
Technical Notes

ARM ETM On-Chip Trace

Contents

Contents.....	1
1 Trace.....	2
1.1 ETM Trigger and Qualifier Configuration.....	3
1.1.1 Resource Configuration – Address/Data Configuration	4
1.1.2 Event Configuration	5
1.1.3 Resource Configuration – Counters Configuration	6
1.1.4 Resource Configuration – Sequencer Configuration	6
1.1.5 Trigger Configuration.....	7
1.1.6 Trace Enable Configuration.....	7
1.1.7 View Data Configuration.....	8
1.1.8 Stall CPU on FIFO Overflow	9
1.2 Troubleshooting Scenarios.....	11
1.2.1 Record everything.....	11
1.2.2 Plain Trigger Configurations	12
1.2.3 Advanced Trigger	21
2 BackTrace.....	33
3 Execution Coverage.....	35
4 Profiler.....	37

1 Trace

ARM (on-chip) ETM provides a fairly good trace interface, which abilities vary from implementations (4, 8 or 16 data pins). Beside these, three status pins are added, through which the CPU reports the type of the CPU cycle: whether the instruction was executed or not, whether branch or trigger was executed, etc. When a relative branch was executed by the CPU, the ETM reports the new address through the ETM data pins. When an absolute branch is executed, the address must not be reported, since it can be calculated by the debugger with the help of the download code.

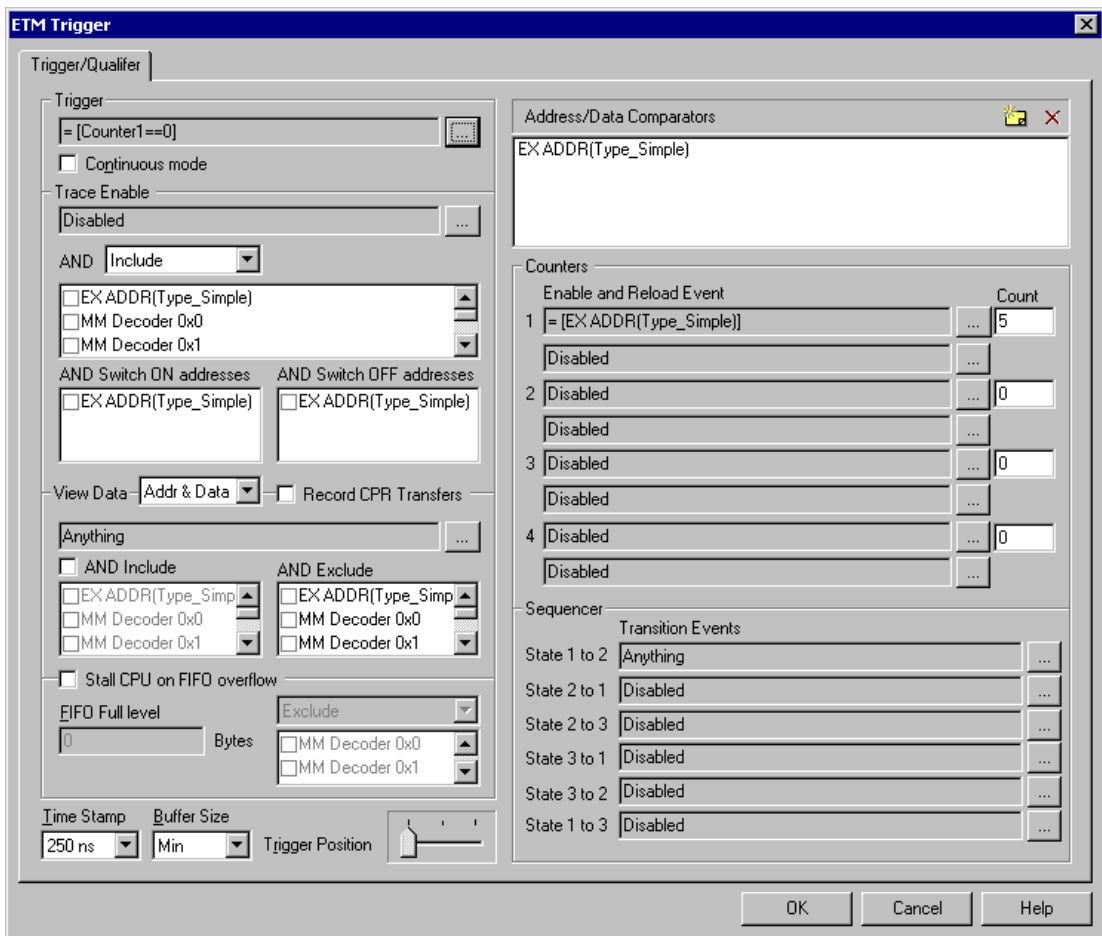
The most valuable information, reported by the ETM, is whether an instruction was executed or not. Based on the information, saved by the debugger, with the help of the download code the program execution can be reconstructed off-line.

ETM can also save data cycles, even in combination with the code. If the requirements for data transfer through the ETM are too large, there are two options: either the CPU can stop, if the data was not yet transmitted, or the data can go lost. These options can be set in the debugger. Also, the ETM incorporates its own trigger system.

Trace Features (iTRACE PRO):

- External trace buffer
- Instruction and Data Trace
- On-Chip Trigger and Qualifier
- Time Stamps
- AUX inputs
- ETM Reconstruction
- Advanced external trigger
- Profiler
- Execution Coverage

1.1 ETM Trigger and Qualifier Configuration



ETM Trigger dialog

Recording is stopped after the trigger event occurs and the trace buffer fills up. Number of samples recorded before and after the trigger condition can be selected with Trigger Position setting, if the maximum buffer is selected.

The Buffer Size determines the depth of the trace buffer (depending on the hardware used). If possible, always use smaller buffer sizes. This will decrease the loading time and size of the Analyzer file.

With each ETM sample recorded the snapshot of the free-running timer is also saved. The period of this timer can be selected in Time Stamp box. Time stamp of the trigger sample is always zero. Samples before the trigger are marked with negative values; samples after the trigger are marked with positive values. If you select a clock cycle, then the CPU cycle counter is recorded.

The resource configuration, triggering and filtering is composed in such a way so it matches the ETM specification as closely as possible, therefore it is recommended to study the 'Embedded Trace Macrocell Specification' document before configuring the trigger and qualifier in the software.



The ETM has a configurable number of event resources. The resource types are:

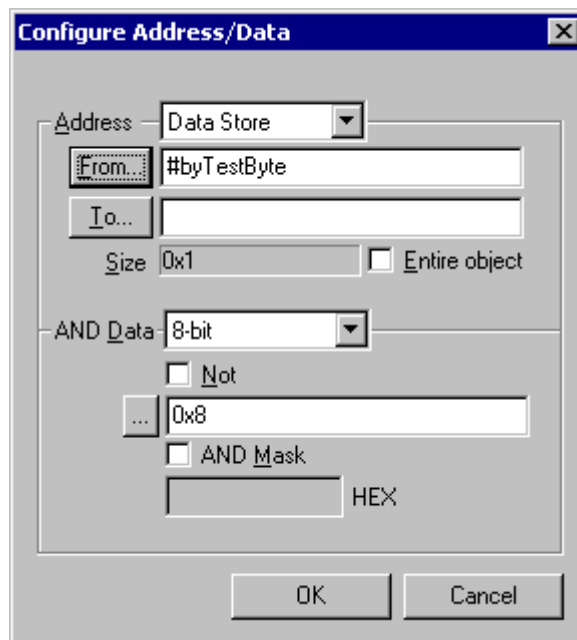
- address comparators
- data comparators
- context ID comparators
- memory map decoders (where supported)
- EmbeddedICE module watchpoint comparators (in cores supporting **RANGEOUT**)
- counters
- a three-state sequencer
- external inputs
- trace start/stop.

The event resources are shared for:

- Trigger
- Trace Enable
- View Data
- FIFO Full

1.1.1 Resource Configuration – Address/Data Configuration

To specify a new comparator, press the ‘New’  button. To modify the existing comparator, double-click it, to remove a comparator, press the ‘Remove’  button.



Address/Data Configuration

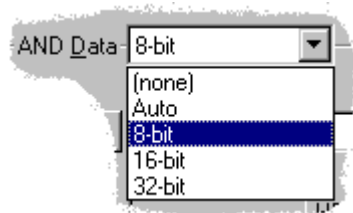
First, the address to monitor is configured. An address can be selected by pressing the ‘From...’ button. If the entire object is to be covered, select the ‘Entire object’ option. If a range is to be configured, select the ‘To...’ address either by entering the address or by selecting the object. The type of access to that address can also be defined.



Data access type

Also, the data loaded or stored to that address can be filtered; in this case the comparator will be active only in the case of this exact data or mask.

To do this, select the length of the data from the drop-down menu.

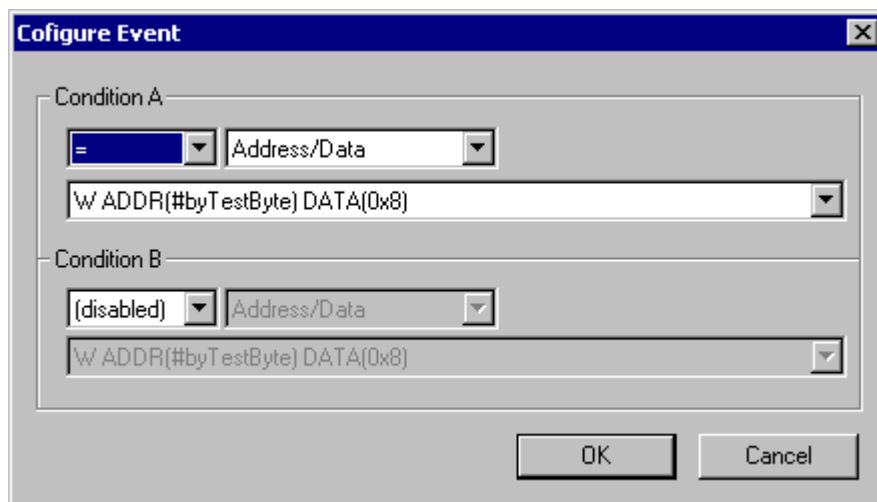


Data length configuration

If the length is set to (none), the data is not monitored.

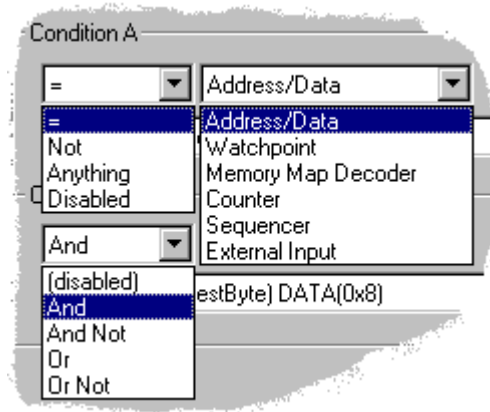
The data value can be inserted, if the comparator should be true only when this value is not in the memory address, select the 'Not' option. Also, the mask can be defined by selecting the 'AND Mask' option and the required mask entered.

1.1.2 Event Configuration



Event Configuration Dialog

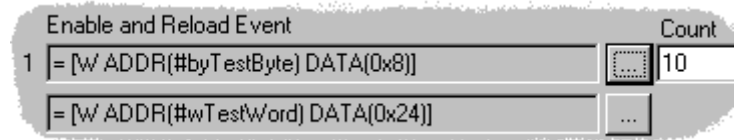
One or two conditions can be defined for every event. The options differ from the condition; the source of the condition is defined for both conditions the same.



Condition options

1.1.3 Resource Configuration – Counters Configuration

Up to four counters can be configured with each having an enable counter and a reload event.



Configuring Counters

Each time an 'Enable' event is reached, the 'Count' counter is decreased. Each time a 'Reload' event is reached, the 'Count' counter is reset to the default value, entered in this dialog. The trigger event can be set any occasion when any counter reaches 0.

In the above example, every write to address TestByte with the value of 0x8 decreases the counter. Every write to address TestWord with the value of 0x24 resets the counter to its default value, 10 in the above example.

1.1.4 Resource Configuration – Sequencer Configuration

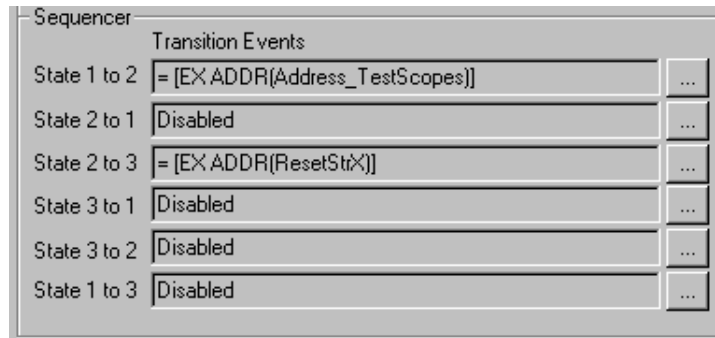
The events that trigger changing states of the tri-state state machine can be defined here.



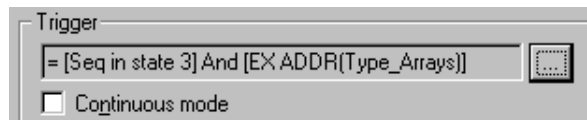
Transition Event Configuration

The trigger can be set to any of the three states.

Example sequencer configuration:



The transition events defined above define state changes.



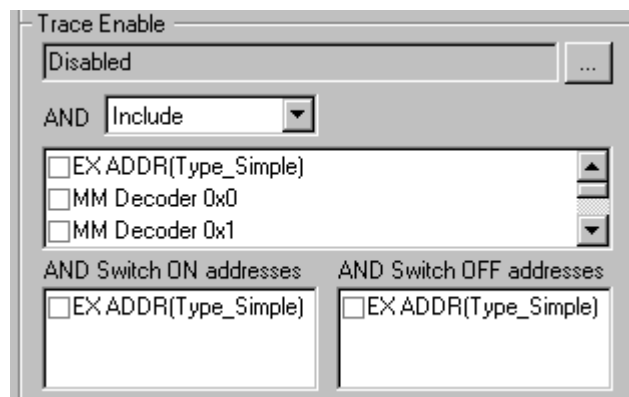
The trigger configuration in this way allows sequential conditions. In this case, first the access to Address_TestScopes must be performed; this switches from state 1 to state 2. Next, the access to ResetStrX switches from state 2 to state 3, which then completes the first trigger condition. Next, the access to Type_Arrays must be performed, in this case the trigger condition is met and saving begins.

1.1.5 Trigger Configuration

The trigger can be set to 'Anything' or to any event configured by pressing the '...' button and opening the event configuration.

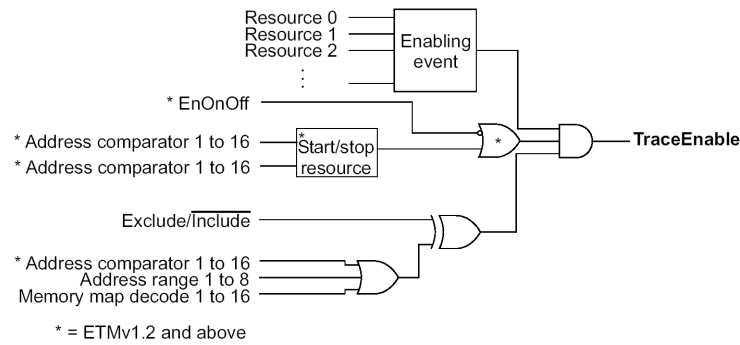
By selecting 'Continuous Mode' the recording continues until the CPU is stopped by the user or if the user has manually selected to stop recording in the trace window.

1.1.6 Trace Enable Configuration



Trace Enable part of ETM Trigger dialog

This field configures which part of instructions will be recorded beside the events enabling trace. The Include/Exclude regions can be defined and addresses which turn recording on and the addresses which turn it off. All conditions must be met (i.e. set to 1) to enable trace.



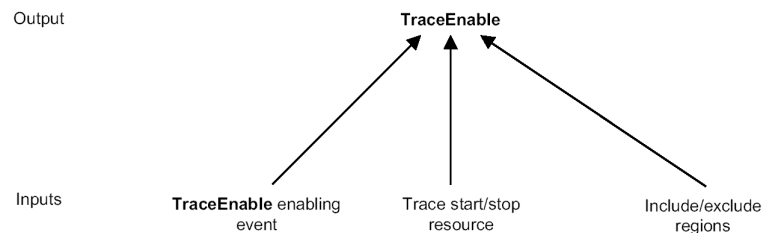
TraceEnable Configuration

The first option enables the Enabling event. If this event is set to 'Disabled', trace will never be enabled (always set to 0). If it is set to 'Anything', this event will always be set to 1.

The second option defines either the include or exclude regions. First, the region type is selected (either Include or Exclude), next the regions can be set. If no region is checked and 'Exclude' is selected, this condition will always be set to 1. Note that if no region is checked and 'Include' is selected, this condition will always be set to 0 and consequently trace will never be enabled.

Start/stop resources are defined in the third part of the Trace Enable configuration, defined as 'Switch ON' and 'Switch OFF' addresses.

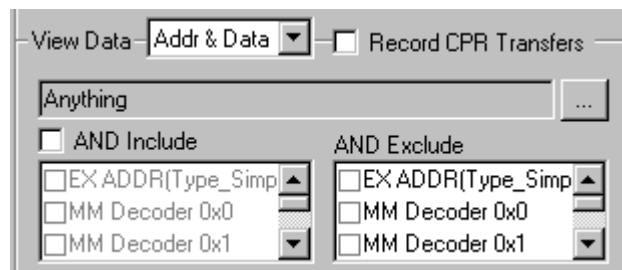
If no resource is selected, the start/stop logic is turned off and always set to 1. If any resource is selected, the logic is enabled and checks for events. When a 'Switch ON' condition is met, at that point the logic will be set to 1. If a 'Switch OFF' condition is met, the logic will be set to 0.



Programming the TraceEnable Logic

Please see the examples for better understanding of Trace Enable configuration.

1.1.7 View Data Configuration

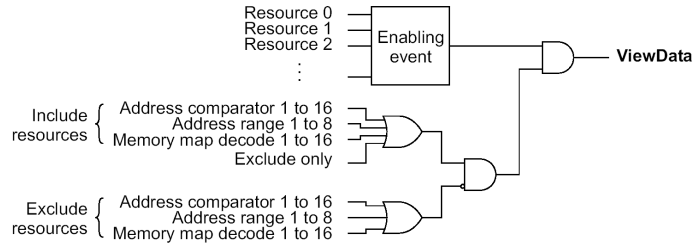


View Data part of Trigger/Qualifier dialog

This field configures which data will be recorded.

In the pull-down box, which data is saved is selected. This option allows you to store addresses, data or both. If this option is set to '(disabled)', no data will be saved. If **'Record CPR Transfers'** is selected, coprocessor transfers will be recorded too.

The event at which data is recorded can be configured. The data regions to exclude and regions to include can be selected too.



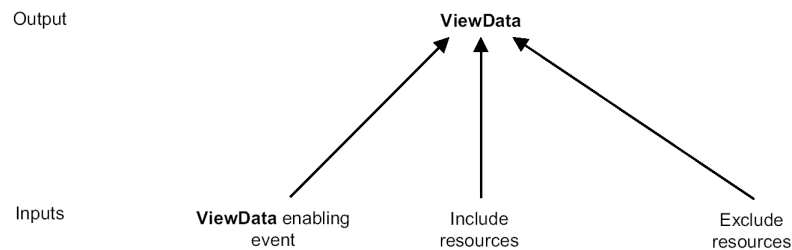
ViewData Configuration

First, the enabling event is selected. If this event is set to 'Disabled', no data will be saved (the event always set to 0). If it is set to 'Anything', this event will always be set to 1.

Next, the Exclude resources can be selected. If any selected resource is true, the event will be set to 0.

Also, the Include resources can be selected. If the checkbox in front of the 'AND Include' is deselected, this logic is turned off and only the exclude logic will be used. If it is selected, this event will be set to 1 only if the selected resource is true. If the 'AND Include' option is checked and no resource is selected, this event will never be set to 1 and therefore no data will be saved.

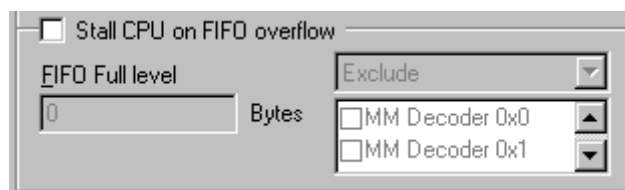
If no exclude or exclude options are to be configured, uncheck the 'AND Include' option and deselect all resources in the 'AND Exclude' box. In this case this event will be set to 1 and only the Enabling event will be considered.



Programming the ViewData Logic

Please see the examples for better understanding of View Data configuration.

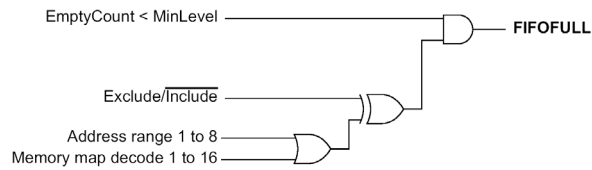
1.1.8 Stall CPU on FIFO Overflow



Stall CPU on FIFO overflow part of Trigger/Qualifier dialog

If this option is enabled, the system stalls the CPU when the FIFO overflows. The **'FIFO Full Level'** specifies the number of free bytes of the FIFO when the level of the FIFO is considered full. The memory regions in

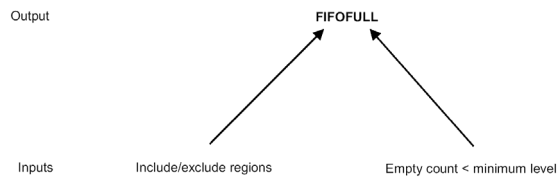
which CPU stalling is enabled can be selected by using the Include/Exclude pull-down and selecting either include or exclude regions.



FIFOFULL Generation

Either include or exclude regions can be specified. If include regions are specified ('Include' is selected in the drop-down box), the FIFO buffer level will only be checked in these regions. If exclude regions are specified ('Exclude' is selected in the drop-down box), the FIFO buffer will be checked in all regions, except in the specified ones.

If FIFO Full level is being checked, first the number of free bytes in the FIFO is checked. If the number is below the specified number of bytes, the program will check whether the current memory area lies in the Include region or outside the Exclude regions. If this is true, then the CPU will be stalled.



Programming the FIFOFULL Logic

Typically this option remains unchecked.

1.2 Troubleshooting Scenarios

1.2.1 Record everything

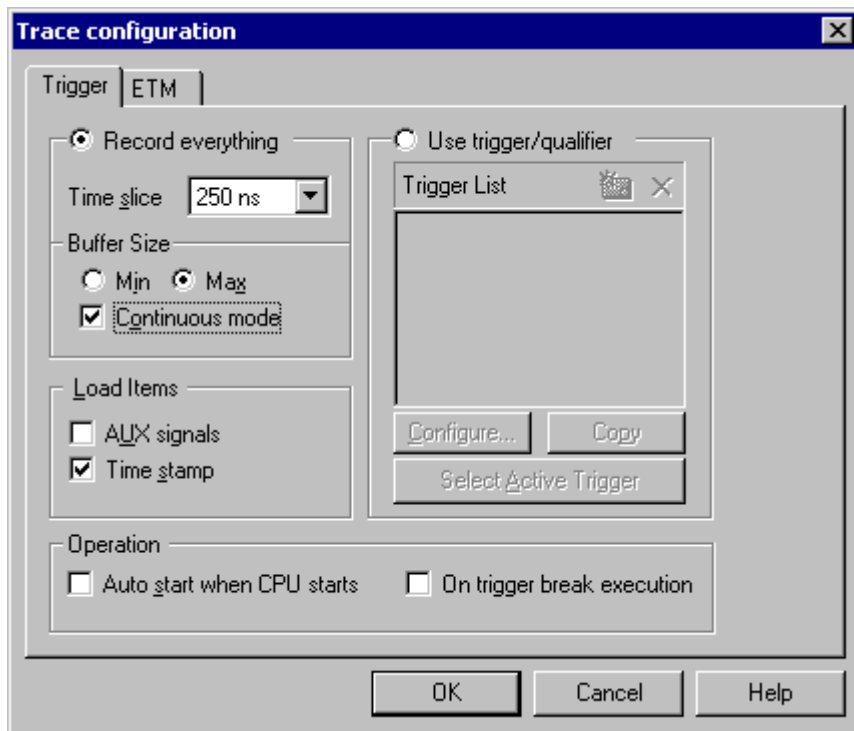
This configuration is used to record the contiguous program flow either from the program start on or up to the moment, when the program stops.

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 : ETM will record the CPU execution (instructions and data accesses) until the CPU is stopped by either the user or the program itself in case of any problems in the application. From the history, any problem originating from the code or the target can be filtered out.

Select the 'Record everything' mode in the 'Trigger List' dialog. Set buffer size to maximum to achieve the best results and check the 'Continuous mode' option.



Before the program is set to run or while it is running already, activate ETM recording via 'Trace begin' tool bar or shortcut key. The ETM stops recording when the program execution is stopped. After the ETM stops recording, the collected information is analyzed and displayed.

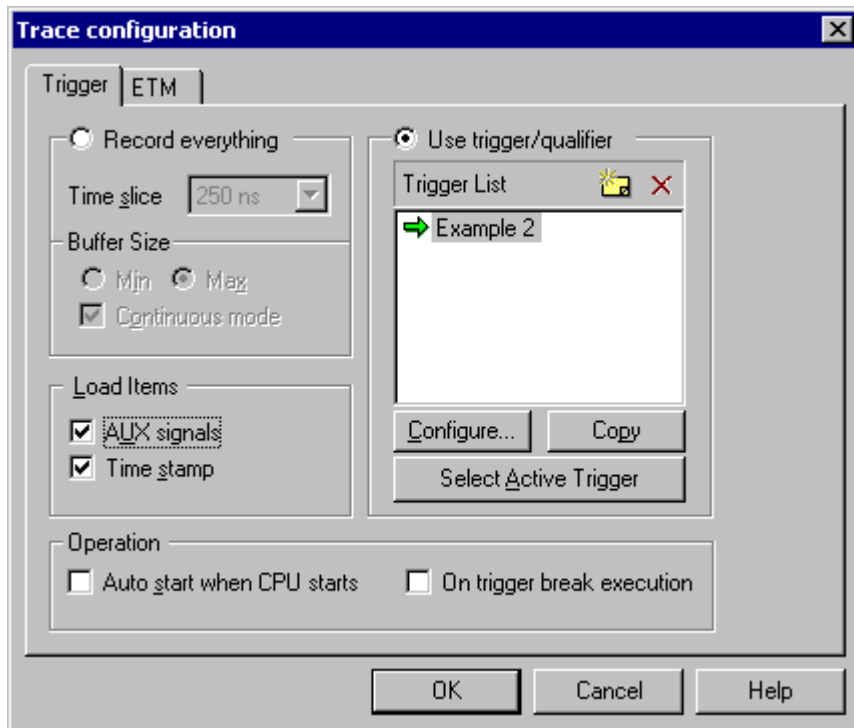
1.2.2 Plain Trigger Configurations

This section describes configuring trace to trigger on a specific function being executed or to record specific variable data accesses.

‘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.

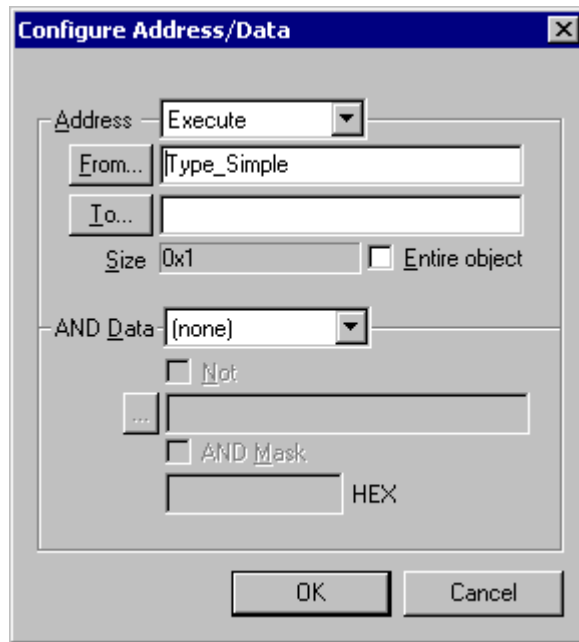
Example: *ETM will trigger on a function `Type_Simple` execution and record instructions.*

Select the ‘Use trigger/qualifier’ mode in the ‘Trigger List’ dialog and configure a new trigger called ‘Example 2’.



Configure the trigger by invoking the ‘ETM Trigger dialog’.

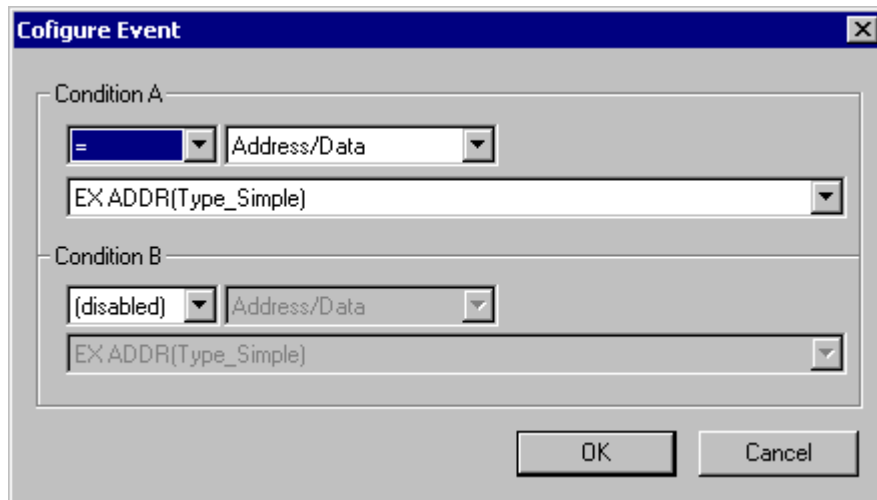
Enter `Type_Simple` address in the address field by invoking the symbol browser by pressing the ‘Add’ button in the dialog.



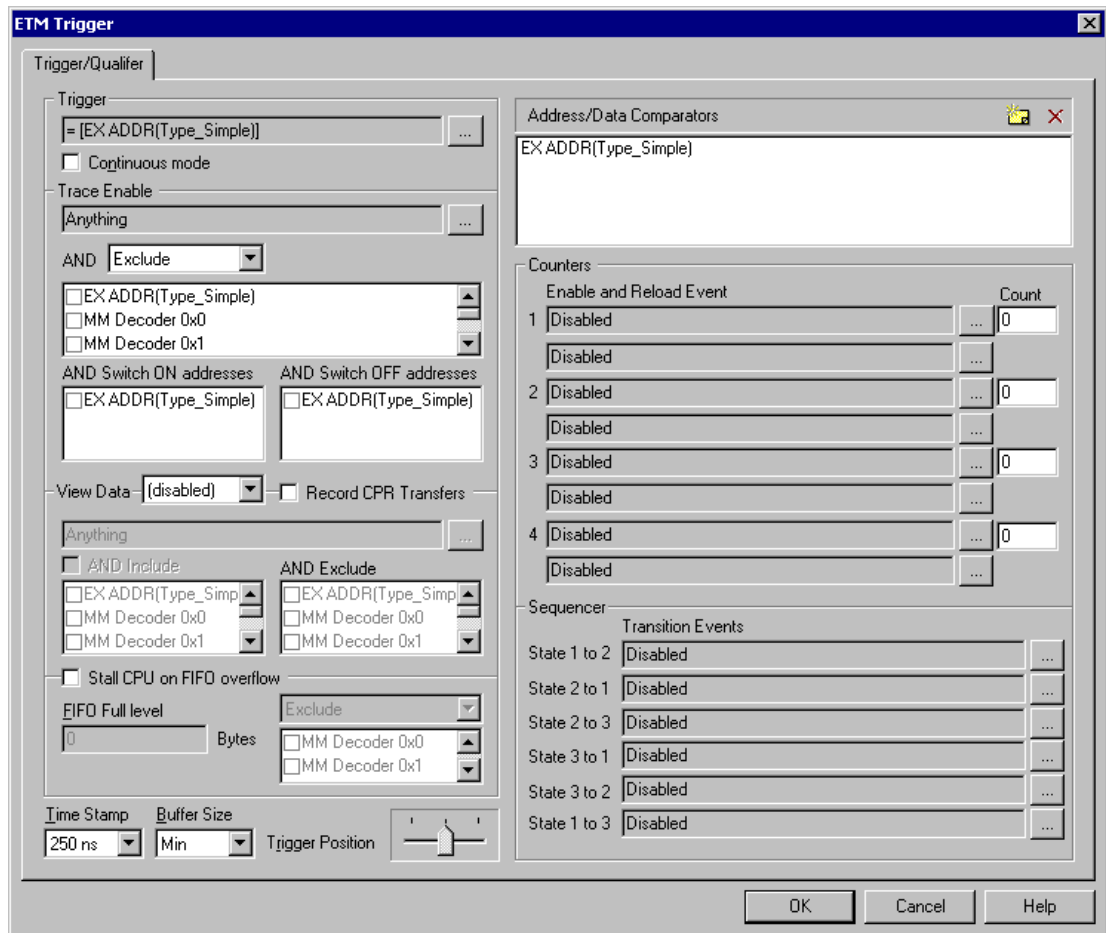
Adding the address

Set the 'Execute' access type.

Now, return to the Trigger/Qualifier menu and specify the trigger event.



Define the condition A, so that it equals to the defined address.

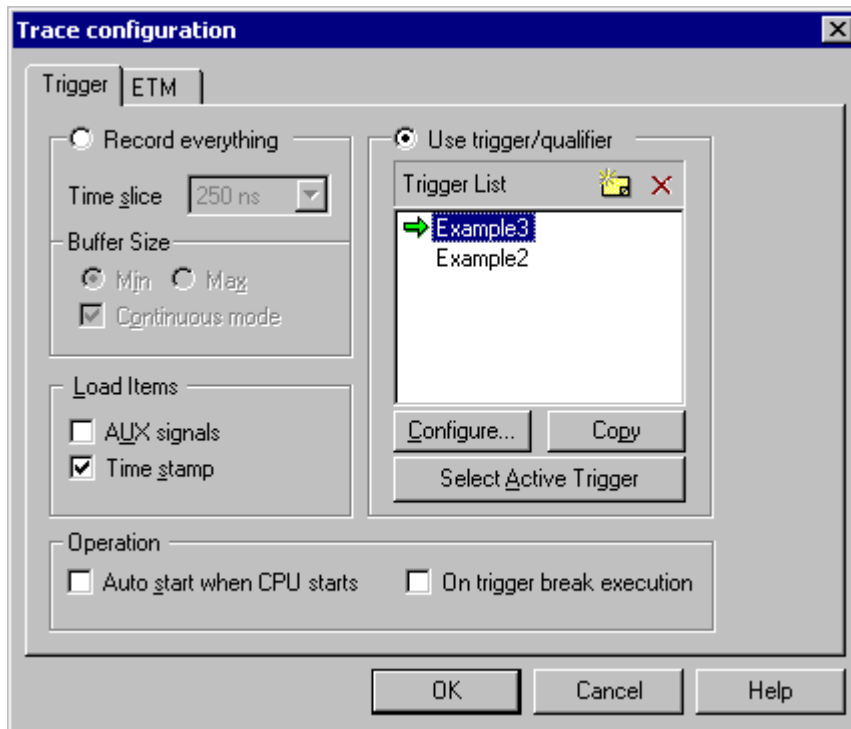


Final configuration of the Trigger/Qualifier

After the trace is being activated, the ETM starts recording after the `Type_Simple` is executed. After the iTRACE buffer is fulfilled, the results are displayed.

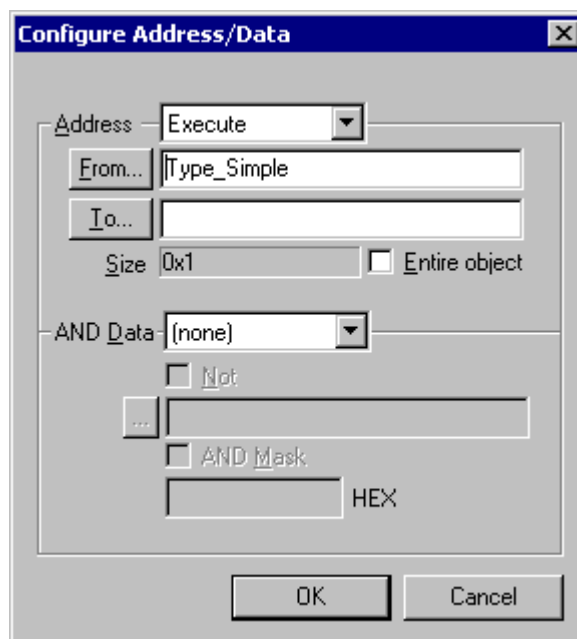
Example: ETM will start recording (instructions and data accesses) after the function `Type_Simple` is executed (called) for the fifth time.

Select the 'Use trigger /qualifier' mode in the 'Trigger List' dialog and configure new trigger called 'Example 3'.



Configure the trigger by invoking the 'ETM Trigger dialog'.

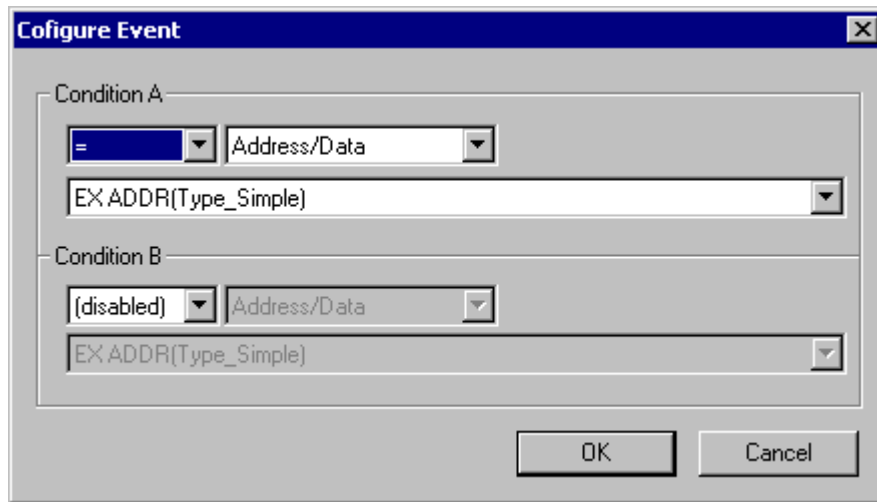
Enter `Type_Simple` address in the address field by invoking the symbol browser by pressing the 'Add' button in the dialog.



Adding the address

Set the 'Execute' access type.

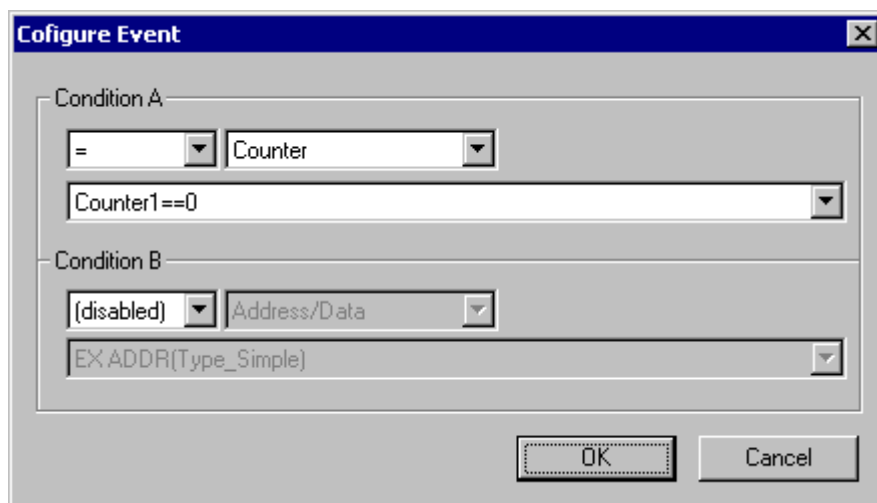
Now, return to the Trigger/Qualifier menu and enable the counter by specifying the counter event.



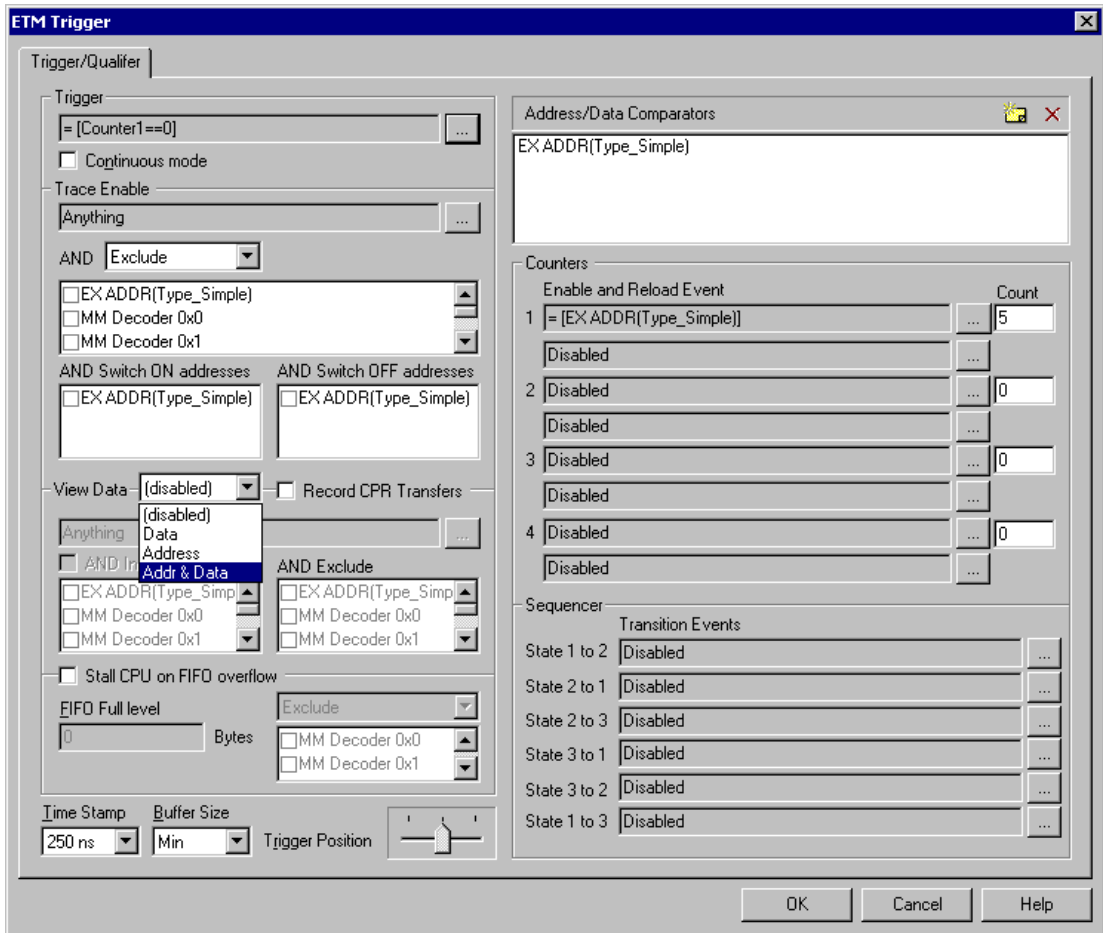
Define the condition A, so that it equals to the defined address.

Now, set the 'Count' to 5 since we want to focus on fifth execution of the function.

Next, set the trigger event with the Event Configuration menu by pressing the '...' button. Set the condition so that it equals to the Counter1 being 0.



Next, back in the main trigger/qualifier window define the View Data field to 'Addr & Data' – this will save both instruction and data accesses.

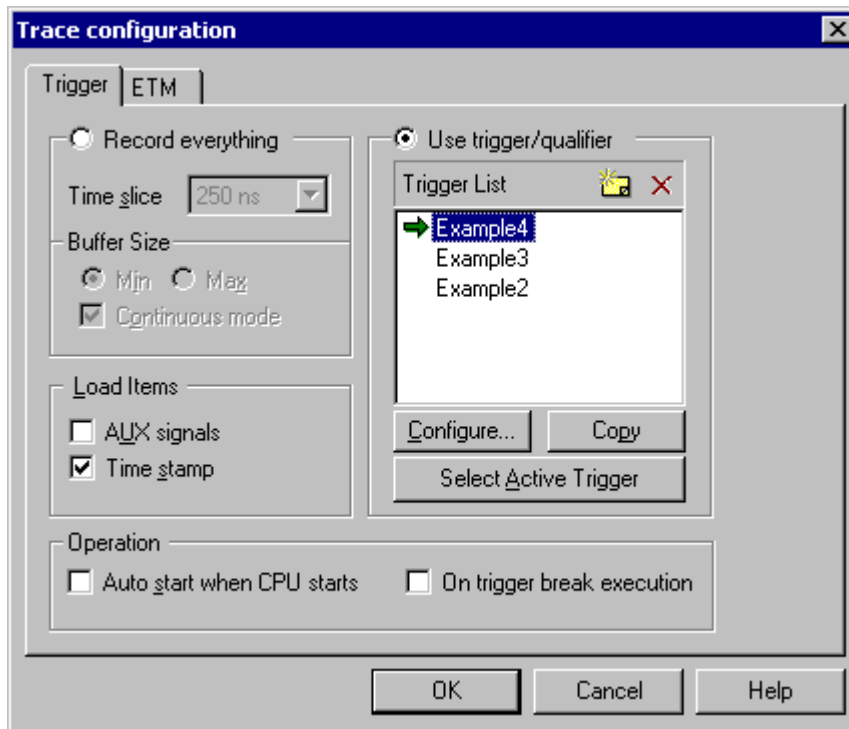


After the trace is being activated, the ETM starts recording after the `Type_Simple` is being executed for the fifth time. After the iTRACE buffer is fulfilled, the results are displayed. Data accesses (both address and data bus) will be recorded beside the default instructions.

However, the user can also check the 'Continuous mode' option in the 'Trigger/Qualifier' dialog. Then again, the trace would not stop recording before the program execution is being stopped. Typically, this is not what the user needs and 'Continuous mode' option is left unchecked.

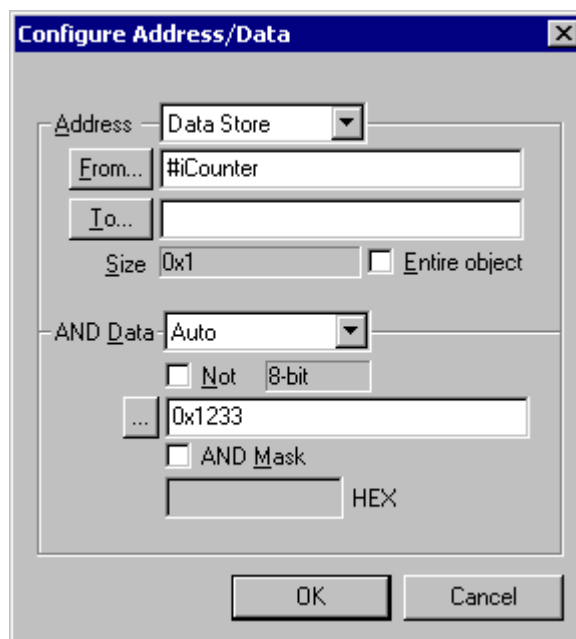
Example: *In this example, an 'illegal' write to an iCounter variable occurs while running the target application. The variable gets a value 0x1233, which is unexpected and the application ceases from operating correctly.*

Select the 'Use trigger /qualifier' mode in the 'Trigger List' dialog and configure new trigger called 'Example 4'.

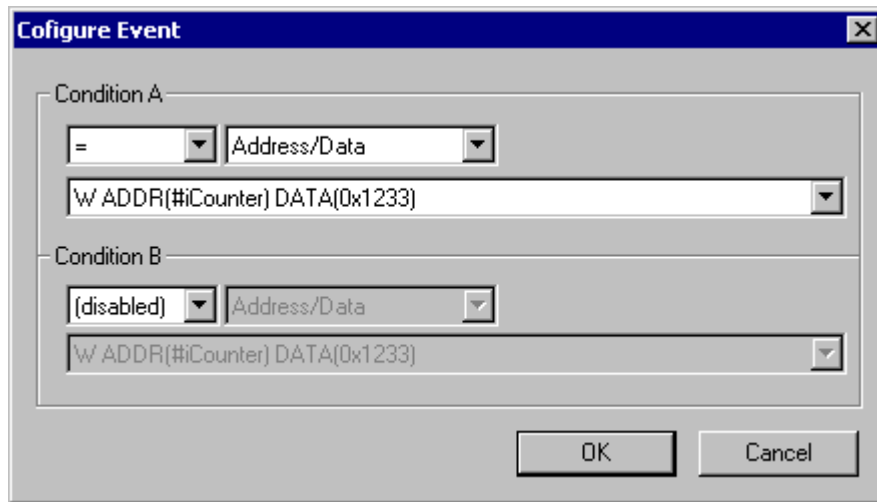


Configure the trigger by invoking the 'ETM Trigger dialog'.

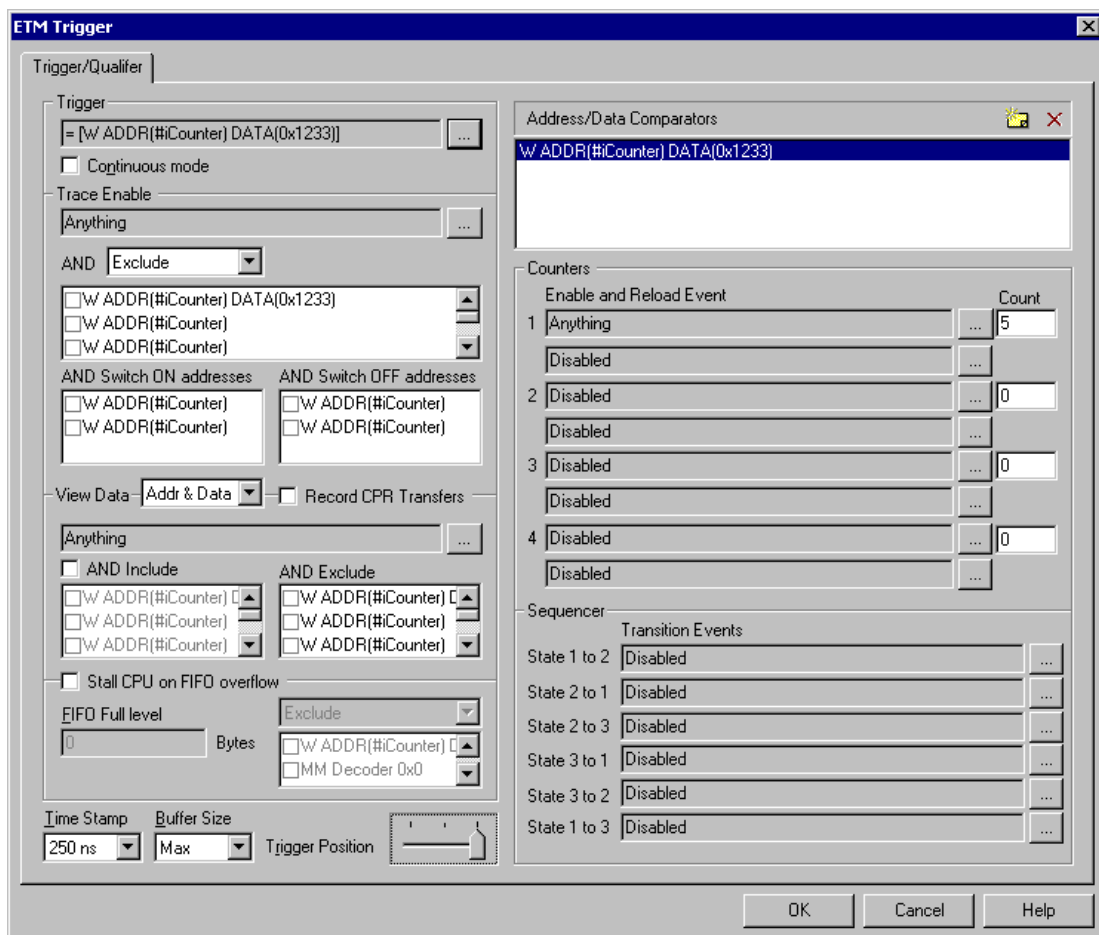
First, define the Address/Data comparator. Select iCounter from the symbol browser. Next select 'Data store' for access type, 'Auto' for data size and enter **0x1233** in the value field.



Next, set the trigger to the newly defined event.



Next, enable the recording of data cycles. In this particular case, set the trigger position to 'End' and buffer size to 'Max' to get the best results from the iTRACE.

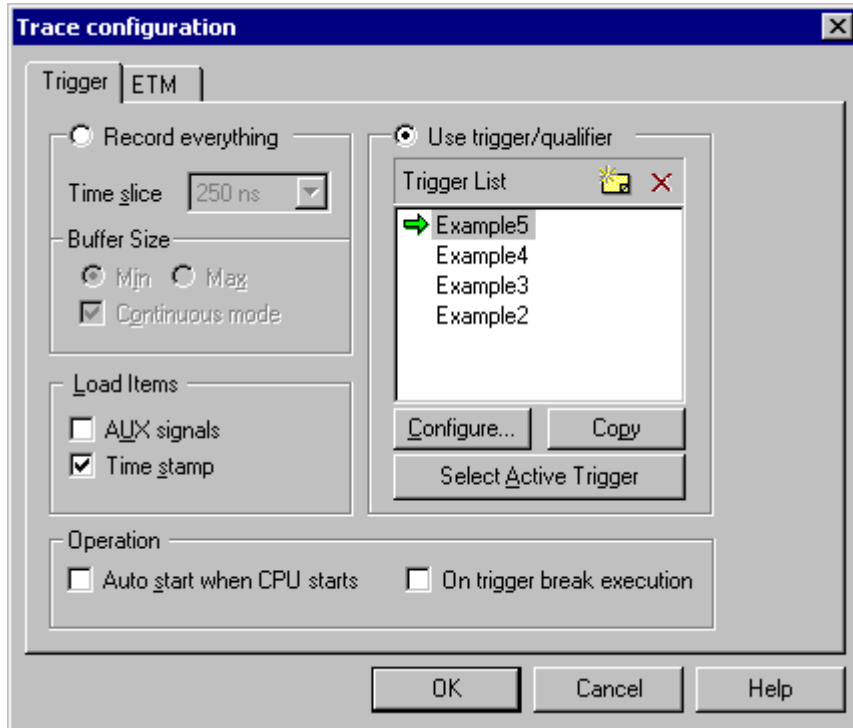


Final configuration

When incorrect write to the iCounter occurs, the iTRACE stops recording after the buffer is fulfilled. Since the trigger was set to the end of the buffer, the user can inspect from the buffer pre-history of the application's behavior before the trigger event occurred. Note that trigger event is troublesome in this particular case.

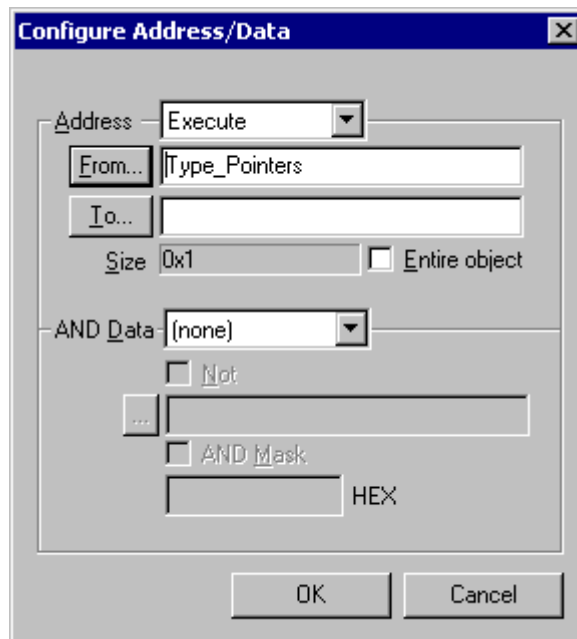
Example: *Type_Pointers* is a function, which is called periodically by the application. Using on-chip ETM, the time between the consecutive calls will be measured.

Select the 'Use trigger /qualifier' mode in the 'Trigger List' dialog and configure new trigger called 'Example 5'.



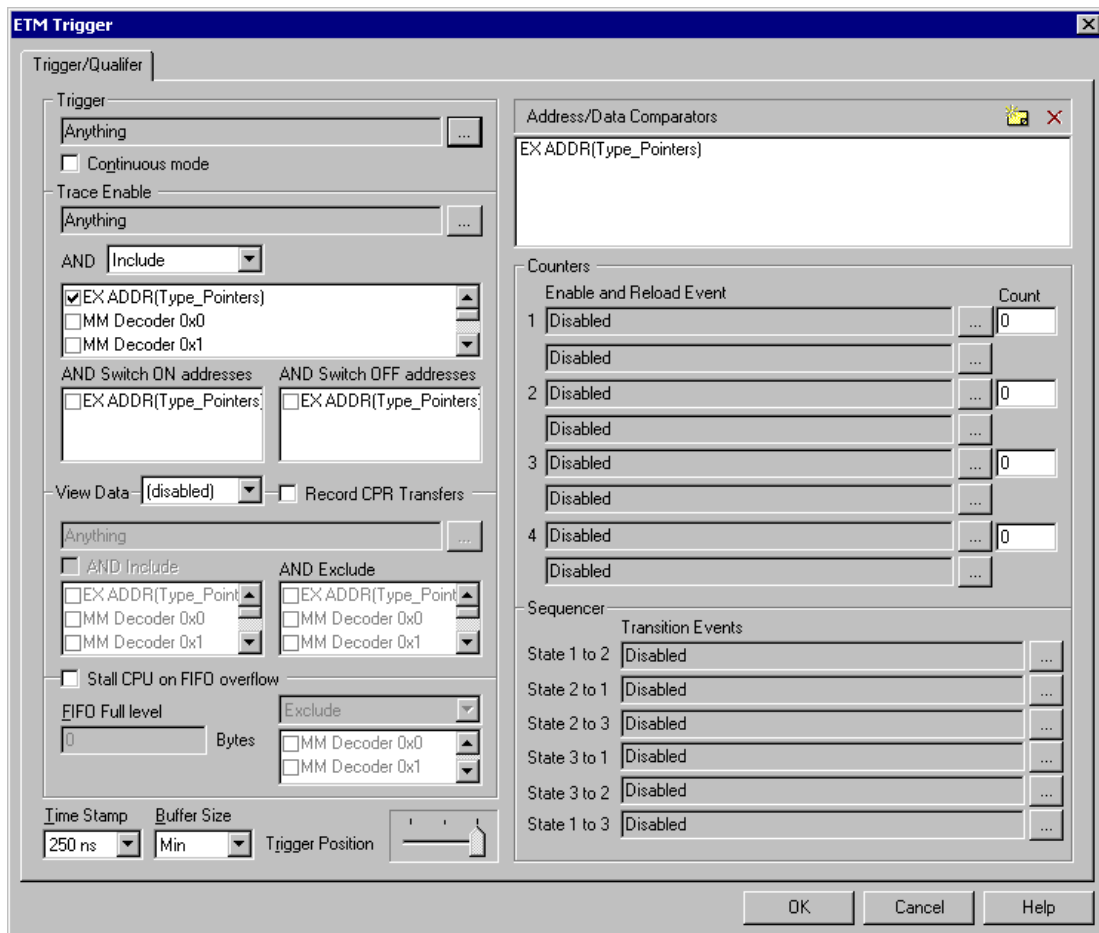
Configure the trigger by invoking the 'ETM Trigger dialog'.

First, define the Address/Data comparator. Select *TypePointers* from the symbol browser and 'Execute' for access type. If the whole function must be recorded instead of an entry point only, check 'Entire object' additionally.



Revert to the Trigger/Qualifier dialog. The trigger is set to ‘Anything’ since we are interested only in time differences between executed function calls. The start event is irrelevant.

Next, select the ‘Include’ and ‘Type_Pointers’ event in the Trace Enable field.



Final configuration

After activating ETM, the iTRACE records every entry to the function `Type_Pointers`. By using ‘Relative time’ display type in the trace window the user can verify the time between any two consecutive function calls.

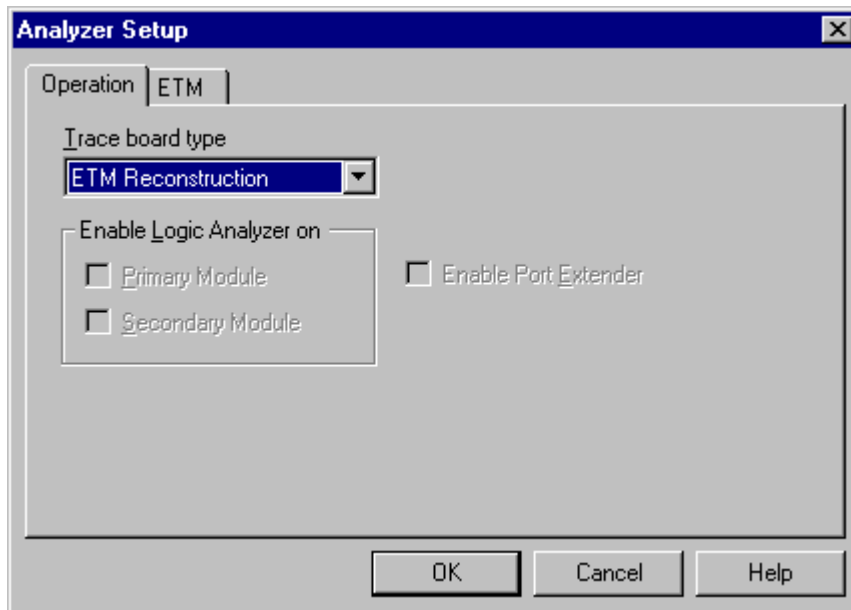
The analyzer module is an optional add-on to a debug system. With its aid most of the emulated CPU lines are recorded without interfering with its operation. This makes the analyzer module and its features a vital component in monitoring the program’s response to real-time events.

1.2.3 Advanced Trigger

iTRACE PRO offers some advanced trace features which are restricted to the instruction execution activity:

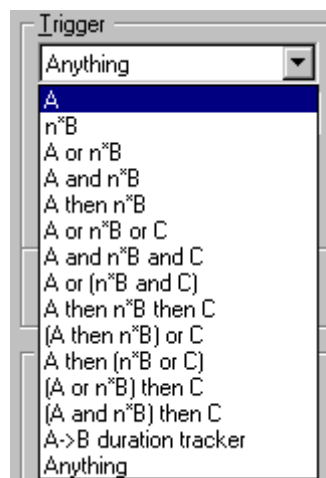
- 3-Level Trigger
- Qualifier
- Watchdog Trigger
- Duration Tracker

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



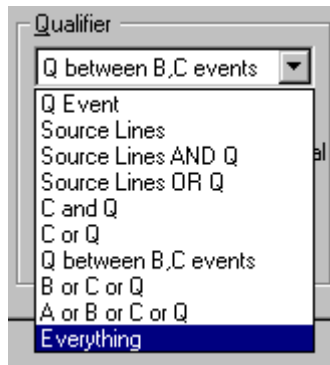
3-Level Trigger

On-chip Nexus resources don't support two or more level triggers, which might be a showstopper sometimes. iTRACE PRO 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.

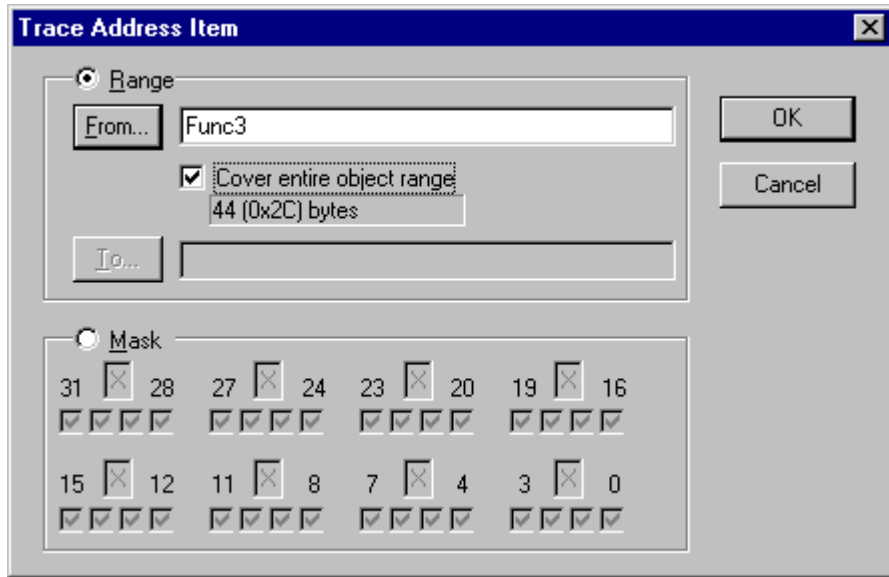


Thereby, the qualifier can be configured in a standard way and then additionally up to 8 CPU cycles can be recorded before and/or after the qualifier. For instance, this allows recording of a function or just its entry point and few instructions recorded before make possible to determine, which code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

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

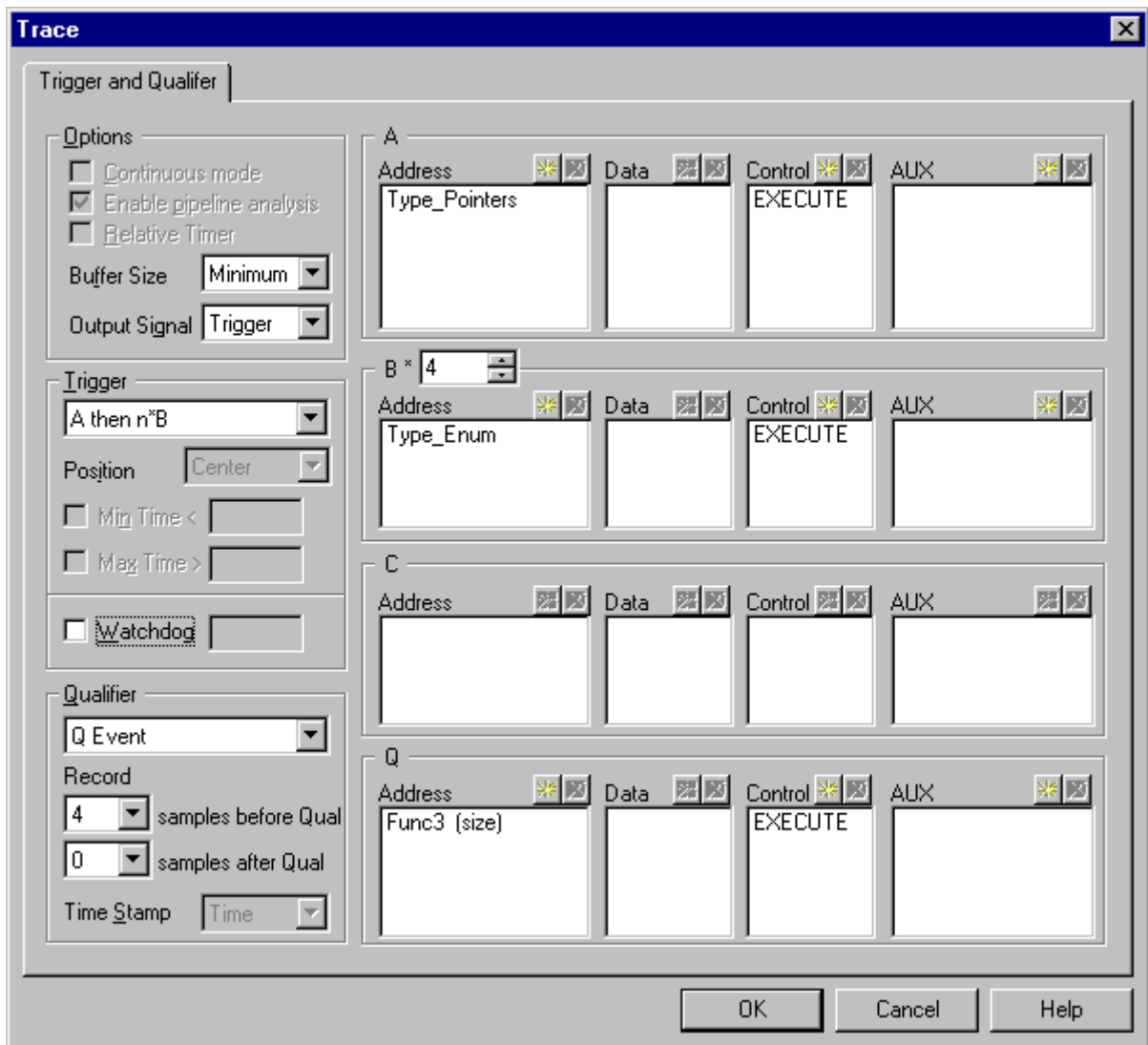
Example: Let's record Func3 execution after the Type_Pointers function is executed and then 4-times Type_Enum function is called.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger 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=&y; 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 stwu 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

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

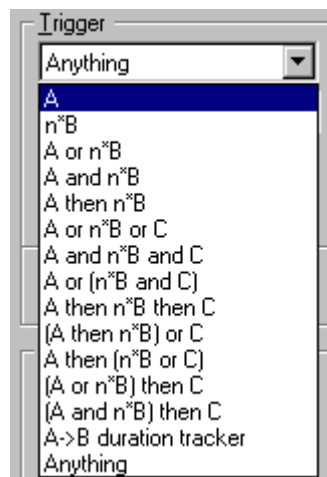
The user needs to enter the trace watchdog time-out period and define the “trace watchdog reset” condition, which can be logically combined from events A, B and C.

- 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

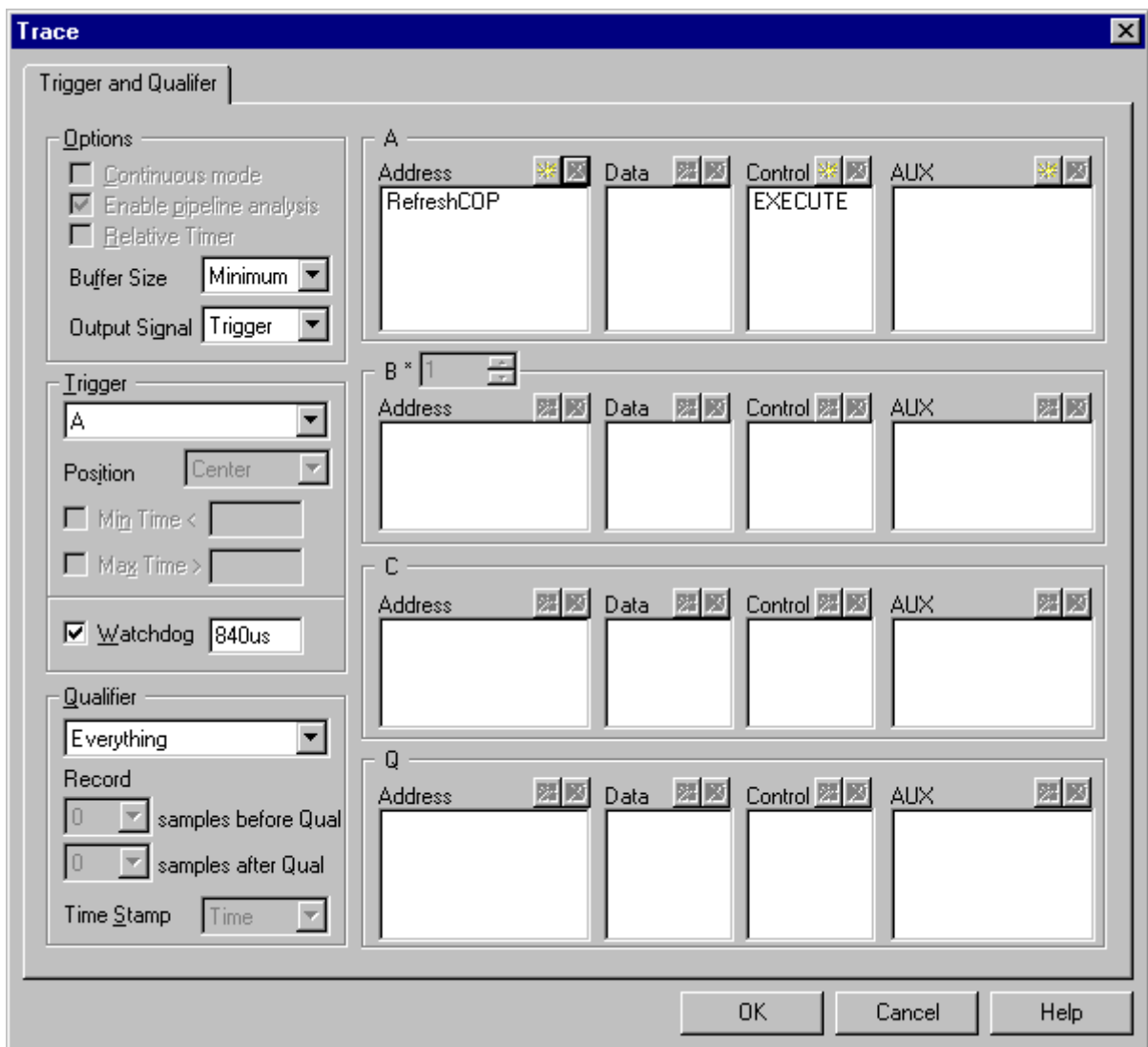
Example: Target application features on-chip COP watchdog, which enables the user to check that a program is running and sequencing properly. When the COP is being used, software is responsible for keeping a free running watchdog timer from timing out. If the watchdog timer times out it’s an indication 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 μs in this particular example. It may vary between the applications since it’s configurable. The watchdog timer is reset within 800 μs 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.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger 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.

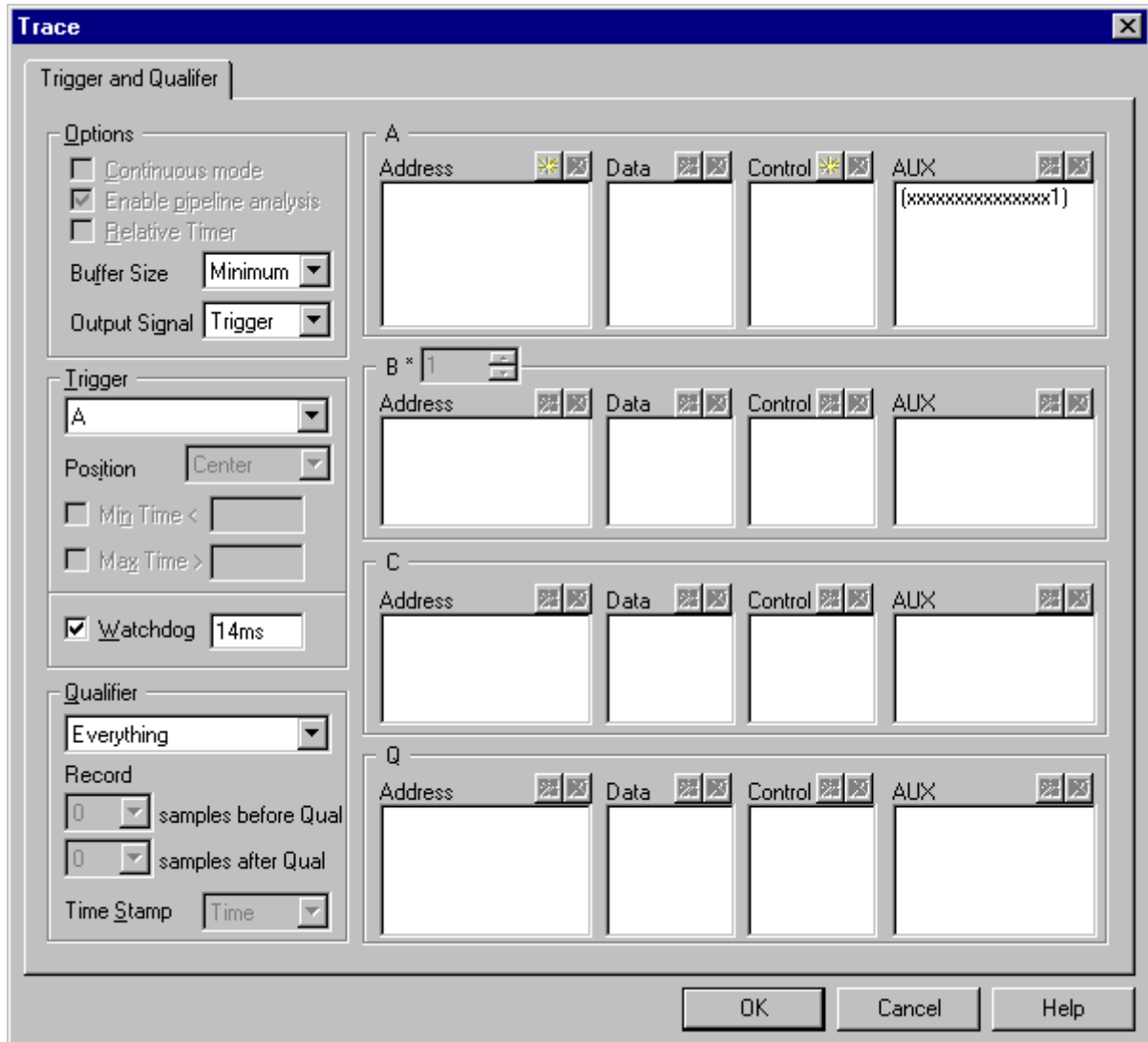
Example: The application features (external) target watchdog timer, which is normally periodically reset every 15 ms by the WDT_RESET target signal.

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.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger 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.

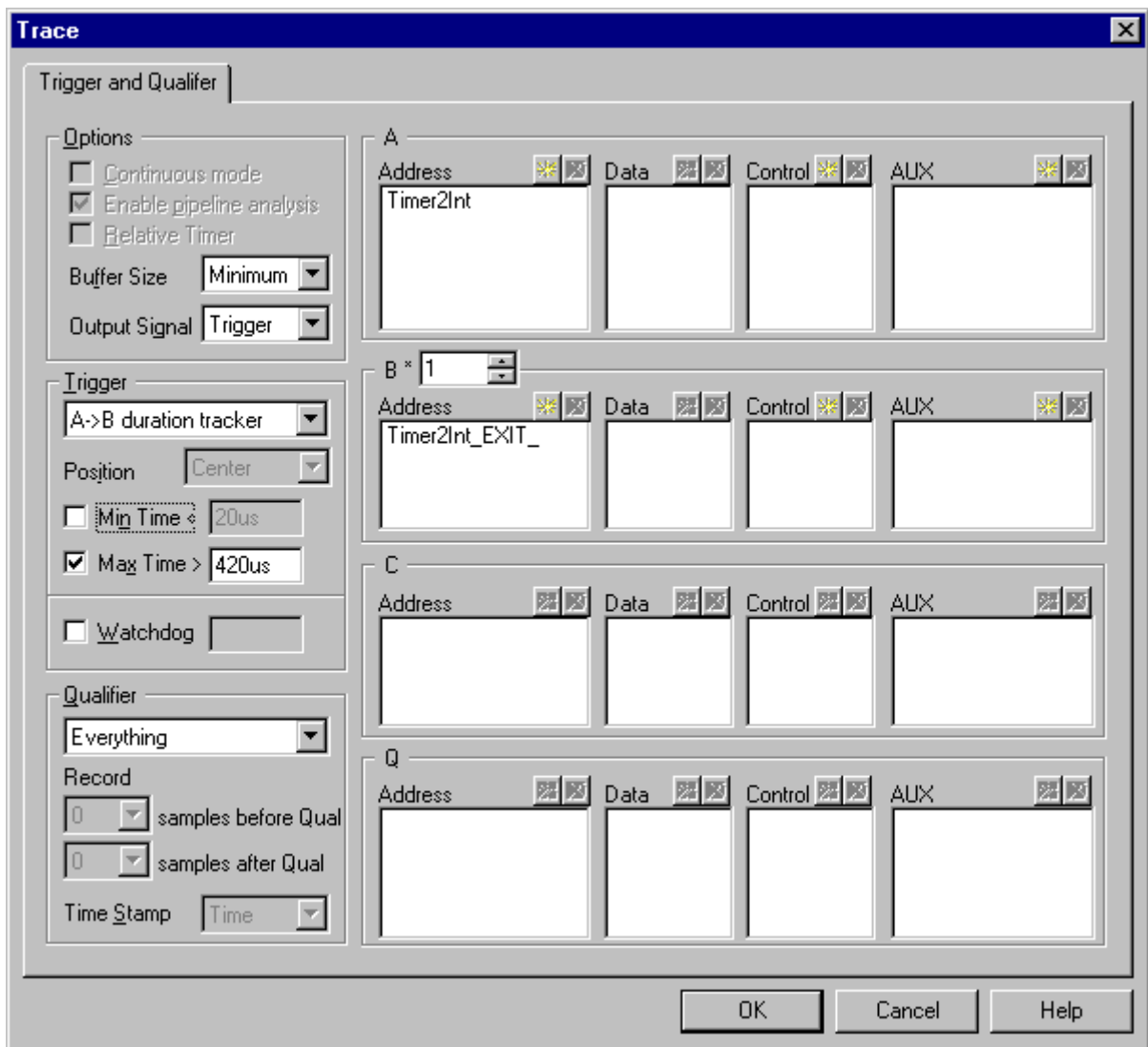
Min Time is compared with the Current Time as soon as the event A is detected or simply, Current Time is compared against Min Time as soon as the program enters the object being tracked.

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 μ s under normal conditions. The user wants to trigger and break the program execution when the `Timer2Int` interrupt routine executes longer than 420 μ s, which represent abnormal behaviour of the application.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Select maximum buffer size and position the trigger at the end of the buffer.
- 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 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
0	0000035C	Type_Enum(); 0000035C 48000551 hl 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.

2 BackTrace

BackTrace is a feature of winIDEA which enables a developer to get a high level view of the history of program execution recorded in a trace recording. Looking at a trace recording only, one can only see CPU machine instruction and data samples in the order in which they were recorded by the trace tool. Seeing the execution path of the program is fairly straightforward just by looking at the sequence of samples in the recording. But what about the state of the program at specific points in program history? Just by looking at the recording, this is quite unimaginable.

This is where BackTrace comes in. After a trace recording has been made, the trace stream is displayed in the trace window. The program can continue running or can be stopped, does not really matter for BackTrace. Now, you can select any instruction sample in the recorded trace stream and use "Set Context" command from the menu which pops up when clicking the right mouse button on a sample in trace window. winIDEA will switch display to the BackTrace mode and the disassembly, memory, variables and watch windows will display the state of the program at the time just before the instruction in the selected trace sample was executed. Registers pane will show contents of registers at that point in time, memory windows will show contents of memory at that point in time, variables and watch windows will show values of variables at that point in time. Real-time watch windows do not apply in BackTrace mode. There will be cases where it is not possible for BackTrace to determine the state of certain registers, bits in a register or memory locations. In such cases, the affected entity will be displayed with questionmark instead of the value.

Once in BackTrace display mode, one can keep selecting different samples in the trace stream and using the "Set Context" command to examine the state of the system at the point of the selected sample. One can also follow the changing of the system state one instruction at a time, forward or backward in time by using F4(forward) and Shift+F4(backward). F4 and Shift+F4 move one assembly line at a time or one source code line at a time, depending on the type of the trace line that was last selected by mouse.

Registers can not be modified in BackTrace mode, since BackTrace displays historical register context. Memory image can not be modified in BackTrace mode as well.

To switch back to the live debug session display, use any debug command and the display will switch back to the live debug session.

If target is running, switching back to live debug session will not be obvious, because registers and memory can not be refreshed while target is running. If the target is stopped, 'Debug/Snapshot (F8)' or 'Debug/Show execution point (Alt+Num*)' will do just fine.

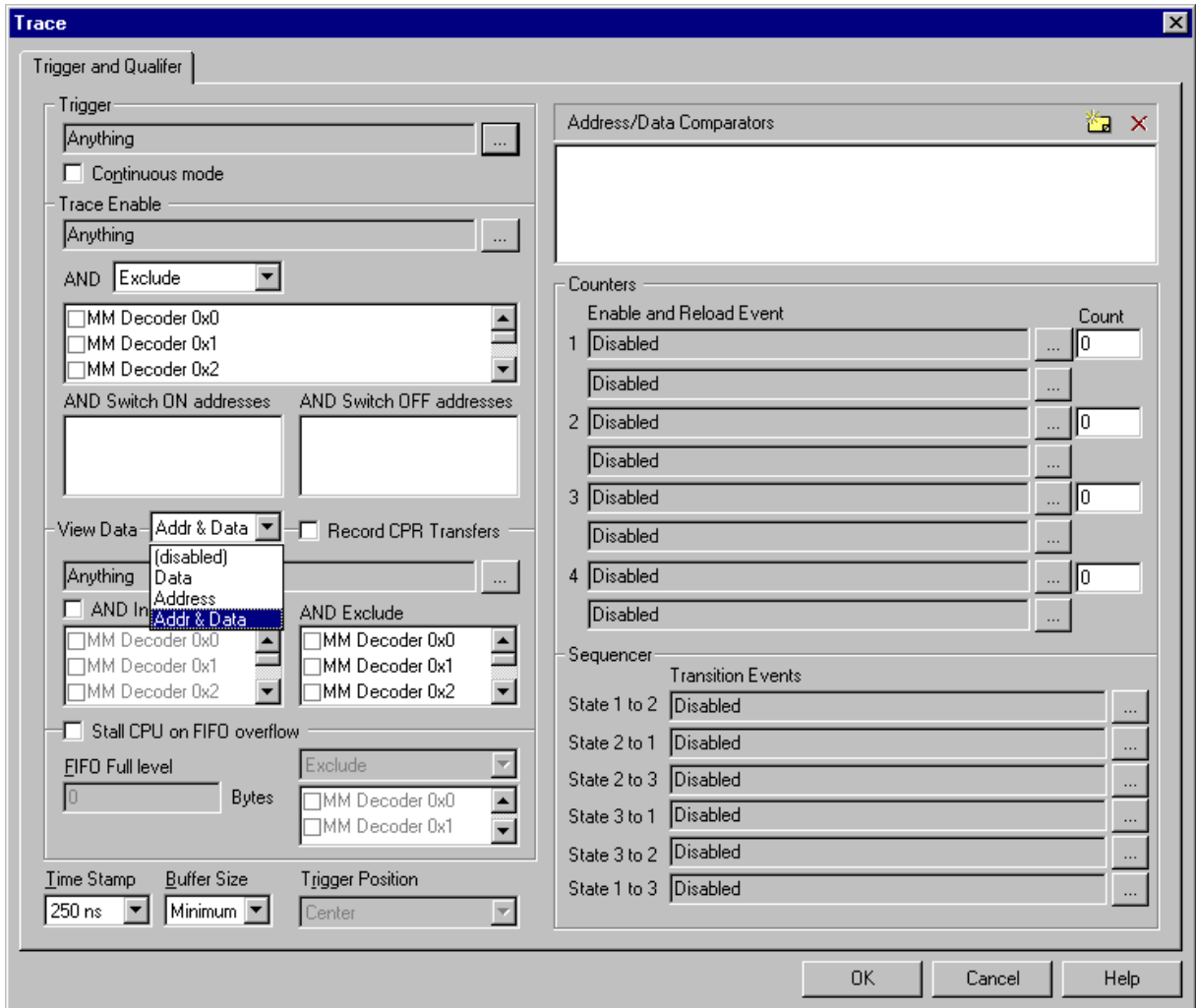
BackTrace works on in-circuit bus trace, ETM and Nexus trace recordings. Quality of BackTrace results also depends on presence of stream errors in the trace stream. Quality is measured in terms of the amount of reconstructed information (register contents and memory locations). All displayed reconstructed information is correct, what could not be reconstructed is displayed as '?'.

In case of in-circuit emulator with entirely visible program and data bus the best results are generally expected, since the trace recording contains all bus activity and there are no errors. In case of ETM, trace recording can either be free of errors or it can contain errors (i.e. FIFO overflows), depending on the CPU frequency, trace port throughput and data trace configuration. When there are no errors in the recorded trace stream the quality of BackTrace results will be the same as in case of bus trace. When there are errors in the stream, the amount of reconstructed information will be affected by the number of errors and will vary depending on how far the point of context reconstruction is from the point of stream error. In case of Nexus trace, the situation is similar as in case of ETM trace. There can be errors in the stream which affect the quality of BackTrace results.

Configuring the trace

Fundamental prerequisite for good quality BackTrace results is the data trace. BackTrace yields worst results if the on-chip trace doesn't feature the data trace or if it is not configured properly for data access recording. Below is an example of ETM Trigger and Qualifier configuration, which results in recorded program flow and data accesses. Trigger is set to anything and all program and data cycles are recorded. To minimize the possibility for overflows, the user may select only data or address to be broadcasted for the data messages but this will also have a negative impact on amount of information displayed by the BackTrace, which relies solely on the

information recorded by the trace. Any trace trigger can be configured without impacting BackTrace results.



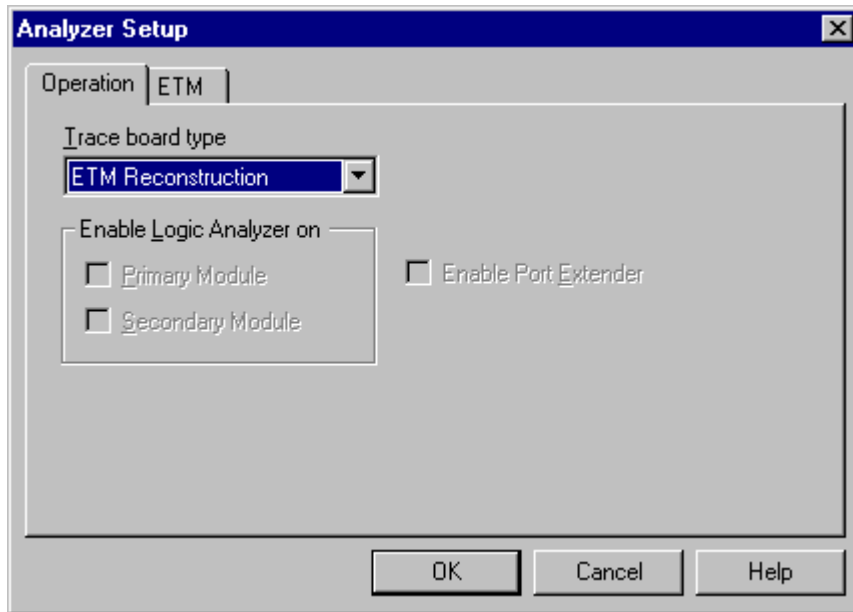
ETM Trigger and Qualifier

3 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.

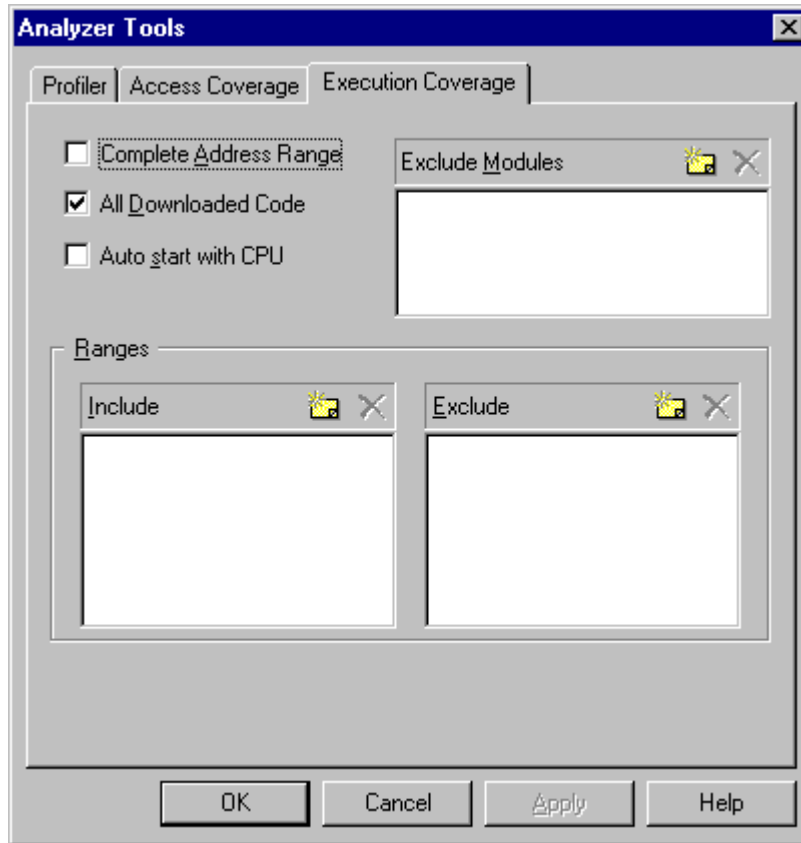
Execution coverage can cover 4MB (iTRACE PRO) CPU address space, which should cover most of existing ARM applications. It can run infinite time, which means in practice that the application can run for days and then the execution coverage results can be analyzed.

Select ‘ETM Reconstruction’ 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.

Refer to software user’s guide for more details on configuring Execution Coverage and its use.



Execution Coverage is configured. Reset the application, start Execution Coverage and then run the application. When it's assumed that the complete application code was executed, stop the Execution Coverage and inspect the results.

	Lines Graph	Lines	Sizes Graph	Sizes
Modules		433/781 (55%)		F00/1FB0 (47%)
+ crt0.s		43/45 (96%)		1AC/B4 (96%)
+ CPUtest.c		3/20 (15%)		30/158 (14%)
+ main.c		2/34 (6%)		18/148 (7%)
+ long main()		2/20 (10%)		18/78 (20%)
+ {		1/1 (100%)		14/14 (100%)
+ {		1/1 (100%)		4/4 (100%)
+ test.c		10/177 (6%)		150/B84 (11%)
+ void Type_Simple()		1/31 (3%)		24/1D0 (8%)
+ d=c;		1/1 (100%)		24/5C (39%)
+ 00000454 - 00000477				
+ void Address_TestScopes()		1/19 (5%)		10/114 (6%)
+ ++X;		1/1 (100%)		10/10 (100%)
+ float Func4(float, unsigned)		8/8 (100%)		11C/11C (100%)
+ {		1/1 (100%)		24/24 (100%)
+ float fRet=(float)0.0;		1/1 (100%)		8/8 (100%)
+ for (i=0;i<5;++i)		1/1 (100%)		14/14 (100%)
+ for (i=0;i<5;++i)		1/1 (100%)		10/10 (100%)
+ *(pC+i)=0xA+i;		1/1 (100%)		1C/1C (100%)
+ fRet+=f+(float)*(pC+i)+(f		1/1 (100%)		88/88 (100%)
+ return fRet;		1/1 (100%)		8/8 (100%)
+ }		1/1 (100%)		20/20 (100%)
+ fp-bit.c		148/219 (68%)		484/674 (70%)
+ dp-bit.c		191/250 (76%)		778/9A4 (77%)
+ libgcc2.c		36/36 (100%)		C0/C0 (100%)

Execution Coverage results

4 Profiler

Profiler records executed function entry and exit points and then run time-analysis over the collected information. As a result it gives details on how much time (minimum, maximum, average) has the CPU spent in the particular function. Available information allows the user to optimize those parts of code, which are most time consuming or time critical.

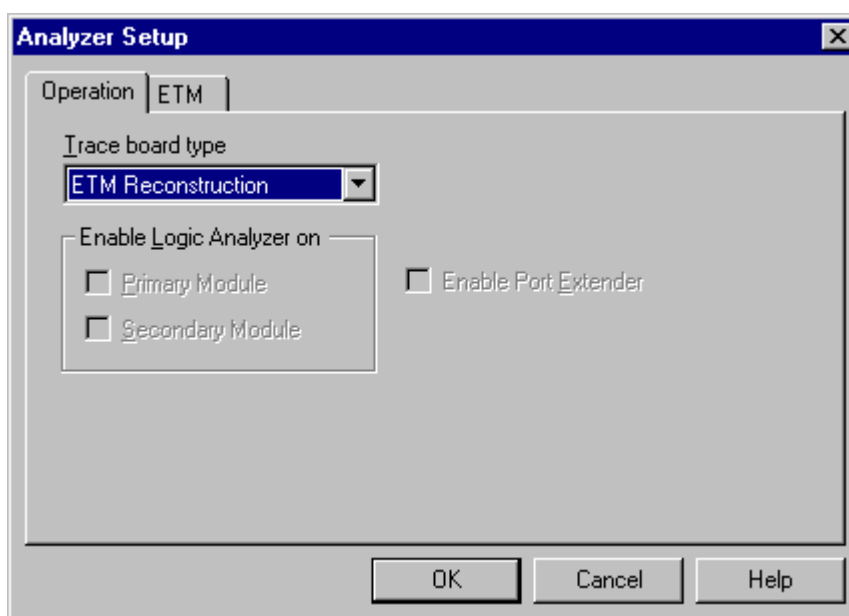
The debug download file must contain accurate debug information when using Profiler to analyze C/C++ application. Normally Profiler extracts all the necessary information from the debug information and becomes useless if configured for wrong function entry and exit points.

Development system offers two types of Profiler:

- Offline Profiler, based on the ETM trace record
- Online Profiler, based on iSYSTEM proprietary ETM reconstruction and yields several times longer profiler session comparing to the offline profiler.

Offline Profiler uses first trace to record a complete program flow and then (offline) 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. Online profiler records only selected function entry and exit points and when the recording ends the statistic is run and the results displayed.

Select 'ETM Reconstruction' in the Hardware/Analyzer Setup dialog for Online Profiler and 'ETM' for Offline Profiler use.



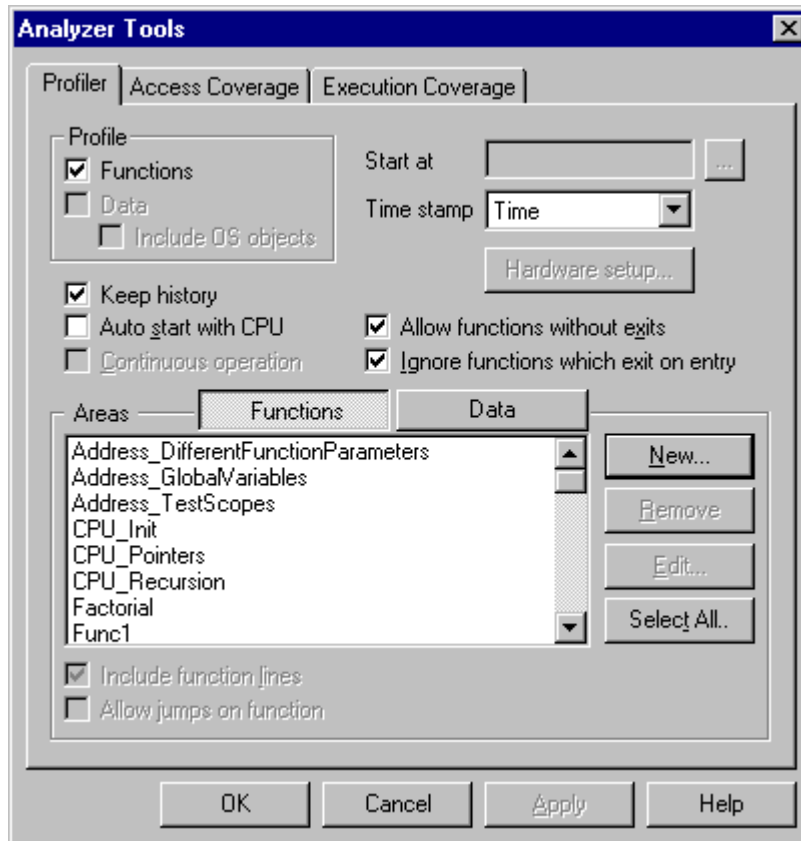
Next, select 'Profiler' window from the View menu and configure Profiler settings. Select 'Functions' option in the 'Profile' field when profiling functions. In order to profile a data variable, 'Data' should be checked instead. For instance, Data Profiler can be used as a Task Profiler, when the operating system writes a unique task ID to a particular global variable at every task switch. The Profiler is then configured to profile that particular variable.

Make sure that 'Keep history' option is checked if Code Execution view is going to be used during results analysis.

Finally, profiled C/C++ functions are selected by pressing 'New...' button. It's recommended that 'All C Functions' is selected for the beginning. Additionally, 'Include lines' can be checked which will yield in time analysis of each source line belonging to the function.

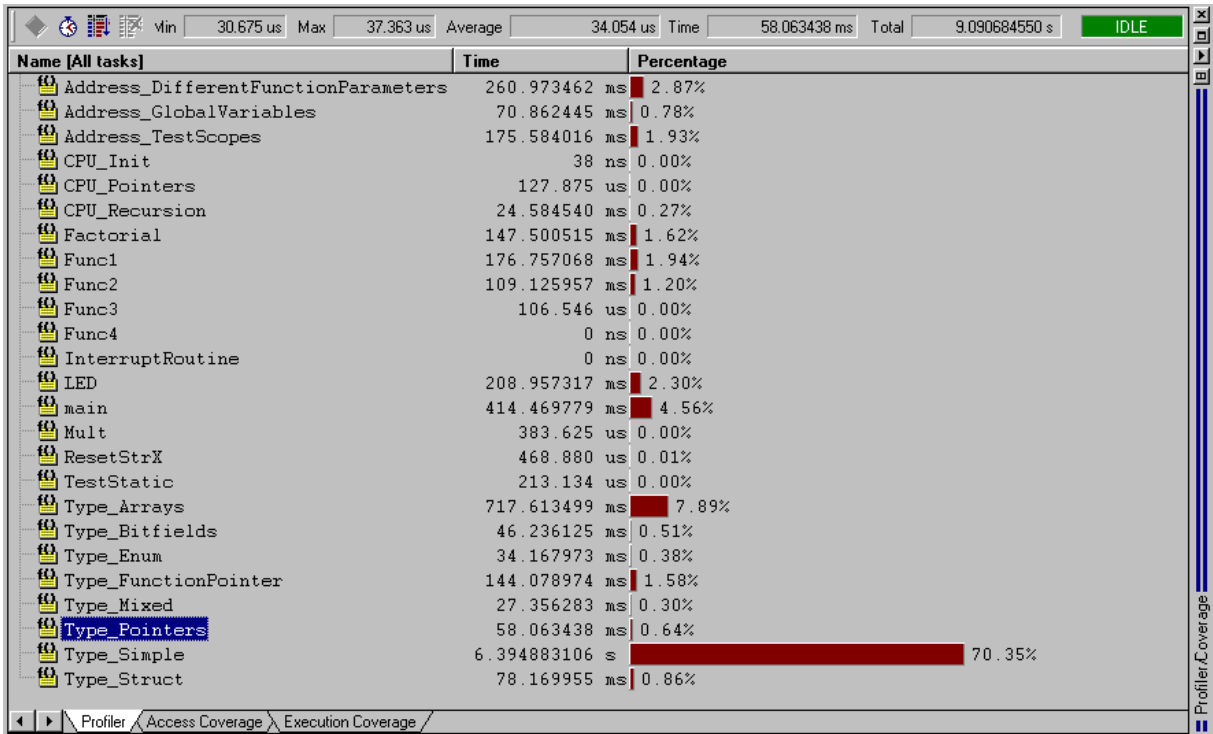
The debugger extracts all the necessary information from the debug info, which is included in the download file and configure hardware accordingly.

Refer to Profiler User's guide for more details on Profiler configuration and use.

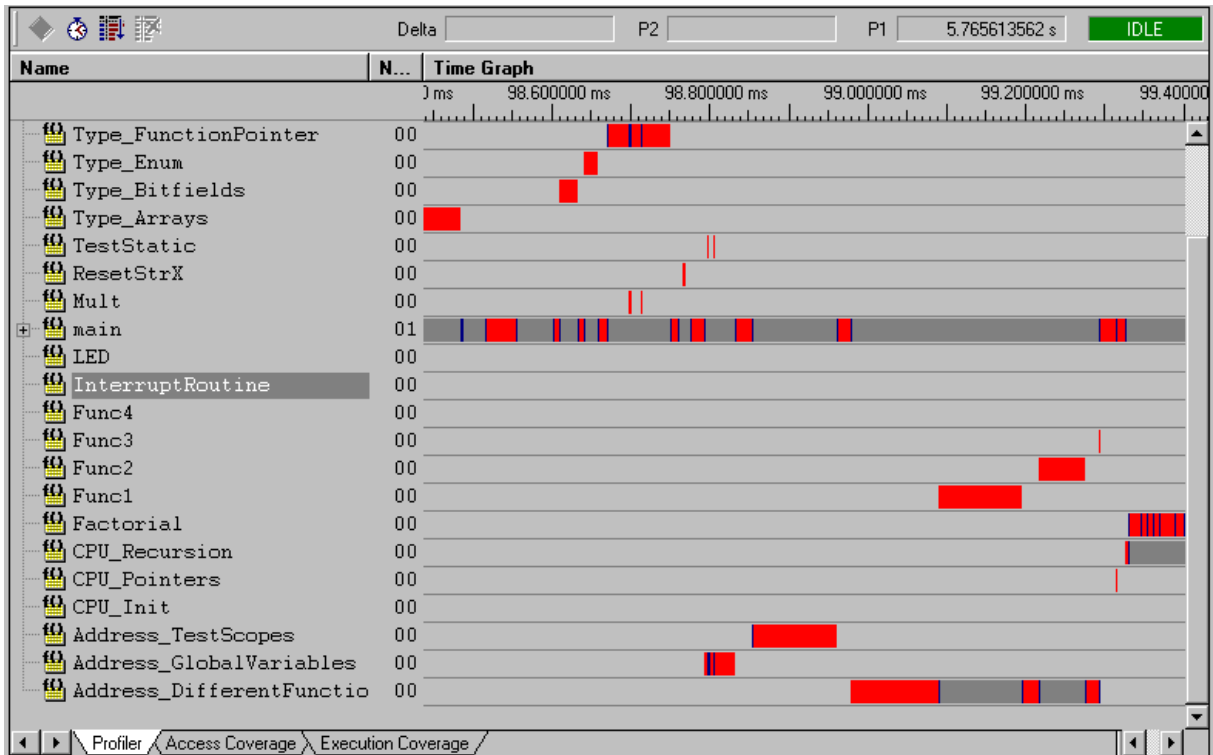


Profiler configuration settings

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



Profiler results – Code Statistics view



Profiler results – Code Execution view

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.