062 ms
57	00000D9C	83EBFFFC	83EBFFFC lwz r31,-04(r11) Executed	18.910062 ms
58	00000DA0	7D615B78	7D615B78 mr r1,r11 Executed	18.910075 ms
59	00000DA4	4E800020	Func3_EXIT_ 4E800020 blr Executed	18.910075 ms

Example: Let's use the trace to measure the time between the ResetStrX interrupt routine calls.

- Create new Trace Trigger in the Analyzer window and open 'Trigger and Qualifier Configuration' dialog.
- Select 'Anything' for the trigger condition, specify Q Event for the Qualifier and then define the Q event.

Following picture depicts current trace settings.



Initialize the complete system, start the trace and run the program. Following picture shows the trace record. Time between two consecutive function calls can be easily measured by selecting 'Relative time' from the trace window local menu.

Number	Address	Data	Content	Time
5	200009F8	9421FFE8	{ ResetStrX 200009F8 9421FFE8 stwu Executed	188.621718 ms
6	200009F8	9421FFE8	{ ResetStrX 200009F8 9421FFE8 stwu Executed	282.918054 ms
7	200009F8	9421FFE8	{ ResetStrX 200009F8 9421FFE8 stwu Executed	282.927299 ms
8	200009F8	9421FFE8	{ ResetStrX 200009F8 9421FFE8 stwu Executed	377.223981 ms

M1-M2: 0 ns (NA) P: 471.539445 ms D: -94.305 M2: 471.539445 M1: 471.539445 IDLE

Watchdog Trigger

A standard trigger condition, logically combined from events A, B and C, is not used to trigger directly the trace, but it's responsible for keeping a free running trace watchdog timer from timing out. The trace watchdog time-out is adjustable.

When the trace watchdog timer times out, the trace triggers and optionally stops the application. The problematic code can be found by inspecting the program flow in the trace history.

Usage

If the application being debugged features a watchdog timer, the trace watchdog trigger can be used to trap the situations when the application watchdog timer times out and resets the system.

While the application executes predictably, it periodically calls watchdog reset routine, which resets the watchdog timer before it times out. In case of an external watchdog timer being serviced (refreshed) by the target signal, the external trace input (AUX) can be configured instead of a routine call.

Time-out period of the trace watchdog timer must be less than the period of the application watchdog so the trace can trigger and record CPU behavior before the application watchdog times out and resets the system.

Configuring Watchdog Trigger

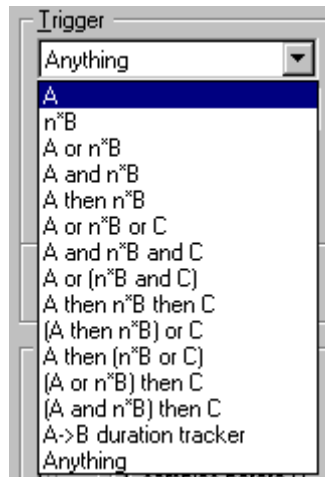
The user needs to enter the trace watchdog time-out period and define the “trace watchdog reset” condition, which can be logically combined from events A, B and C.

- Check the ‘Watchdog’ option and specify the time-out period in the ‘Trigger’ field in the ‘Trigger and Qualifier Configuration’ dialog.



Trigger field

- Next, define the “trace watchdog reset” condition. Typically, only event A is selected for the “trace watchdog reset” condition and then e.g. a reset watchdog routine, resetting the watchdog, is configured for the event A. Of course, a more complex condition can be set up instead of the event A only.



Trigger conditions

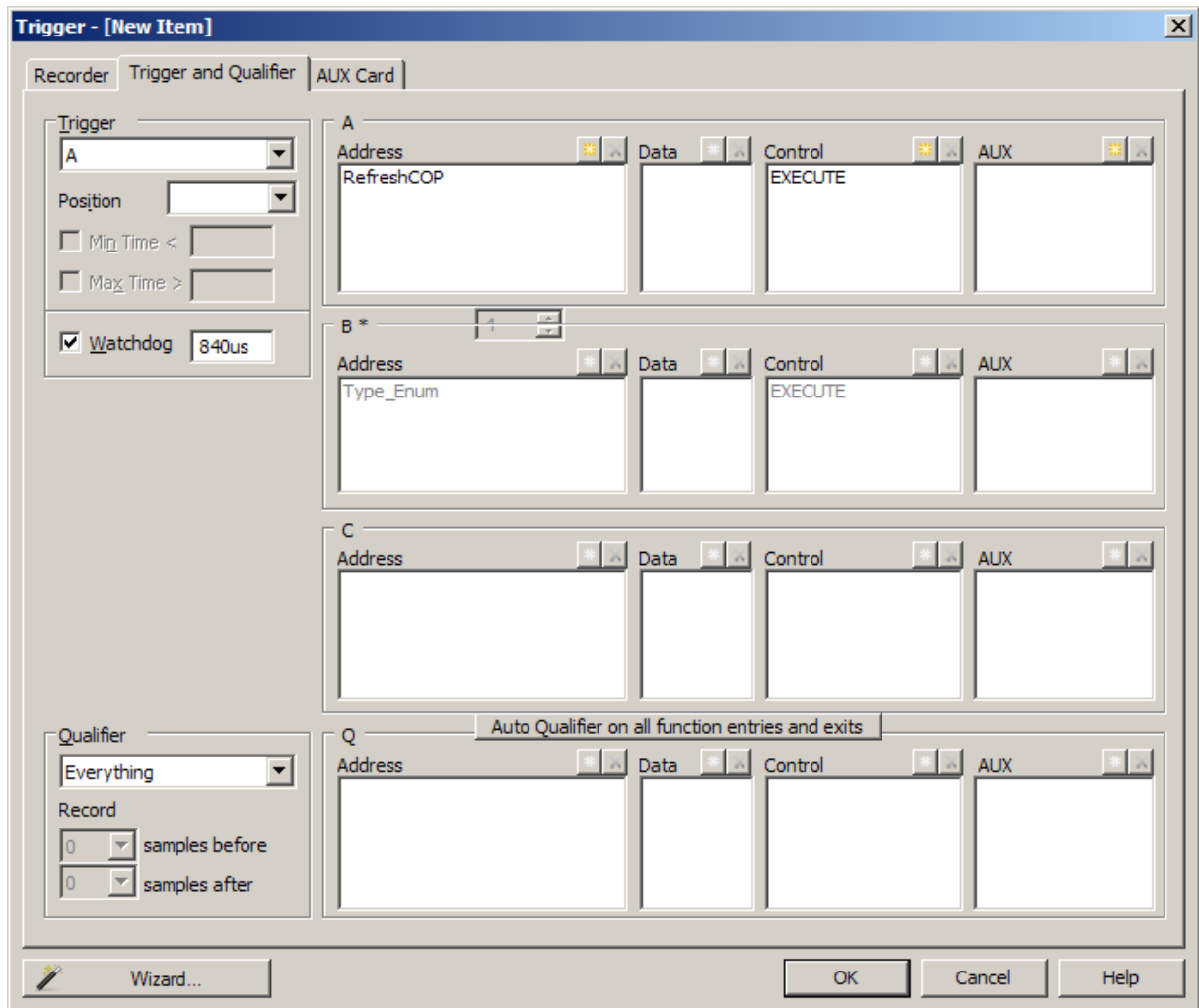
Example: Target application features on-chip COP watchdog, which enables the user to check that a program is running and sequencing properly. When the COP is being used, software is responsible for keeping a free running watchdog timer from timing out. If the watchdog timer times out it's an indication that the software is no longer being executed in the intended sequence; thus a system reset is initiated.

When COP is enabled, the program must call `RefreshCOP` routine during the selected time-out period. Once this is done, the internal COP counter resets to the start of a new time-out period. If the program fails to do this, the part will reset. The COP timer time-out period is 890 μ s in this particular example. It may vary between the applications since it's configurable. The watchdog timer is reset within 800 μ s during the normal program flow.

The trace is going to be configured to trap COP time out before it initiates a system reset. The user can find the code where the program misbehaves in the trace history.

- Create new Trace Trigger in the Analyzer window and open 'Trigger and Qualifier Configuration' dialog.
- Select 'A' for the trigger condition, check the 'Watchdog' option and enter 840 μ s for the trace watchdog timer time-out period.
- Specify `RefreshCOP` function call for an event A (reset sequence). Don't forget to select 'Executed' for the Control bus.

Below picture depicts current trace settings.



While the application operates correctly, the trace never triggers. The trace triggers when the application misbehaves, that is when `RefreshCOP` is no longer called within 840 μ s, and record the program behavior before that. The user can find out why the watchdog wasn't serviced by analyzing the trace record.

If longer trace history is required, select medium or maximum trace buffer size and position the trace trigger at the end of the trace buffer.

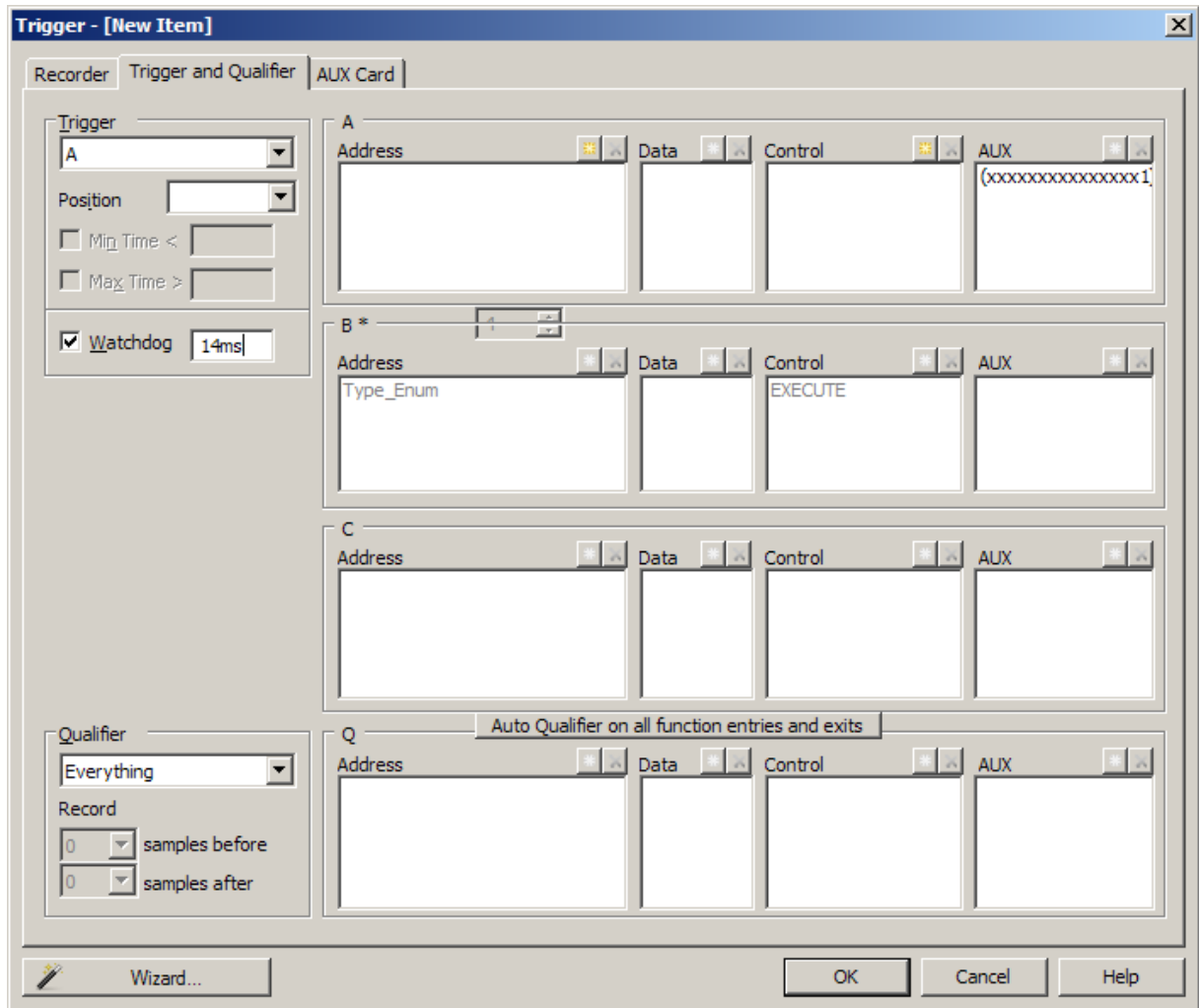
Example: The application features (external) target watchdog timer, which is normally periodically reset every 15 ms by the `WDT_RESET` target signal.

The trace needs to be configured to trap the target watchdog timer time out before it initiates a system reset Then the user can find the code where the program misbehaves using the trace history.

The `WDT_RESET` target signal is connected to one of the available external trace inputs (e.g. `AUX0`). Refer to the hardware reference document delivered beside the emulation system to obtain more details on locating and connecting the `AUX` inputs.

- Create new Trace Trigger in the Analyzer window and open 'Trigger and Qualifier Configuration' dialog.
- Select 'A' for the trigger condition, check 'Watchdog' option and enter 14 ms for the trace watchdog timer time-out period.
- Configure `AUX0=1` for the event A.

The trace will trigger as soon as the target WDT_RESET signal stops resetting the target watchdog within 14 ms period. Below picture depicts current trace settings.



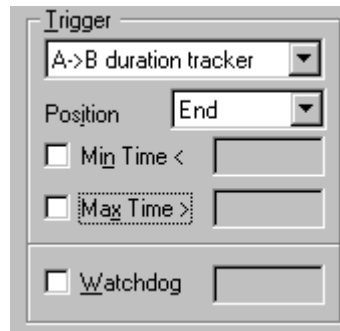
While the application operates correctly, the trace never triggers. The trace triggers when the application misbehaves, that is when RefreshCOP is no longer called within 840 μ s, and record the program behavior before that. The user can find out why the watchdog wasn't serviced by analyzing the trace record.

If longer trace history is required, select medium or maximum trace buffer size and position the trace trigger at the end of the trace buffer.

Duration Tracker

The duration tracker measures the time that the CPU spends executing a part of the application constrained by the event A as a start point and the event B as an end point. Typically, a function or an interrupt routine is an object of interest and thereby constrained by events A and B. However, it can be any part of the program flow constrained by events A and B.

Both events can be defined independently as an instruction fetch from the specific address or an active trace auxiliary (AUX) signal.



Trigger field

Duration Tracker provides following information for the analyzed object:

- Minimum time
- Maximum time
- Average time
- Current time
- Number of hits
- Total profiled object time
- Total CPU time

Duration tracker results are updated on the fly without intrusion on the program execution. The duration tracker can trigger when the elapsed time between events A and B exceeds the limits defined by the user. Then the code exceeding the limits can be found in the trace window. Maximum (Max Time) or minimum time (Min Time) or both can be set for the trigger condition.

Set maximum time when a part of the program e.g. a function must be executed in less than T_{MAX} time units.

Set minimum time when a part of the program e.g. a function taking care of some conversion must finish the conversion in less than T_{MIN} time units.

Max Time is evaluated as soon as the event B is detected after the event A or simply, Current Time is compared against Max Time after the program leaves the object being tracked.

Min Time is compared with the Current Time as soon as the event A is detected or simply, Current Time is compared against Min Time as soon as the program enters the object being tracked.

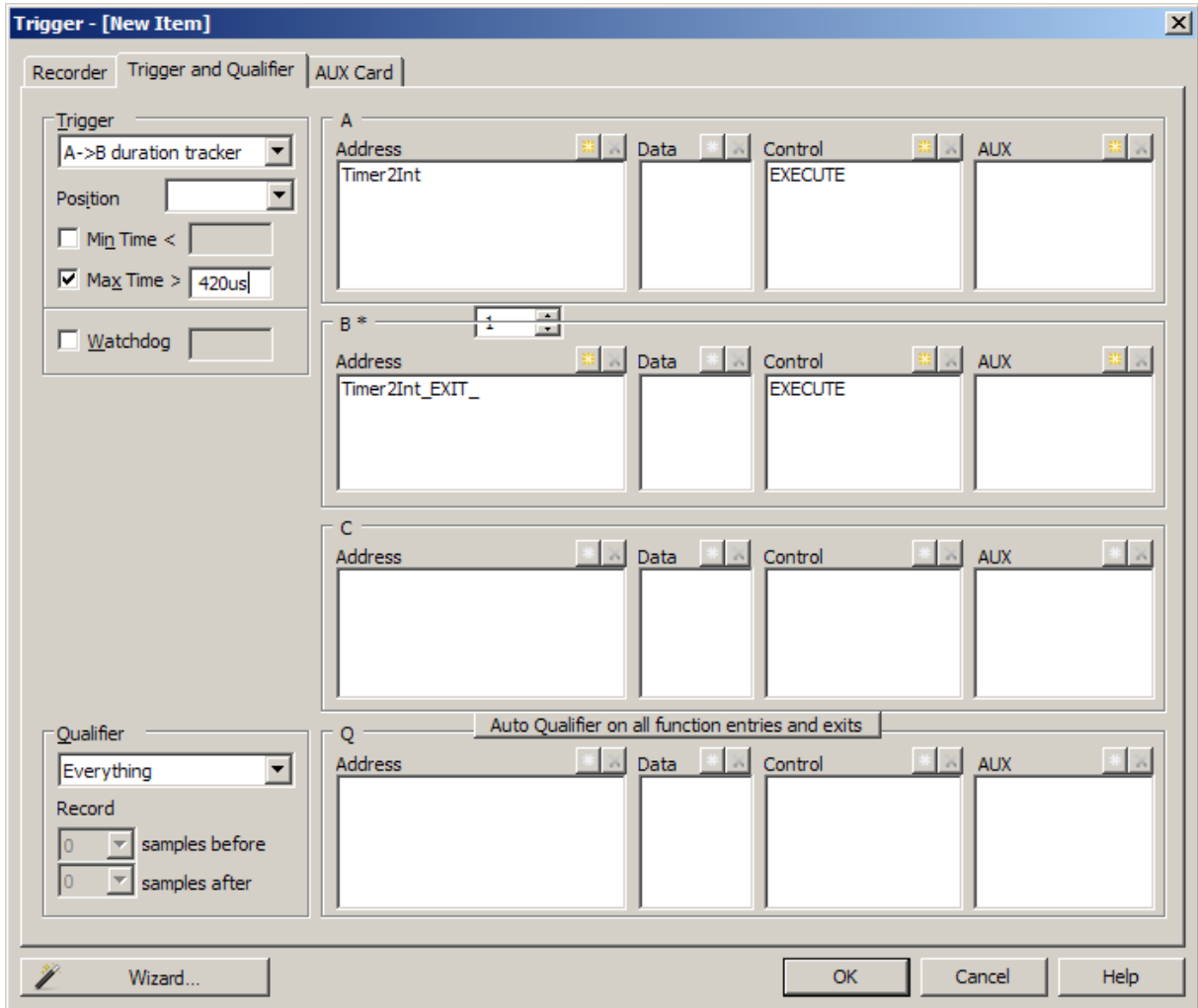
Based on the trace history, the user can easily find why the program executed out of the normal limits. Trace results can be additionally filtered out by using the qualifier.

Example: There is a `Timer2Int` interrupt routine, which terminates in $420 \mu s$ under normal conditions. The user wants to trigger and break the program execution when the `Timer2Int` interrupt routine executes longer than $420 \mu s$, which represent abnormal behaviour of the application.

- Create new Trace Trigger in the Analyzer window and open 'Trigger and Qualifier Configuration' dialog.
- Select, 'A->B duration tracker' for the trigger condition.

- Next, we need to define the object of interest. Select, `Timer2Int` entry point for the event A and `Timer2Int_EXIT_` exit point for the event B. Make sure you select 'Fetch' access type for the control bus for both events since the object of our interest is the code.
- Check the 'Max Timer >' option and enter 420 μ s for the limit.

Below picture depicts current trace settings.



Before starting the trace session, open Duration Tracker Status Bar using the trace toolbar (Figure 34). Existing trace window is extended by the Duration Tracker Status Bar, which displays results proprietary for this trace mode.



Duration Tracker Status Bar toolbar

The trace is configured. Initialize the system, start the trace and run the application.

First, let's assume that the application behaves abnormally and the trace triggers. It means that the CPU spent more than 420 μ s in `Timer2Int` interrupt routine. Let's analyze the trace content (Figure 35).

	Number	Address	Content	Time
	-5	00000898	00000898 800B0004 lwz Executed	-976 ns
	-4	0000089C	0000089C 7C0803A6 mtlr Executed	-976 ns
	-3	000008A0	000008A0 83EBFFFC lwz Executed	-963 ns
	-2	000008A4	000008A4 7D615B78 mr Executed	-963 ns
	-1	000008A8	Timer2Int_EXIT_ 000008A8 4E800020 blr Executed	-951 ns
T	0	0000035C	Type_Enum(); 0000035C 48000551 bl Executed	0 ns
	1	000008AC	{ Type_Enum 000008AC 9421FFD8 stwu Executed	662 ns
	2	000008B0	000008B0 7C0802A6 mflr Executed	675 ns
	3	000008B4	000008B4 93E10024 stw Executed	675 ns

Trace Window results

Go to the trigger event by pressing 'J' key or selecting 'Jump to Trigger position' from the local menu. The trace window shows the code being executed 420 μ s after the application entered Timer2Int interrupt routine.

By inspecting the trace history we can find out why the Timer2Int executed longer than 420 μ s. Normally, the routine should terminate in less than 420 μ s.

Next, let's analyze duration tracker results displayed in the Duration Tracker Status Bar.

Duration tracker statistics		Count	27			×
Min	54.950 us	Current	416.850 us	Total	27.884550 ms	
Max	416.850 us	Average	235.896 us	Total region	6.369216 ms (22.84%)	

Duration Tracker Status Bar

Duration Tracker Status Bar reports:

Timer2Int minimum execution time was 54.95 μ s

Timer2Int average execution time was 235.90 μ s

Timer2Int maximum and current execution time was 416.85 μ s

Last execution of the Timer2Int took longer than 420 μ s, since we got a trigger, which stopped the program. This time cannot be seen yet since the program stopped before the function exited. The Status Bar displays last recorded maximum and current time.

Timer2Int routine completed 27 times.

The CPU spent 6.37 ms in the Timer2Int routine being 22.85% of the total time.

The duration tracker ran for 27.88 ms.

If the `Timer2Int` routine doesn't exceed `Min Time` or `Max Time` values, the debugger exhibits run debug status and the duration tracker status bar displays current statistics about the tracked object from the start on. Status bar is updated on the fly while the application is running.

Note 1: Events A and B can also be configured on external signals. In case of an airbag application, the event A can be a signal from the sensor unit reporting a car crash and the event B can be an output signal to the airbag firing mechanism. Duration tracker can be used to measure the time that the airbag control unit requires to process the sensor signals and fire the airbags. Such an application is very time critical and stressed. It can be tested over a long period using Duration Tracker, which stops the application as soon as the airbag doesn't fire in less than T_{MIN} and display the critical program flow.

Note 2: Duration Tracker can be used in a way in which it works like the execution profiler (one of the analyzer operation modes) on a single object (e.g. function/routine) profiting two things, the results can be uploaded on the fly while the CPU is running and the object can be tracked over a long period. Define no trigger and the duration tracker updates statistic results while the program runs.

3.3 Troubleshooting

- **Missing program code**

If a "missing program code" message is displayed in the trace, it means that the program was executed at addresses where no code image is available in the download file. The debugger needs complete code image for the correct trace reconstruction! The code not reported in the download file or a self modifying code cannot be traced. In order to analyze which code is missing in the trace, click on the last valid trace frame before the "missing program code" message. This will point to the belonging program point in the source or disassembly window. Set a breakpoint there, run the program until the breakpoint is hit and then step the program (F11) from that point on to see where the program goes.

Number	Address	Data	Content	Time
192.8	000002D4	608402DC	"D:\DOCUME~1\gd63671\LOCALS~1\Temp\cc00aa 608402DC ori r4,02DC Instruction	195.631 us
192.9	000002D8	4E800020	"D:\DOCUME~1\gd63671\LOCALS~1\Temp\cc00aa 4E800020 blr Instruction	195.818 us
195.0	40000000	00000000	Missing program code	195.961 us
198.0	000002DC	7CA803A6	"D:\DOCUME~1\gd63671\LOCALS~1\Temp\cc00aa 7CA803A6 mtlr r5 Instruction	197.004 us
198.1	000002E0	4E800020	"D:\DOCUME~1\gd63671\LOCALS~1\Temp\cc00aa 4E800020 blr Instruction	197.190 us

- **Trigger position**

With Nexus trace, which is a message based trace, actual trigger point (frame 0) is most likely not to be displayed next to the instruction which generated the trigger event. The Nexus trace port broadcasts only addresses of non-sequential branch jumps. All the sequential code in between is reconstructed by the debugger based on the code image available from the download file. There is no exact information to which of the inserted (reconstructed) sequential instructions the trigger event belongs. Nexus trace port broadcasts a dedicated trace trigger message beside the non-sequential branch messages.

For example, if there are 30 sequential instructions between the two non/sequential jumps and there was a trigger event in between, trace will always depict the trigger at the same position regardless which one of the 30 instructions generated the trigger event. That's why you probably see the misalignment between the trigger event and the belonging code.

4 Profiler

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.

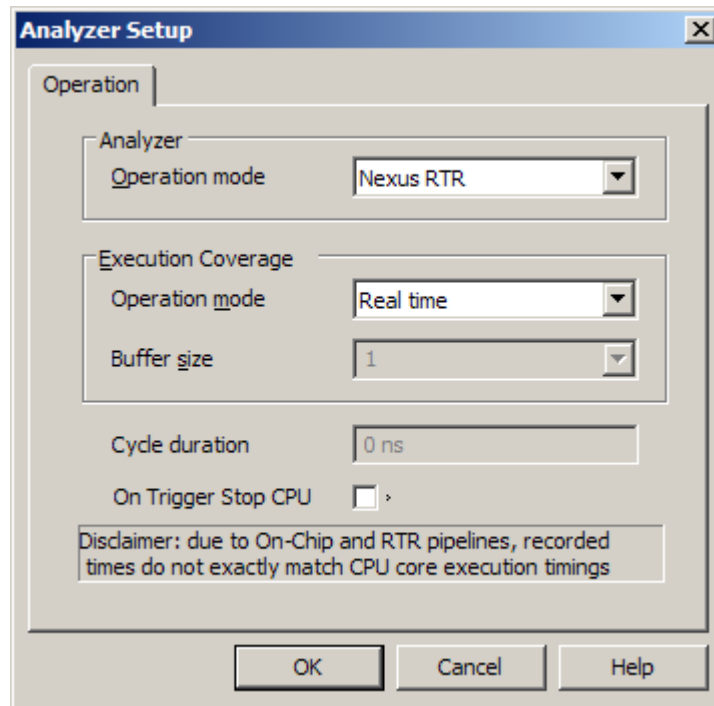
The development system features a so called real-time profiler and off-line profiler. Off-line profiler is entirely based on the trace record. It first uses trace to record a complete program flow and then off-line, functions' entry and exit points are extracted by means of software, the statistic is run over the collected information and finally the results are displayed. Real-time profiler is based on iSYSTEM RTR technology, which allows the profiler to capture only functions' entry and exit points and not complete program flow. This way, profiler session time is increased in most cases. Note that total session time depends on the application and amount of profiled objects.

Refer to a separate document titled Analyzer User's Manual for more details on general handling & configuring the analyzer window and its use. Next, refer to a separate document titled Profiler User's Guide for more details on profiler and its use.

Nexus RTR is implemented on iTRACE GT for 12-bit MDO Nexus port only and it supports Power ISA instruction set only. VLE instruction set is not supported.

4.1 Typical Use

To use off-line profiler select 'Nexus' analyzer operation mode on iTRACE GT and 'iTRACE/On-Chip' on iC5000 development system.



For real-time profiler use, select 'Nexus RTR' trace operation mode in the 'Hardware/Analyzer Setup' dialog. **Note that this operation mode is not suitable for applications running under OS since it doesn't support data profiling. OS typically keeps the task ID in global data variable which must be profiled beside the functions.**

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

When using functions profiler in the application with an operating system, the task switch absolutely and unconditionally must be profiled too! Task switch can be profiled via Data (typically) or OTM profiler.

If the application doesn't use any operating system, following Profiling OS Task Switches explanation is irrelevant and can be skipped.

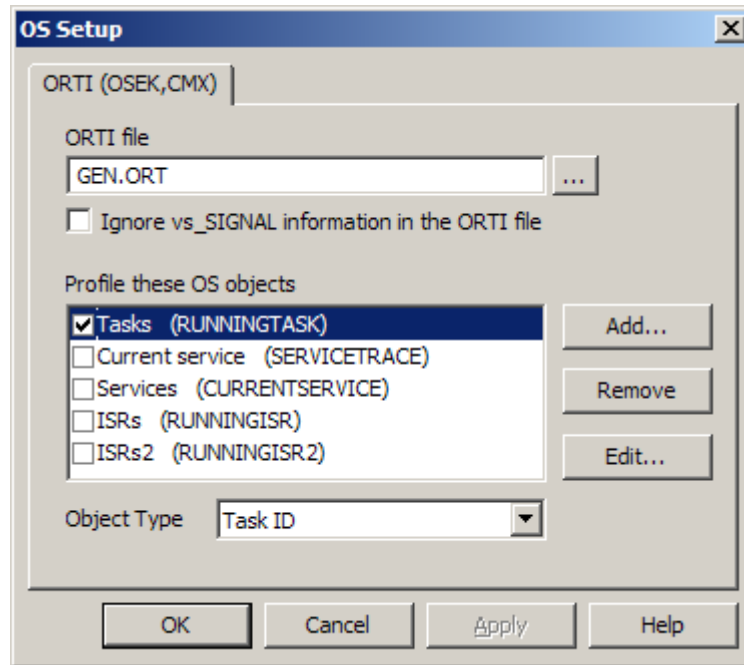
Profiling OS Task Switches

Typically, per default an operating system has a global variable, which keeps the information on current task ID. Operating system writes to this variable on every task switch, which can then be traced and profiled by the Data profiler.

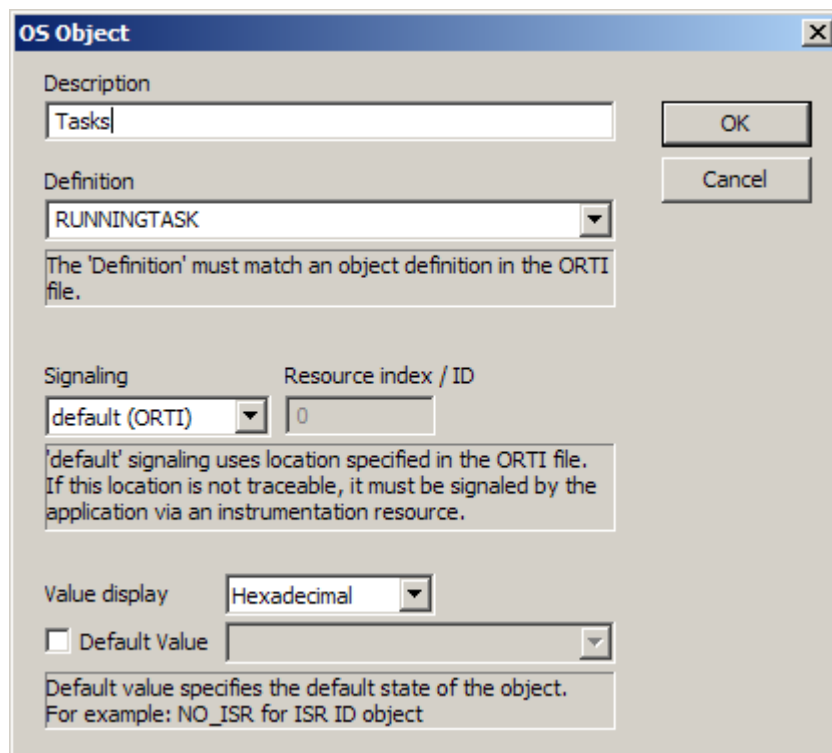
Operating system (OS) awareness is configured in the 'Debug/Operating System' dialog. . In case of OSEK operating system, debugger requires a so called orti file (.ort), which is generated by the OSEK and contains all viable information for debugger's OS awareness.

Make sure that only 'Tasks (RUNNING TASK)' selection is checked in the 'Profile these OS objects' field. Otherwise profiler may report profiler error: "2: Configuration error: Too many Data Areas defined".

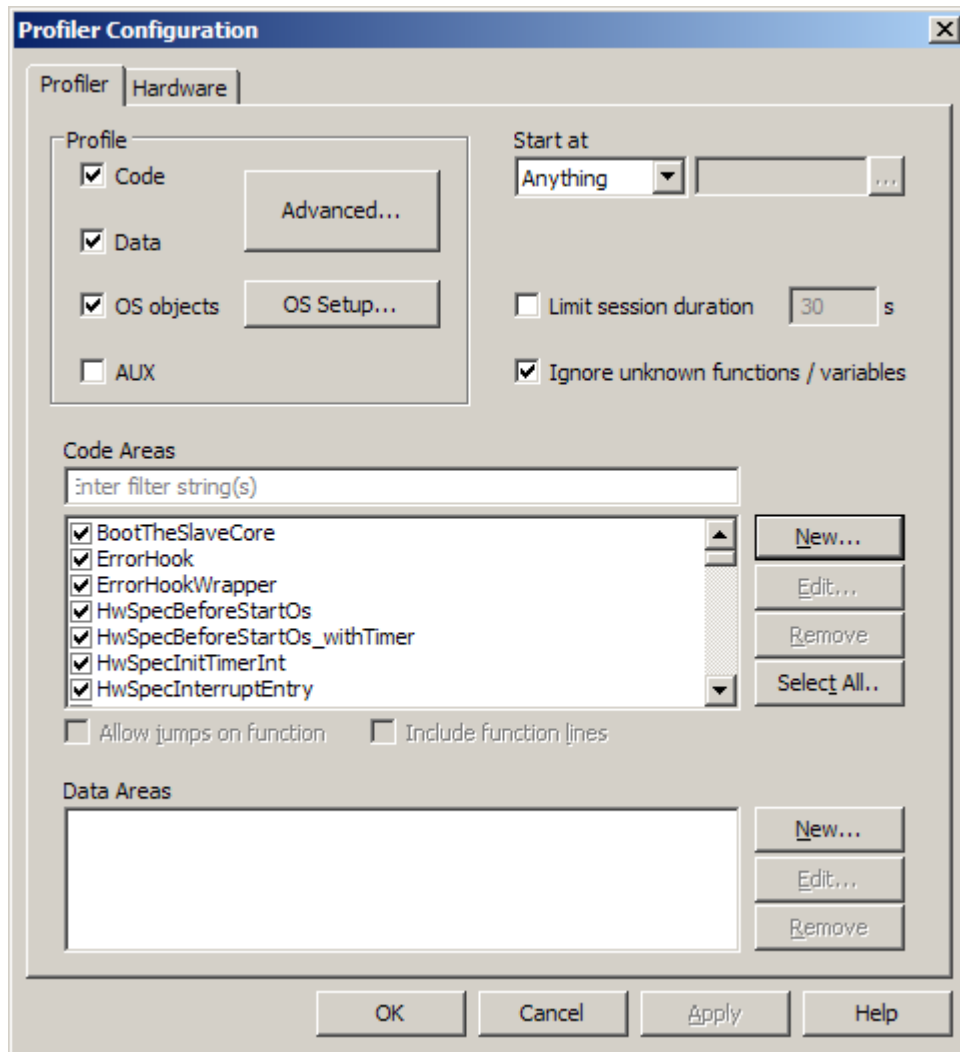
Note: ORTI file must match with the download code. Any misalignment will yield wrong profiler results



Double click on the checked 'Task (RUNNINGTASK)' selection in the OS Setup dialog. Under Signaling 'default (ORTI)' value is selected. Since ORTI typically reports task ID in a global variable, the debugger will, based on this setting, configure data profiler to profile a global data variable keeping the current task ID.



Next picture shows the necessary Data profiler settings. Refer to the Profiler User's Guide for more details on Data profiler settings and use. These settings would be sufficient to profile OS tasks only. It is recommended to run the profiler with these settings first to see if the OS tasks are profiled and if the sequence of task switches makes sense.

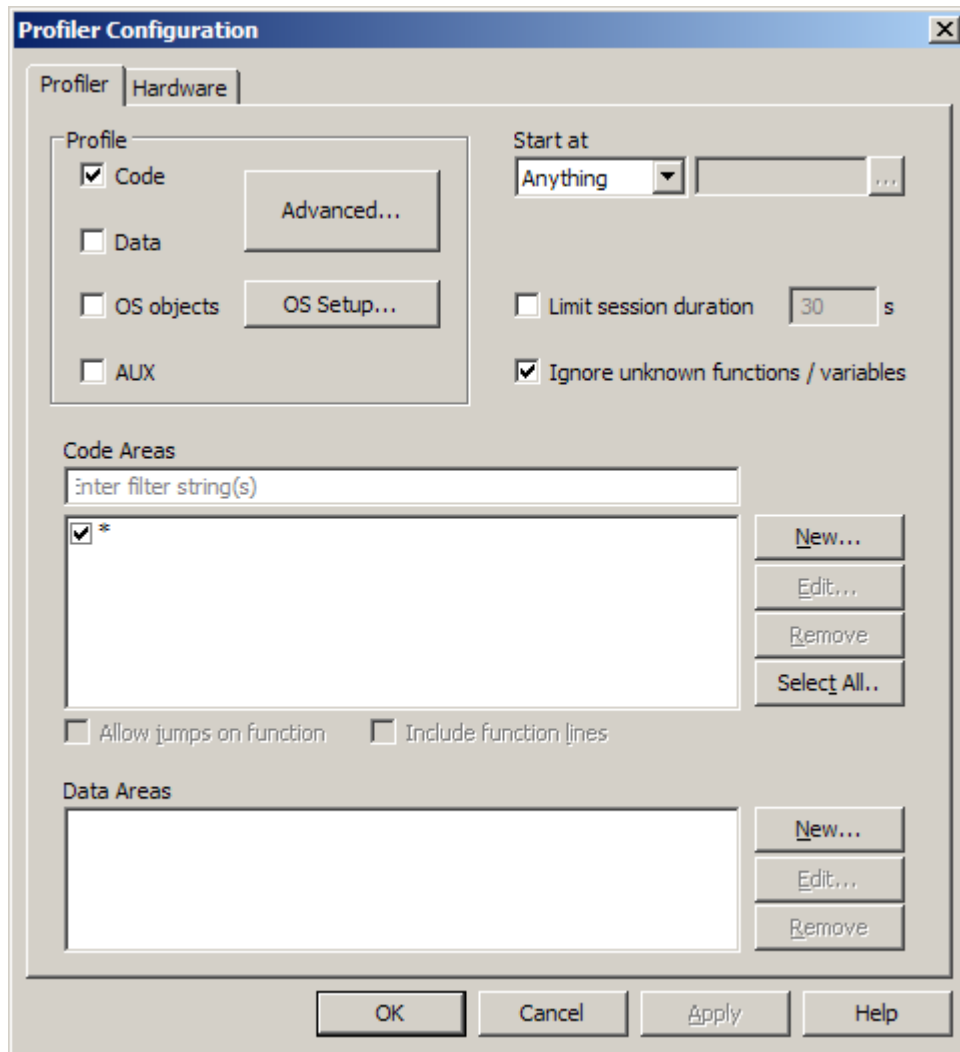


Profiler settings for application with an operating system

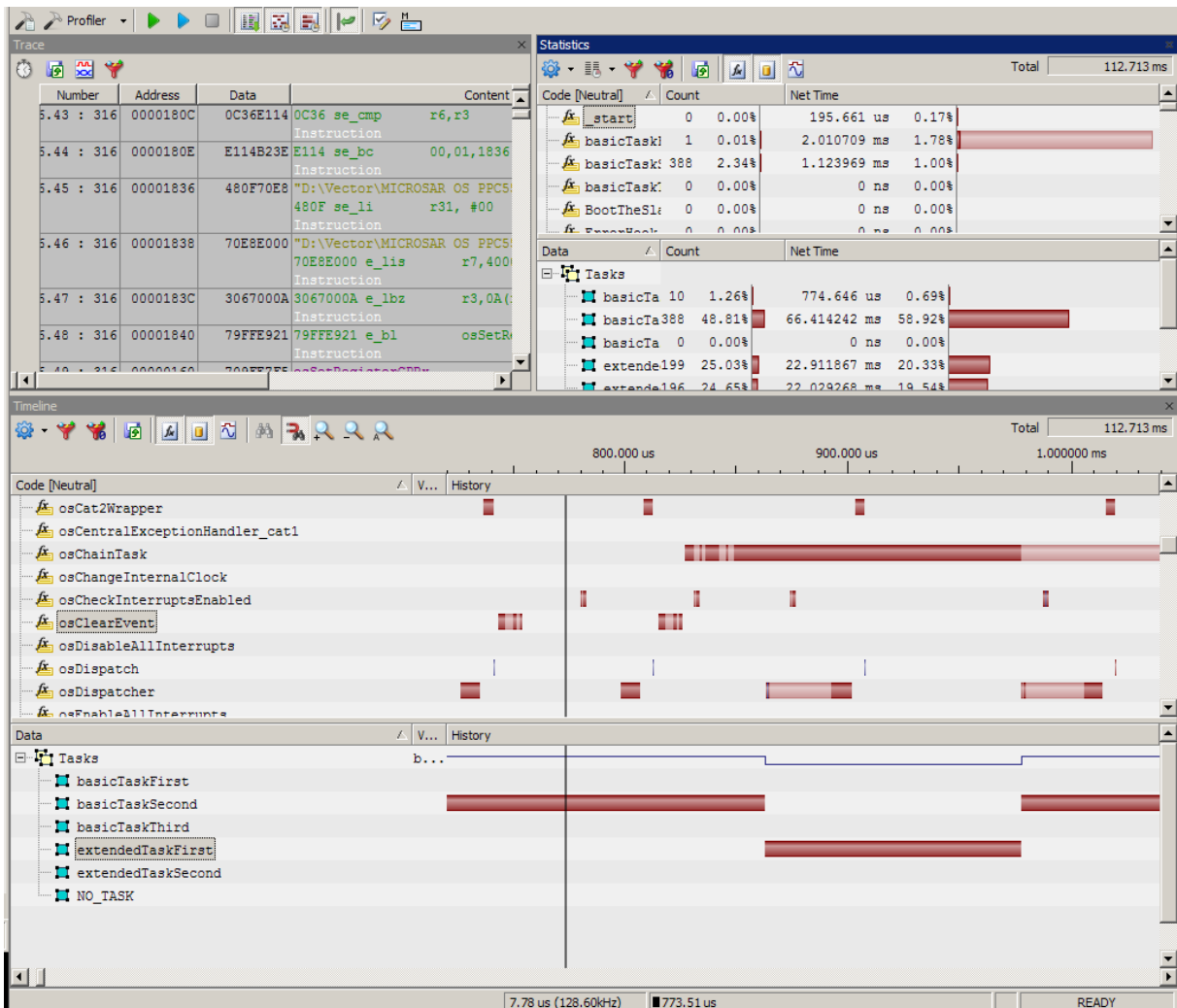
Profiled functions are selected by pressing ‘New...’ button. It’s recommended that ‘All Functions’ option is selected for the beginning.

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

Next picture shows valid profiler configuration for application with no operating system.



Profiler is configured. Reset the application, start Profiler and run the application. The Profiler will stop recording data 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.



Profiler results in the Analyzer window

4.2 Troubleshooting

- Incorrect time information

If the application uses any operating system (e.g. OSEK) make sure that the debugger is aware of it. Profiling functions without task awareness yields bad profiler results. Operating system awareness is configured in the 'Debug/Operating System' dialog and data profiler (must profile OS task ID) is configured within the profiler window.

5 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 and real-time 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 trace depth) and then offline executed instructions and source lines are extracted by means of software and finally the results displayed.

Real-time execution coverage is based on a hardware logic, which in real-time registers all executed program addresses. It features statement coverage but no decision coverage since it keeps the information on all executed

program addresses but without any history, which would tell which address was executed when and in what order.

The major advantage of the real-time execution coverage is that it can run indefinitely, which does not apply for the off-line coverage. On the other hand, off-line execution coverage provides decision coverage metrics which real-time execution coverage doesn't. It's up to the user then which one to use.

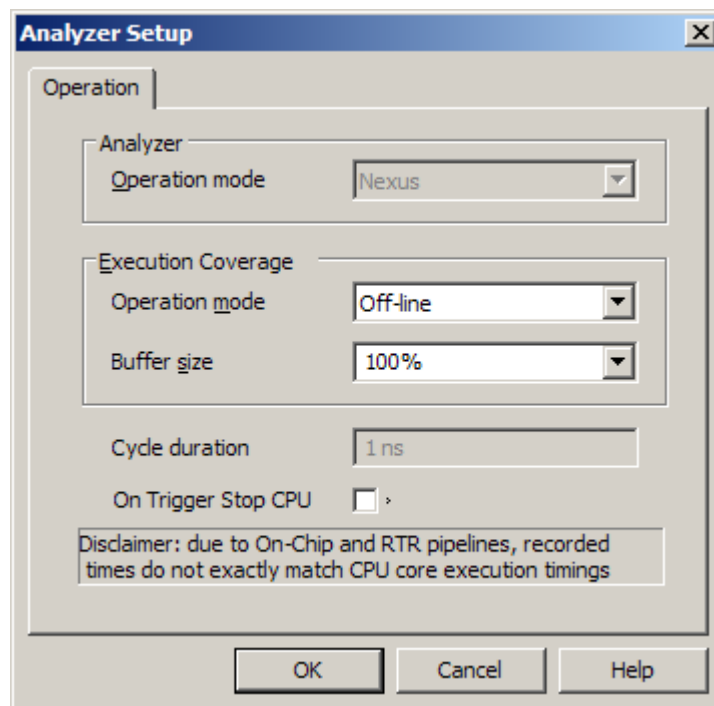
Refer to a separate Execution Coverage User's Guide for more details on execution coverage configuration and use.

Nexus RTR is implemented on iTRACE GT for 12-bit MDO Nexus port only and it supports Power ISA instruction set only. VLE instruction set is not supported.
Execution Coverage cannot be used with Trace or Profiler at the same time.

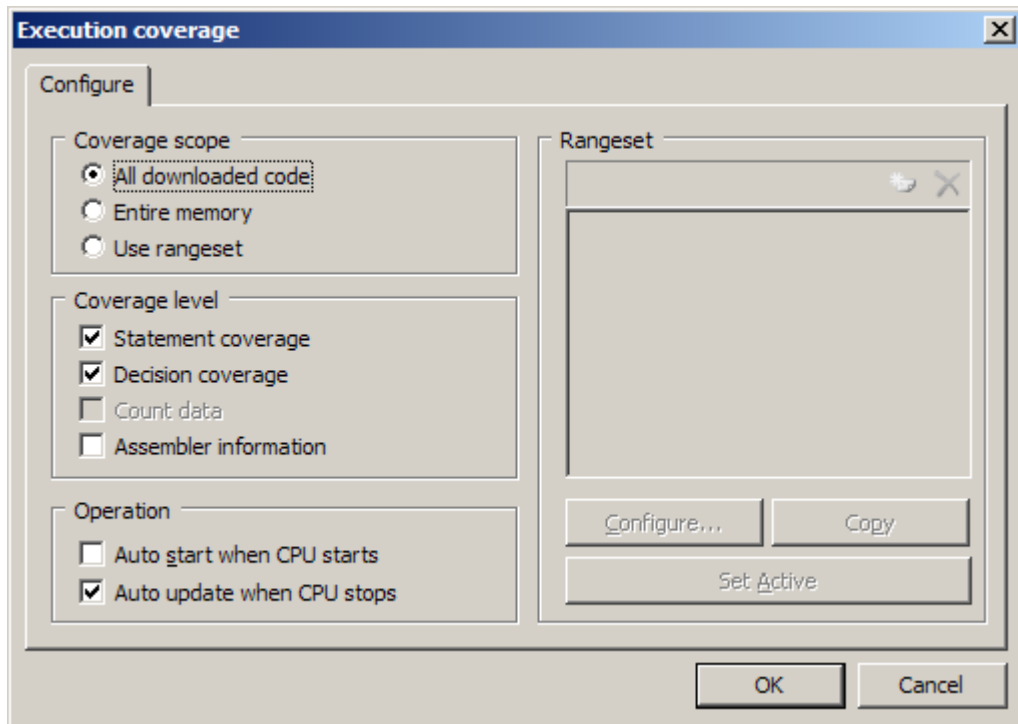
5.1 Typical Use

To use off-line execution coverage, select 'Nexus' analyzer operation mode on iTRACE GT and 'iTRACE/On-Chip' on iC5000 development system and specify working execution coverage buffer size in the 'Hardware/Analyzer Setup' dialog.

For real-time execution coverage use (where available), select 'Nexus RTR' analyzer operation mode. Buffer size setting is not applicable for this mode.



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
..\.\.\.\.\.BSW\O\		388/987 (39%)		0x12DE/0x280E (47%)	(55t,25f,17b)/153 (37%)	0...267697
D:\Vector\MICROSAR O		309/805 (38%)		0xBE8/0x1B30 (44%)	(51t,18f,11b)/125 (36%)	0...87336
D:\Vector\MICROSAR O		1/28 (4%)		0x2/0xC6 (1%)	(0t,0f,0b)/5 (0%)	0...1
D:\Vector\MICROSAR O		0/4 (0%)		0x0/0x8 (0%)		
D:\Vector\MICROSAR O		41/82 (50%)		0x1C2/0x334 (55%)	(4t,4f,1b)/14 (36%)	0...956
tcb\						
main.c		37/68 (54%)		0x24A/0x370 (67%)	(0t,0f,3b)/4 (75%)	0...13289
ErrorHook						
{		0/6 (0%)		0x0/0x2C (0%)		
gErrorCode = Error		0/1 (0%)		0x0/0x10 (0%)		
guiError = OSEError		0/1 (0%)		0x0/0x2 (0%)		
usrErrorHook();		0/1 (0%)		0x0/0xA (0%)		

Execution Coverage results

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.