
Technical Notes

National CRX Family On-Chip Emulation

Contents

Contents.....	1
1 Introduction	2
2 Emulation options.....	3
2.1 Hardware Options.....	3
2.2 Initialization Sequence	4
2.3 JTAG Scan Speed.....	6
3 CPU Setup.....	7
3.1 General Options.....	7
3.2 Debugging Options.....	8
3.3 Advanced Options	9
4 FLASH programming.....	9
5 Real-Time Memory Access	9
6 Access Breakpoints	10
7 Trace.....	11
7.1 On-Chip Trace Trigger Configuration.....	12
7.2 On-Chip Trace Qualifier Configuration	13
7.3 Troubleshooting Scenarios	15
7.3.1 Record everything.....	15
7.3.2 Plain Trigger Configurations	16
7.3.3 Advanced Trigger.....	23
8 RTR Execution Coverage.....	35
9 Profiler.....	37
9.1 On-Chip Profiler.....	37
9.2 RTR Execution Profiler	39
10 Getting Started.....	42
11 Troubleshooting.....	42

1 Introduction

The CRX is the next generation CompactRISC architecture. This architecture extends the earlier CR16 family of products by providing full 32-bit processing and memory space, additional registers, ultra-fast context switching and architectural extensibility features.

The CRX supports a variety of debugging features, which are utilized by the hardware debug module (SDI). Optionally, Nexus Trace Module can be built into the CPU, which allows external hardware to implement advanced features like trace, profiler and execution coverage. External development system communicates with CRX SDI via standard IEEE1149.1 (JTAG) port.

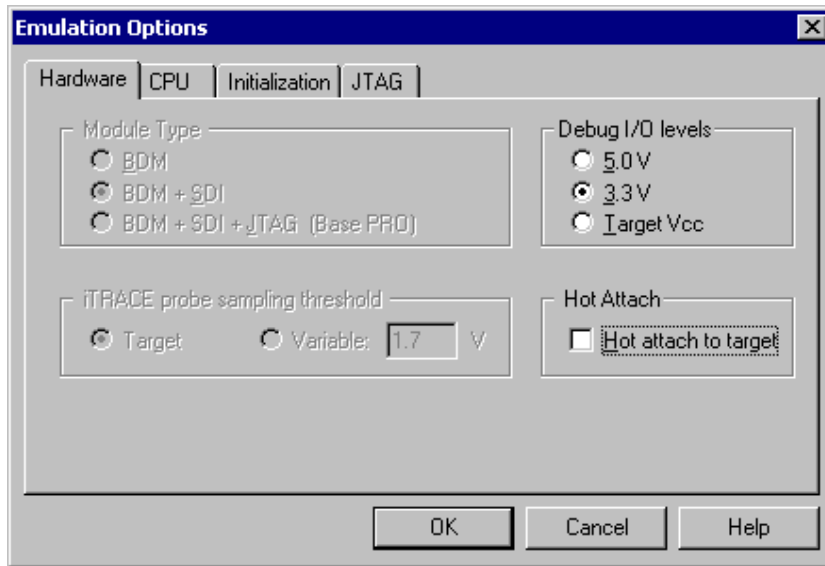
Debug Features

The CRX Emulation System features:

- Up to 16 hardware breakpoints
- Unlimited software breakpoints, including in the internal FLASH
- Access breakpoints
- Real-time access
- Fast internal FLASH programming
- On-Chip Nexus Trace
- RTR Execution Profiler
- RTR Execution Coverage

2 Emulation options

2.1 Hardware Options



Emulation options, Hardware pane

Debug I/O levels

The development system can be configured in a way that debug (BDM/JTAG) signals are driven at 3.3V, 5V or target voltage levels. When 'Target Vcc' Debug I/O level is selected, a voltage applied to the belonging reference voltage pin (target debug connector) is used as a reference voltage for driving debug (BDM/JTAG) signals. Make sure that the target reference voltage pin is connected when 'Target Vcc' Debug I/O level is selected

Hot Attach

The JTAG module supports the Hot Attach function. This is a function, which enables the emulator to be connected to a working target device and have all debug functions available.

The procedure for Hot Attach:

1. The target application should be running.
2. Hot Attach should be selected in the software.
3. A download should be performed, but without the JTAG cable connected. The emulator will be initialized and the ATTACH status will be shown.
4. Connect the JTAG cable.
5. Select the Attach option in the Debug menu. When this option is selected, the emulator tries to communicate through JTAG. If it is successful, it shows the STOP or RUNNING status. At this point, all debug functions are available.
6. When the debugging is finished, the CPU should be set to running and Detach selected from the Debug menu. The status shown is ATTACH. Now the JTAG cable can be safely removed.

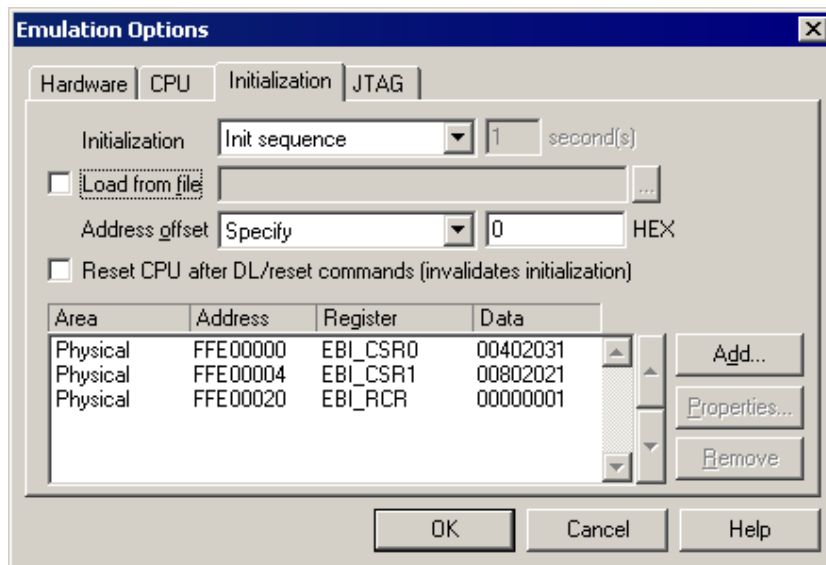
2.2 Initialization Sequence

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

Note: Normally, there is no need for initialization sequence in case of a single chip application/CPU.

The initialization sequence can be set up in two ways:

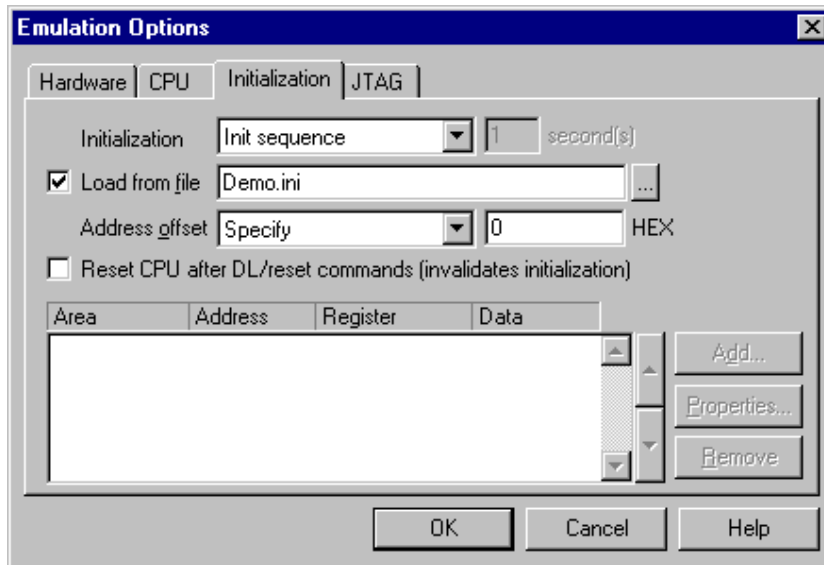
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



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

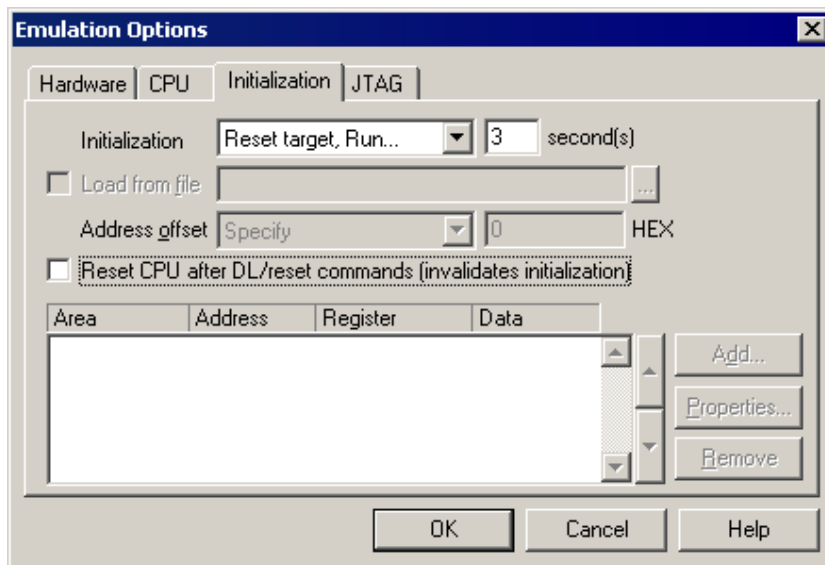
Excerpt from the Demo.ini:

```
S EBI_CSR0 L 0x00402031 // CS0 - ext. flash, 4 wait states
S EBI_CSR1 L 0x00802021 // CS1 - ext. SRAM
S EBI_RCR L 0x00000001 // remap internal RAM
```

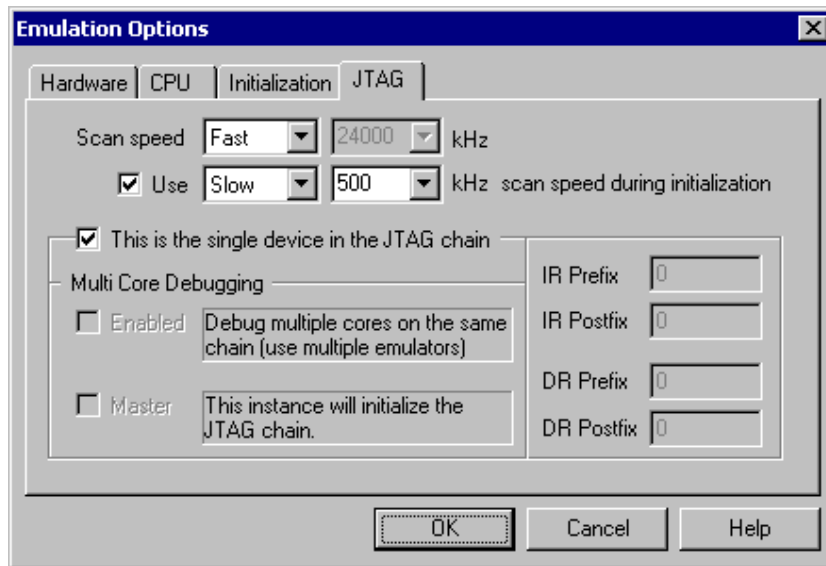


The advantage of the second method is that you can simply distribute your .ini file among different workspaces and users. Additionally, you can easily comment out some line while debugging the initialization sequence itself.

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



2.3 JTAG Scan Speed



JTAG Scan Speed definition

Scan speed

The JTAG chain scanning speed can be set to:

- Slow - long delays are introduced in the JTAG scanning to support the slowest devices. JTAG clock frequency varying from 1 kHz to 2000 kHz can be set.
- Fast – the JTAG chain is scanned with no delays.
- Burst – provides the ability to set the JTAG clock frequency varying from 4 MHz to 100 MHz.
- Burst+ - provides the ability to set the JTAG clock frequency varying from 4 MHz to 100 MHz

Slow and Fast JTAG scanning is implemented by means of software toggling the necessary JTAG signals. Burst mode is a mixture of software and hardware based scanning and should normally work except when the JTAG scan frequency is an issue that is when the JTAG scan frequency used by the hardware accelerator is too high for the CPU. In general, selecting an appropriate scan frequency usually depends on scan speed limitations of the CPU. In Burst+ mode, complete scan is controlled by the hardware accelerator, which poses some preconditions, which are not met with all CPUs. Consequentially, Burst+ mode doesn't work for all CPUs.

In general, Fast mode should be used as a default setting. If the debugger works stable with this setting, try Burst or Burst+ mode to increase the download speed. If Fast mode already fails, try Slow mode at different scan frequencies until you find a working setting.

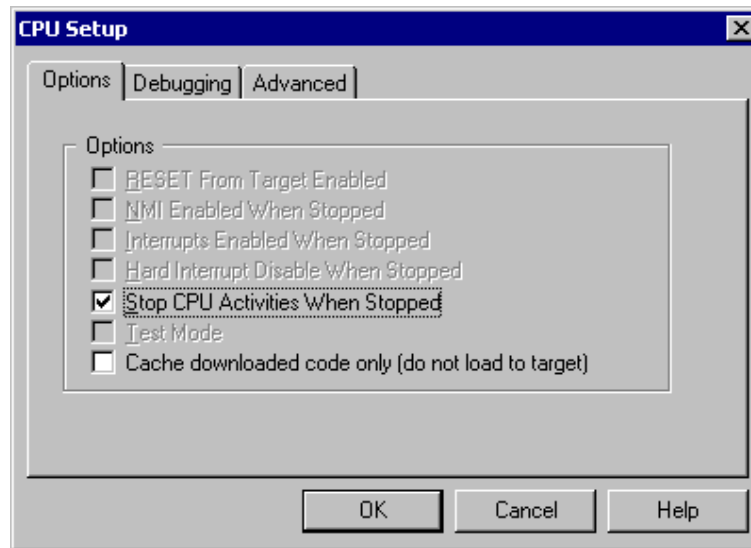
Note: Burst and Burst+ modes are implemented only for PowerPC and ARM CPUs, including XScale.

Use – Scan Speed during Initialization

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

3 CPU Setup

3.1 General Options



General options dialog

Stop CPU Activities When Stopped

When the option is checked, all internal peripherals like timers and counters are stopped when the application is stopped. Otherwise, timers and counters remain running while the program is stopped. Usually, when the option is checked, the emulation system behaves more consistently while stepping through the program. While being aware of the consequences, it is up to the user whether the option is checked or not.

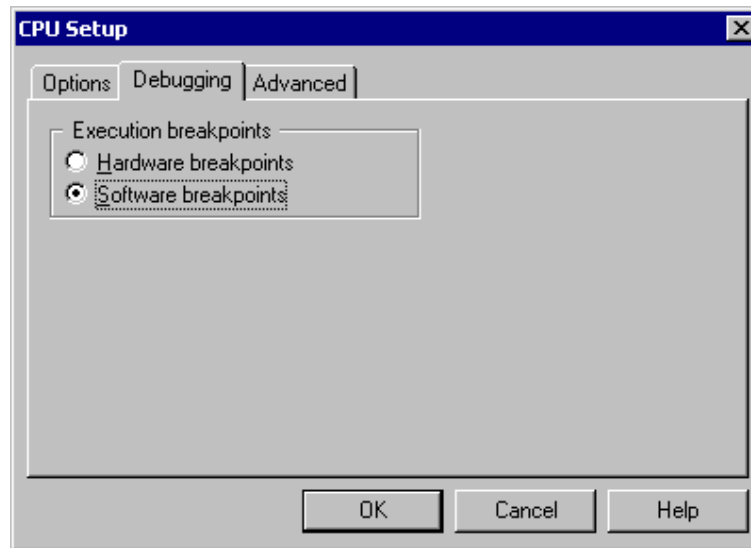
For instance, it's recommended that a timer, which generates interrupts, is stopped when the application is stopped. Otherwise, the CPU would first service all pending interrupts (generated by the timer while the application was stopped) after the application is resumed. Such behaviour is far away from the actual behaviour of the target application.

Cache Downloaded Code only (do not load to target)

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

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

3.2 Debugging Options



Debugging options dialog

Execution breakpoints

Hardware Breakpoints

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

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

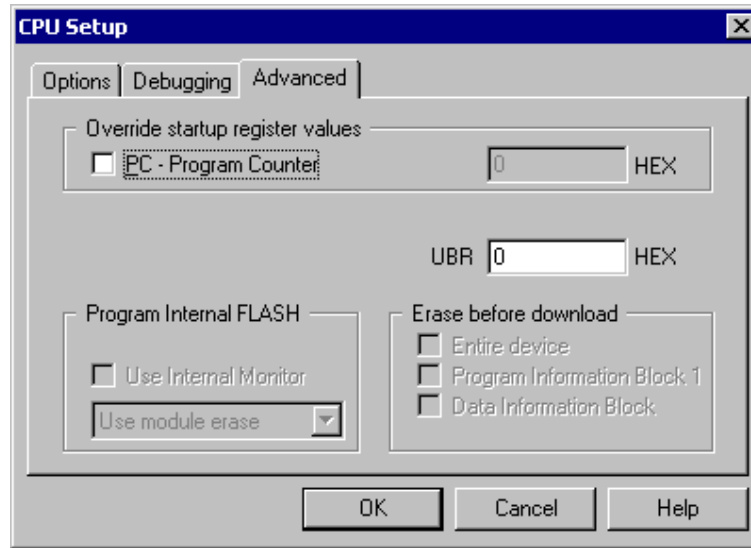
The same on-chip debug resources are shared among hardware execution breakpoints, access breakpoints and on-chip trace trigger. Consequentially, debug resources used by one debug functionality are not available for the other two debug functionalities. In practice this would mean that no trace trigger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

Software Breakpoints

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

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

3.3 Advanced Options



Advanced Emulation Options

Override startup register values

The user can preset PC after reset.

UBR

The User Base register (TMR) is used to specify the address of the Ownership Trace register, the CPU has to write to in order to force an Ownership Trace message (if Ownership Trace is enabled).

For more information on Ownership Trace and the UBR register, please see the CRX user's manual.

4 FLASH programming

The CRX CPUs have internal flash, which is programmed through standard debug download; thereby no standard FLASH setup dialog is available. The debugger recognizes which code from the download file fits in the FLASH. All necessary FLASH programming settings are done in the 'CPU Setup/Advanced' dialog.

5 Real-Time Memory Access

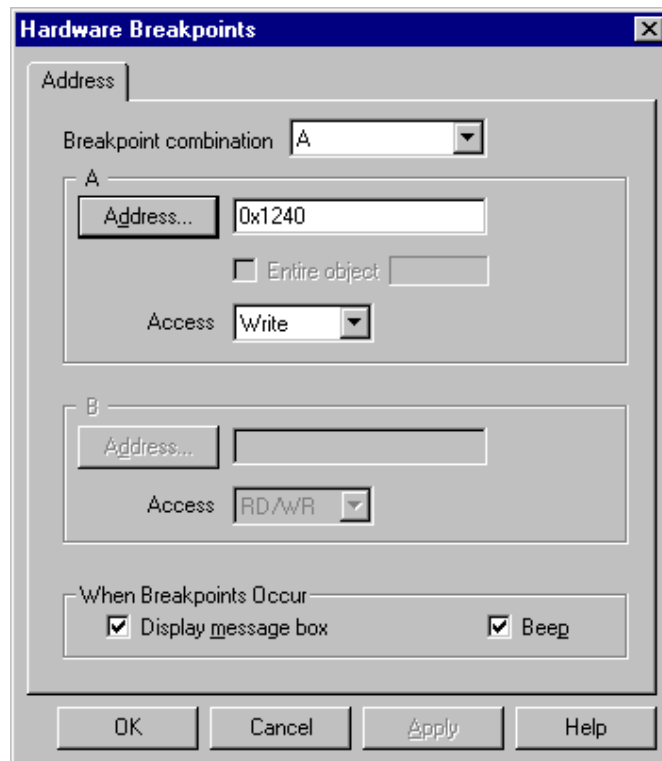
CRX family debug module supports real-time memory access. Watch window's ***Rt.Watch*** panes can be configured to inspect memory without stalling the CPU. Optionally, memory and SFR windows can be configured to use real-time access as well.

Please refer to the Software User's Guide for more information on Real-Time watches.

6 Access Breakpoints

The same on-chip debug resources are shared among hardware execution breakpoints, access breakpoints and 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.

Four independent breakpoint modules are available. Each module has two breakpoint registers, which enable the following functions: breakpoint A, breakpoint B, breakpoint A or breakpoint B, area (from A to B) and breakpoint A then B. For each breakpoint the access method can be selected (execution, data read, data write, data read or write). The software automatically uses 3 breakpoint modules as execution breakpoints (i.e. 5 user breakpoints and one reserved for debugging), the fourth breakpoint module is used as an access breakpoint module with data read, data write and data read or write access types.



CRX Hardware Breakpoints dialog

Breakpoint combination

Access breakpoint can be set on a debug event combined from two watchpoints A and B. Possible combinations: A, B, A OR B, A THEN B or a range defined by A and B.

A, B

The address of the trigger condition is defined here. Access type can be Execute, Read/Write, Write or Read.

When Breakpoints Occur

A beep can be issued and/or a message displayed indicating that an access breakpoint has occurred.

7 Trace

CRX family implements on-chip trace support according to the Nexus standard.

Instruction Trace

Using a branch-trace mechanism, the instruction 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 trace 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 trace 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, don't contain valid time information. Any interpolation with the recorded addresses containing valid time stamp is misleading for the user. Thereby, more frames displayed in the trace window have the same time stamp value.

Data Trace

The data trace feature 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 control.

Transmitted information about the memory access cannot be compressed fundamentally since each memory access is distinctive and not predictable. Errors in the trace may appear when the CPU executes too many data accesses in short period, which yield numerous Nexus messages (internal data message FIFO overflow), which cannot be sent out through the narrow Nexus port to the external development system on time.

Therefore, it's highly recommended to configure on-chip message control to restrict data trace recording only to data areas of interest to minimize possible overflows.

OTM Trace

Ownership trace messages (OTM) are generated, when the application, for instance operating system writes to the process ID register (PID0) or the memory mapped ownership trace register (OTR).

OTM trace can be used for Task Profiler. For instance, a unique value named task ID is assigned to each task. Then it's presumed that the application writes task ID to PID0 or OTR on every task switch. As a result, the debugger can display how the tasks were switched through the time by collecting OTM messages and relying on known task IDs.

Trace Features (iTRACE PRO):

- Compliant with Nexus standard
- External trace buffer
- Instruction, Data and OTM Trace
- On-Chip Trigger and Qualifier
- Time Stamps
- AUX inputs
- Trigger Input/Output
- Nexus Reconstruction (RTR)
- Advanced Trigger (RTR)
- RTR Execution Profiler
- RTR Execution Coverage

7.1 On-Chip Trace Trigger Configuration

The same on-chip debug resources are shared among hardware execution breakpoints, access breakpoints and 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 16 execution breakpoints are set already.

Trigger On

Trace can trigger immediately after the trace is started or can trigger on a debug event combined from two watchpoints A and B. Possible combinations: A, B, A OR B, A THEN B or a range defined by A and B.

A, B

The address of the trigger condition is defined here. Access type can be Execute, Read/Write, Write or Read.

Buffer Size

Buffer Size determines the depth of the trace buffer. It's recommended to use minimum buffer size which results in faster trace upload time and smaller trace file.

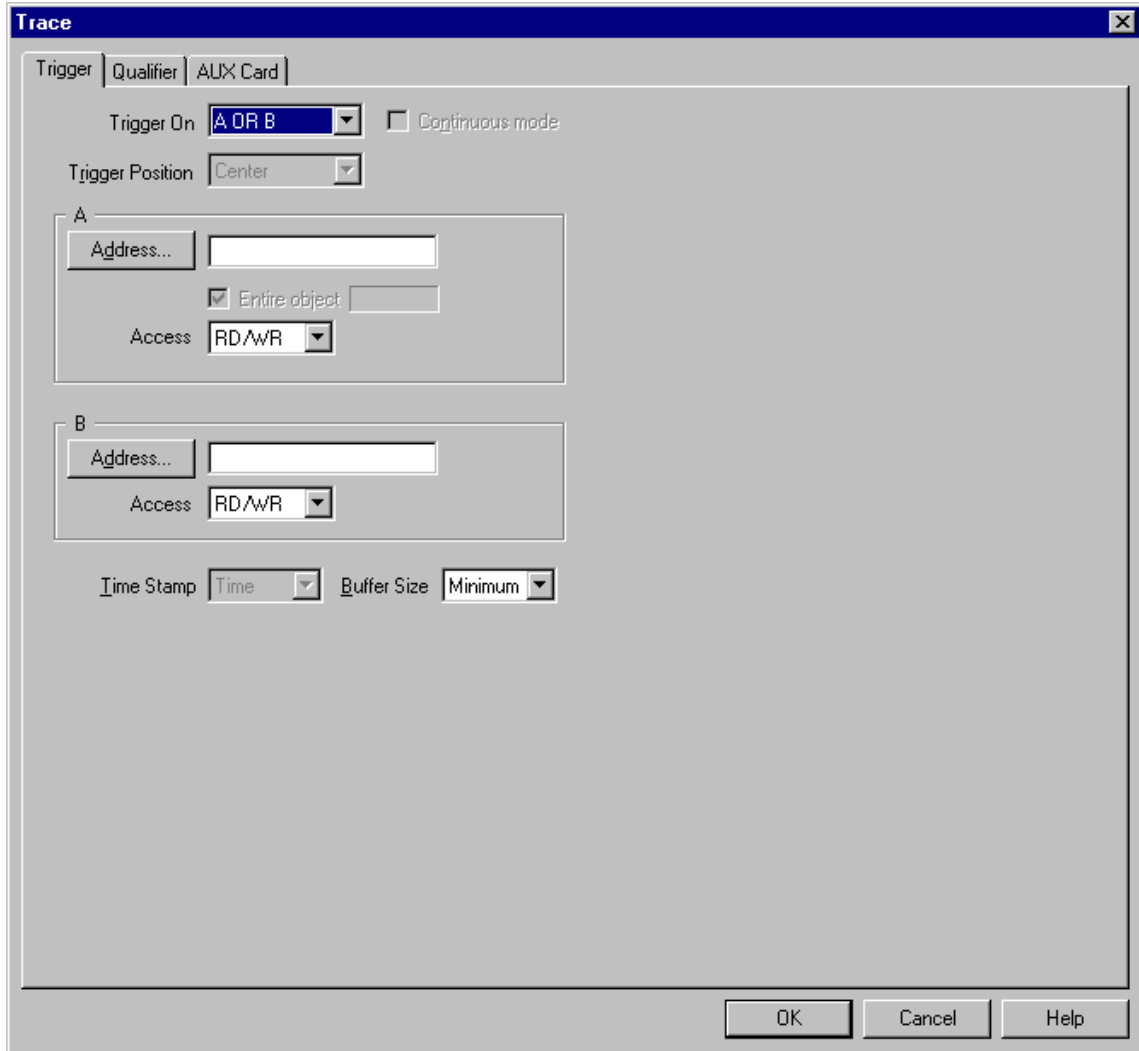
Trigger Position

Depending on the needs, trigger can be located at the beginning, in the center or at the end of the trace buffer. Note that the trigger position is not configurable for minimum buffer. If the user intends to analyze the trace record after the trigger, it makes sense to use 'Begin' trigger position and 'End' trigger position when the trace pre-history that is program behavior before trigger is required.

Continuous mode

Continuous mode keeps recording program execution until the CPU or trace is manually stopped. Normally, the trace records only until the trace buffer is full. This option should be checked, when it's required to analyze

program behavior just before the CPU is stopped. Additionally, the Trigger should be set to 'anything in such configuration.



Trace Trigger dialog

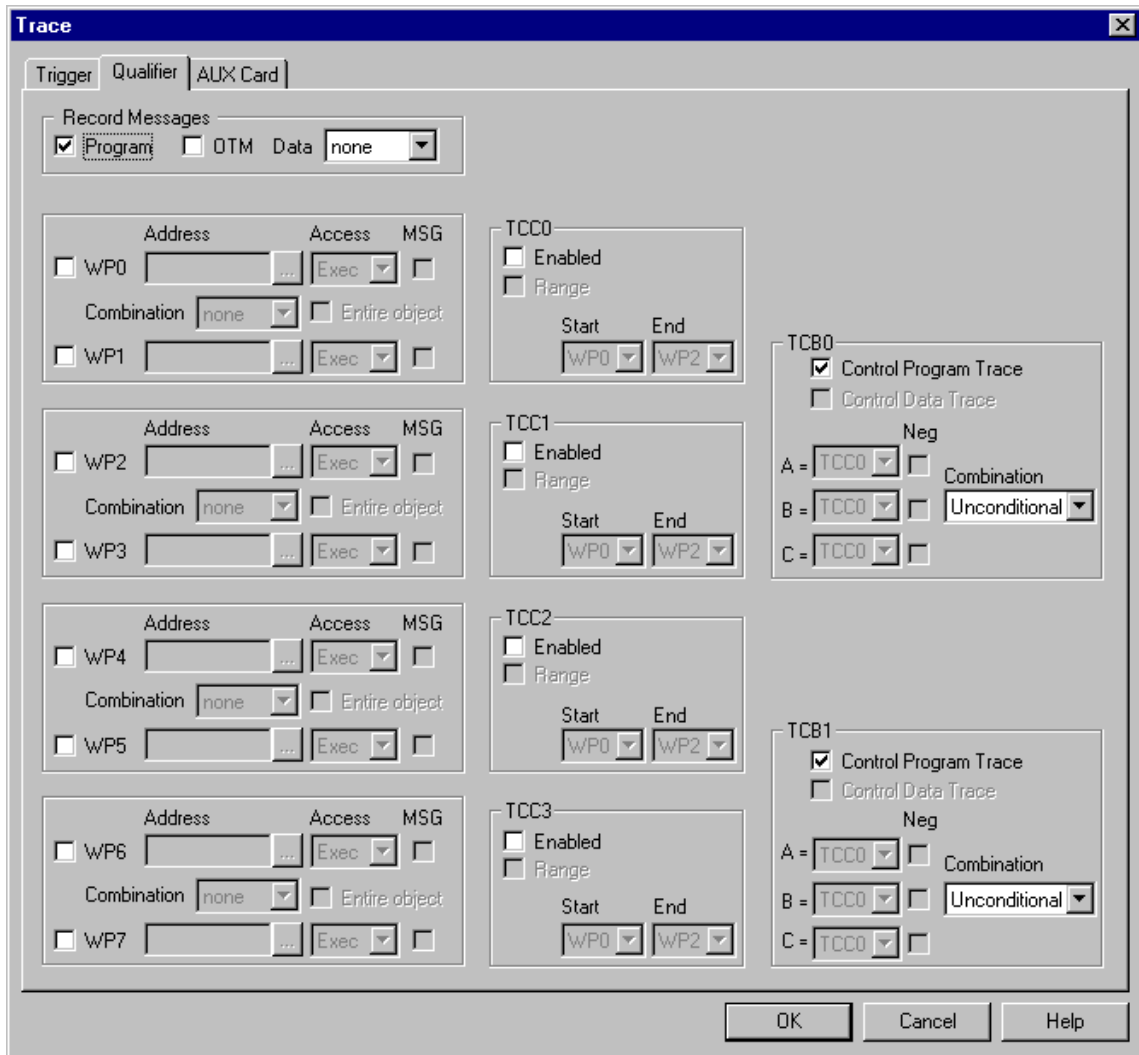
7.2 On-Chip Trace Qualifier Configuration

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 in internal data message FIFO overflows due to a limited Nexus port bandwidth. Instruction path reconstruction fails as soon as it comes to the FIFO overflow.

To stay away from possible overflows, the user should use qualifier(s), which allows defining the data of interest to be traced only. This minimizes the number of data access messages to be sent through the Nexus port.

When Qualifier is used on Program Trace, the trace may capture more program flow (from the aspect of time) using the same trace buffer size.

Refer to CPU user's manual for mode details on configuring qualifier. Qualifier dialog completely matches with available on-chip debug resources and there should be no problems to relate described on chip resources from the user's guide with the qualifier dialog.



On-Chip Trace Qualifier dialog

Record Messages

This is a global setting, which enables Program, Data and OTM trace. Program trace only is enabled by default.

TCB0, TCB1

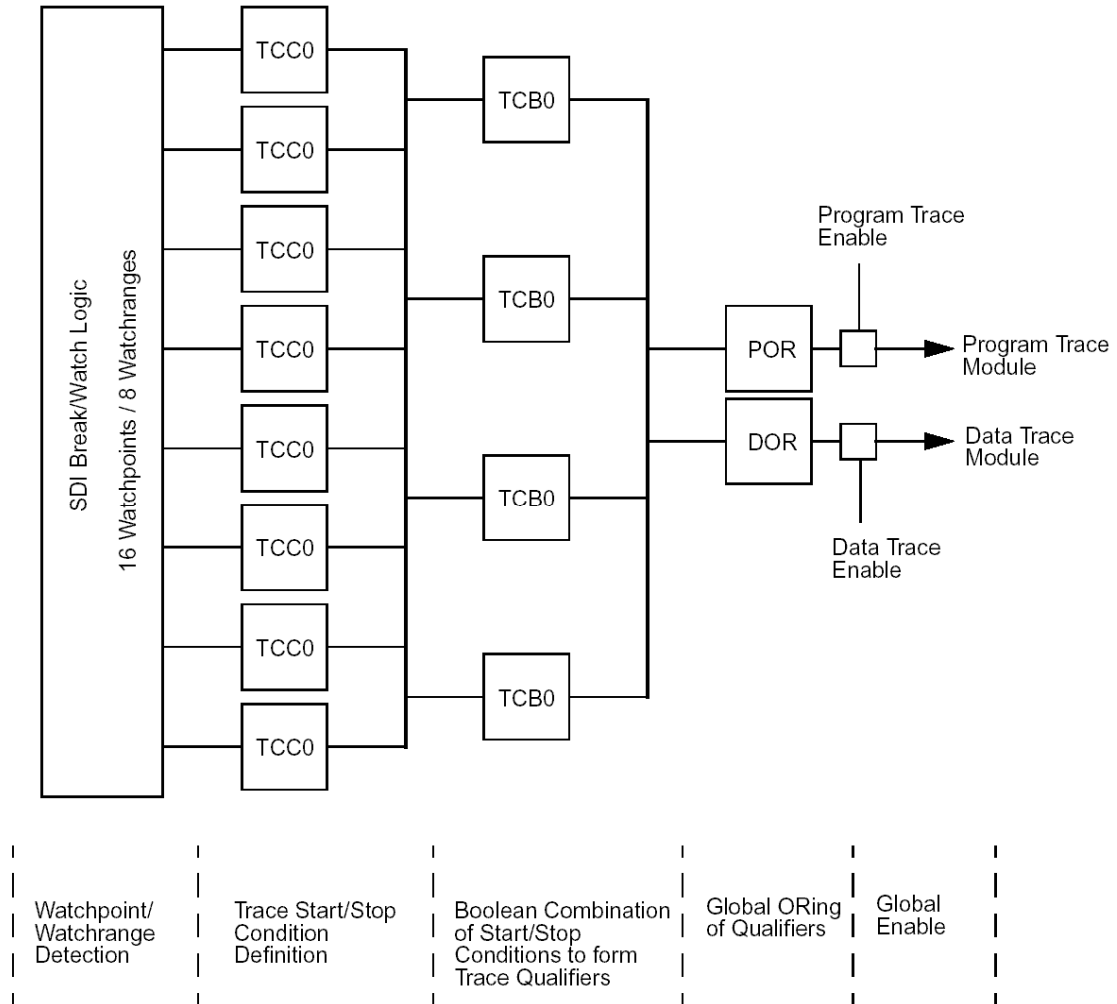
TCB0 and TCB1 can be used to control Program and Data trace. When for instance, a complete program execution must be recorded, use TCB0 to control Program trace and then use TCB1 to record data accesses in limited memory range(s).

When TCB is used to restrict recorded information, it's logically combined from event A, B and C. Each of these events can be TCC0, TCC1, TCC2 or TCC3 event, which are configured independently.

TCC0-TCC3

TCC condition is defined by start and end condition, where start and end condition is selected watchpoint (WP0-WP7) match. As an alternative, TCC valid condition can be watchpoint range match.

8 watchpoints can be used to limit program or data trace recording. They can be configured for 8 address matches or 4 address ranges. Access type can be Execute, Read/Write, Read or Write.



Qualifier on-chip resources

7.3 Troubleshooting Scenarios

7.3.1 Record everything

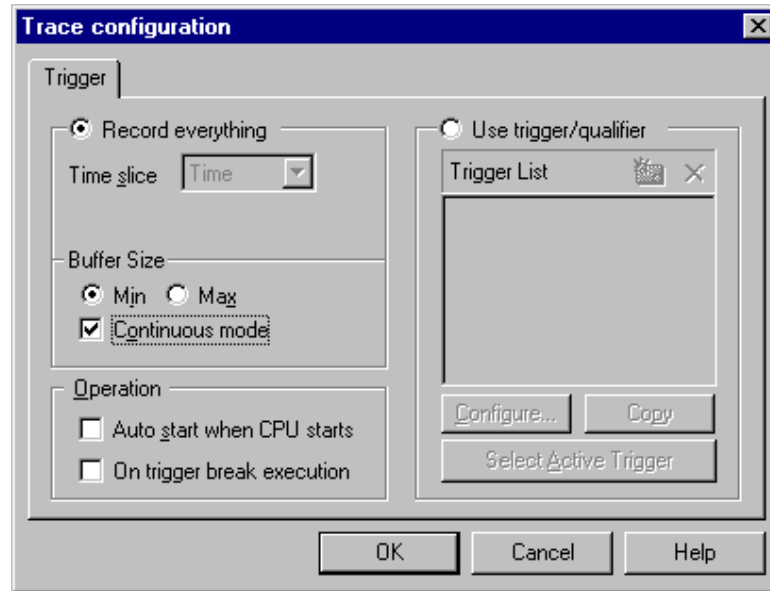
This configuration is used to record the contiguous program flow either from the program start on or up to the moment, when the program stops.

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: 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.

- Select 'Record everything' operation type in the 'Trace configuration' dialog and make sure that 'Continuous mode' option is checked to ensure that the trace buffer rolls over while recording the running program. The trace will stop as soon as the CPU is stopped.
- Select minimum or maximum buffer size depending on the required depth of the trace record. Have in mind that the minimum buffer uploads faster than the maximum.



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.

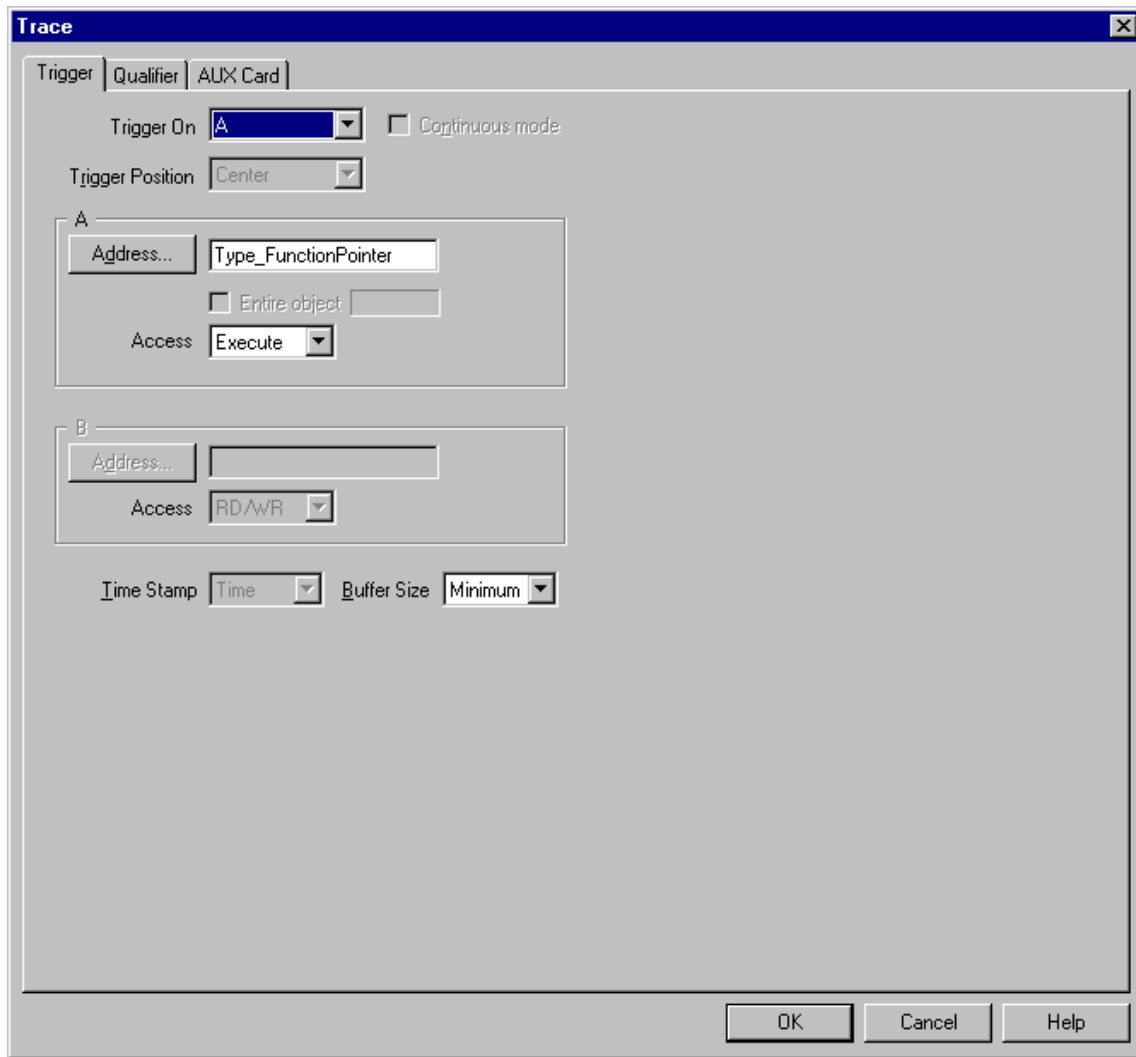
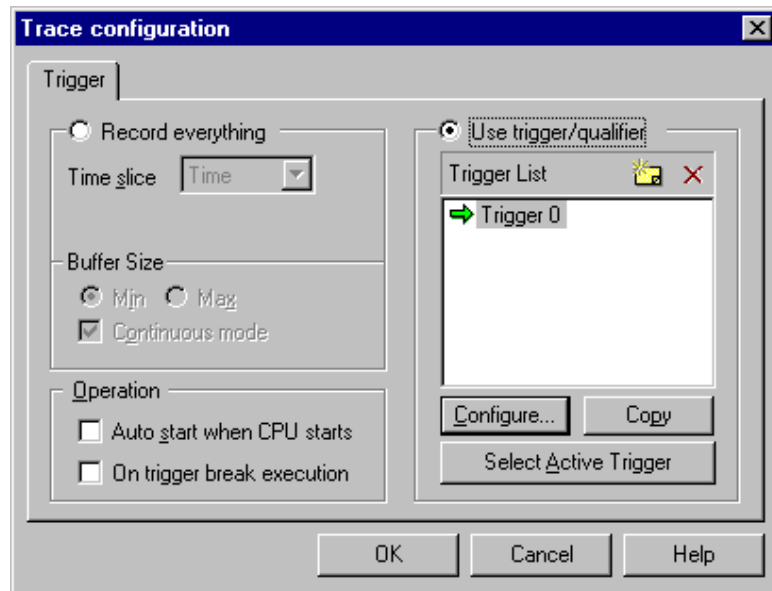
7.3.2 Plain Trigger Configurations

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: Trace starts recording after the `Type_FunctionPointer` function is called. Only data accesses to the `iCounter` variable should be recorded besides the program flow.

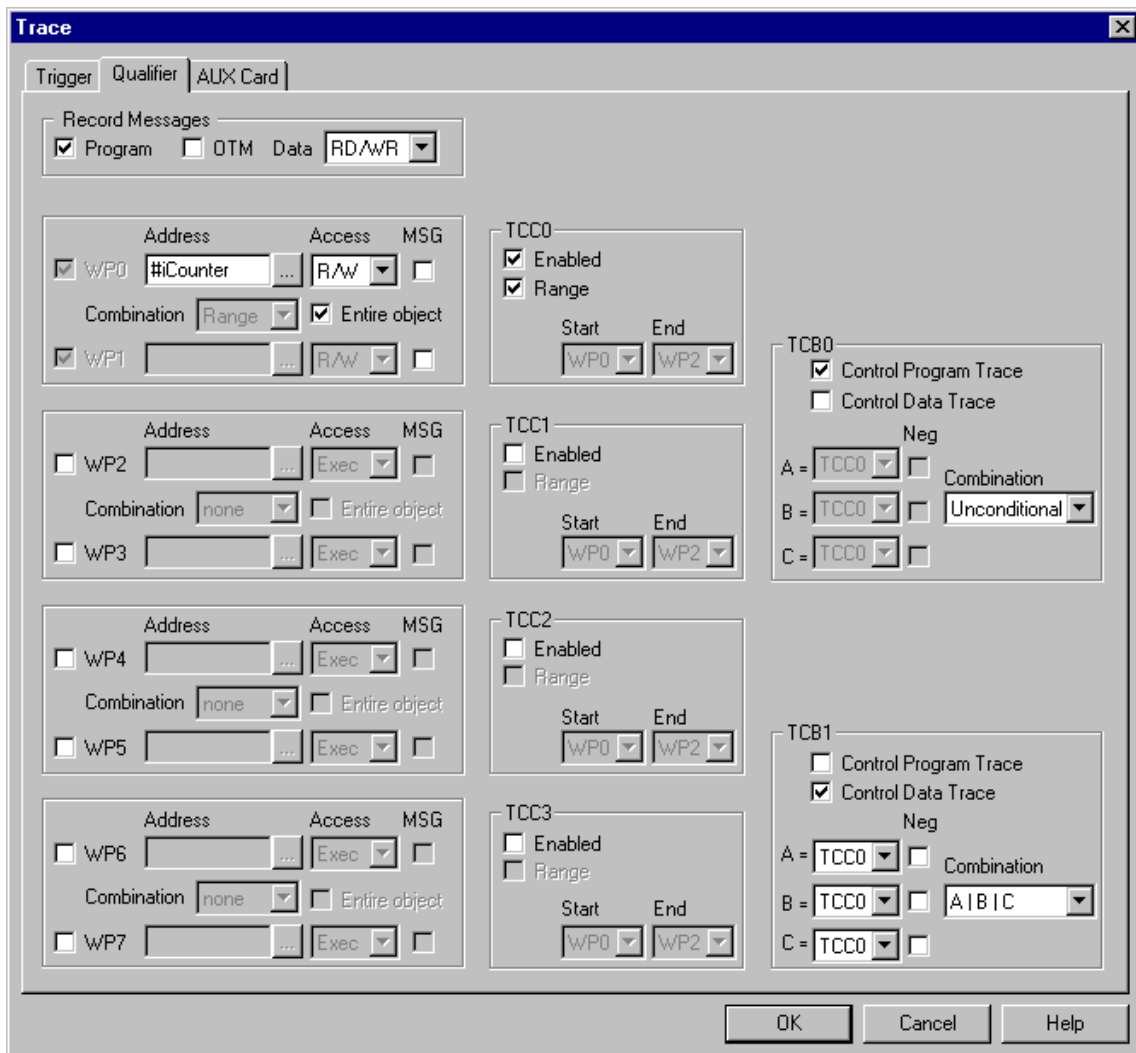
- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add and name a new trigger (e.g. `Trigger0`), and open 'Trigger and Qualifier Configuration' dialog.
- Configure trigger on watchpoint A and define watchpoint A in the Trace Trigger dialog. Set `Type_FunctionPointer` for address and Execute for Access type.



Trace Trigger dialog

- Next, configure qualifier in the Trace Qualifier dialog. First, enable program and data trace in the Record Messages field.
- Let's use TCB0 to control program trace and TCB1 to control data trace. Leave TCB0 at default setting (Unconditional) which results in program trace recording complete program flow.
- Select 'A|B|C' condition for TCB1 Combination and select TCC0 event for events A, B and C.
- Configure watchpoint WP0. Set iCounter for the address; set Combination to 'Range' and Access to 'R/W'. Check the 'Entire object' option.
- Finally enable TCC0 event and check the 'Range' option.

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

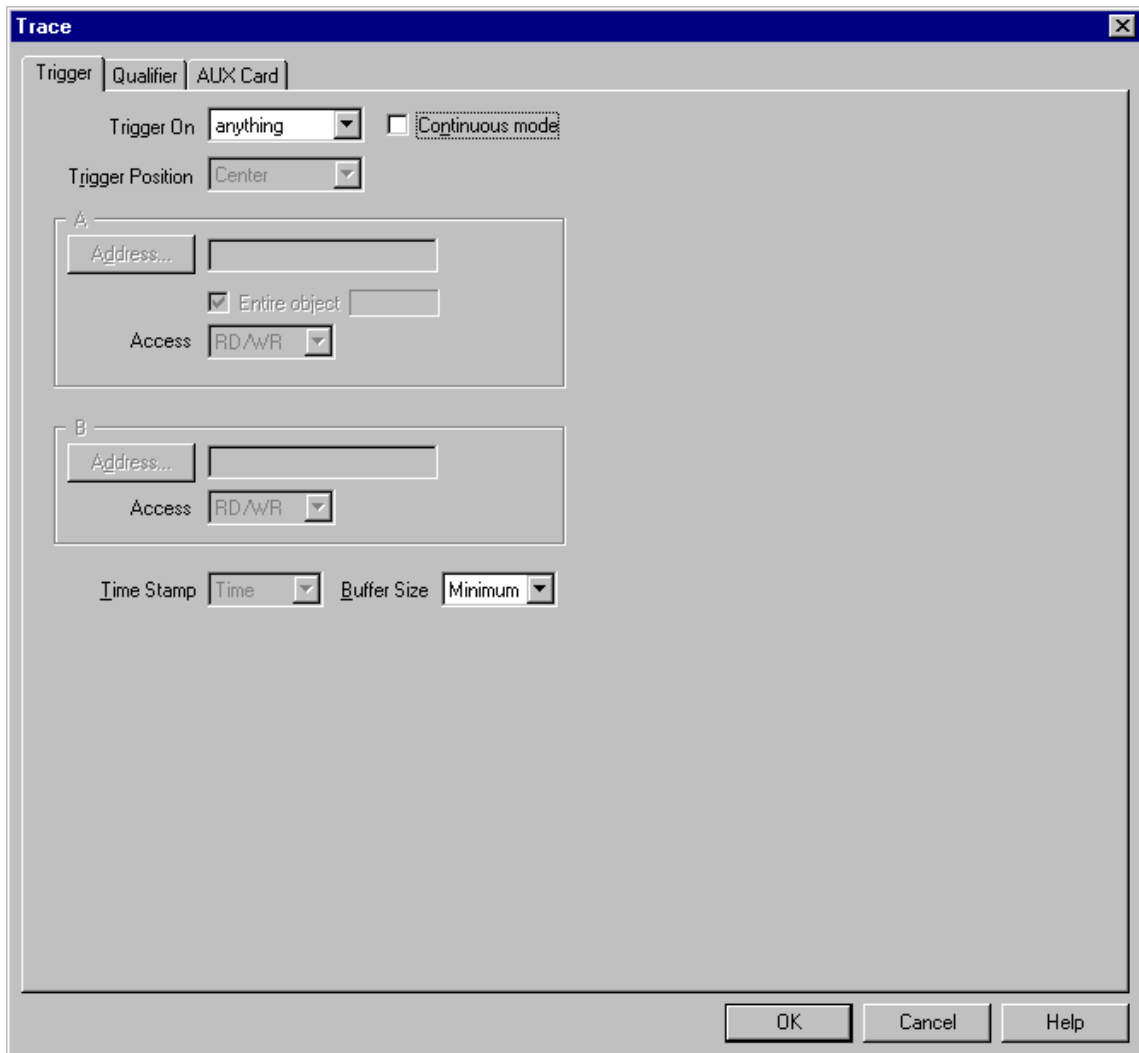


Let's inspect the trace results. Trigger point can be found around frame 0 and is marked as 'Watchpoint' in the Content bus. Two iCounter data accesses can be seen, one is iCounter read, which is followed by write after variable's value is incremented by one.

	Number	Address	Data	Content	Time
	-9	01000526	21100000	Instruction	-15.651 us
	-8	01030000	00065FF4	iCounter iCounter+0 iCounter+0 iCounter+0 Read	-15.651 us
	-7	0100052A	35202110	0100052A 1021 addd \$1,R0 Instruction	-15.651 us
	-6	0100052C	01033520	0100052C 203503010000 stord R0,iCov Instruction	-15.651 us
	-5	0100052E	21CF0000	Instruction	-15.651 us
	-4	01030000	00065FF5	iCounter iCounter+0 iCounter+0 iCounter+0 Write	-15.651 us
	-3	01000532	BAEE21CF	} 01000532 CF21 addd \$12,SP Instruction	-15.651 us
	-2	01000534	0000BAEE	Type_Enum_EXIT_ 01000534 EEBA jr RA Instruction	-15.651 us
	-1	00000000	00000040	Watchpoint	-9.651 us
T	0	0100014E	0000317E	Type_FunctionPointer(); 0100014E 7E310000FF01 bal RA,Type Instruction	0 ns
P	1	01000150	317E01FF	Instruction	0 ns
	2	0100054C	4080346F	{ Type_FunctionPointer 0100054C 6F348040 push SP,{R7,RA} Instruction	0 ns

Example: Trace monitors `iCounter` variable data accesses while the application is running.

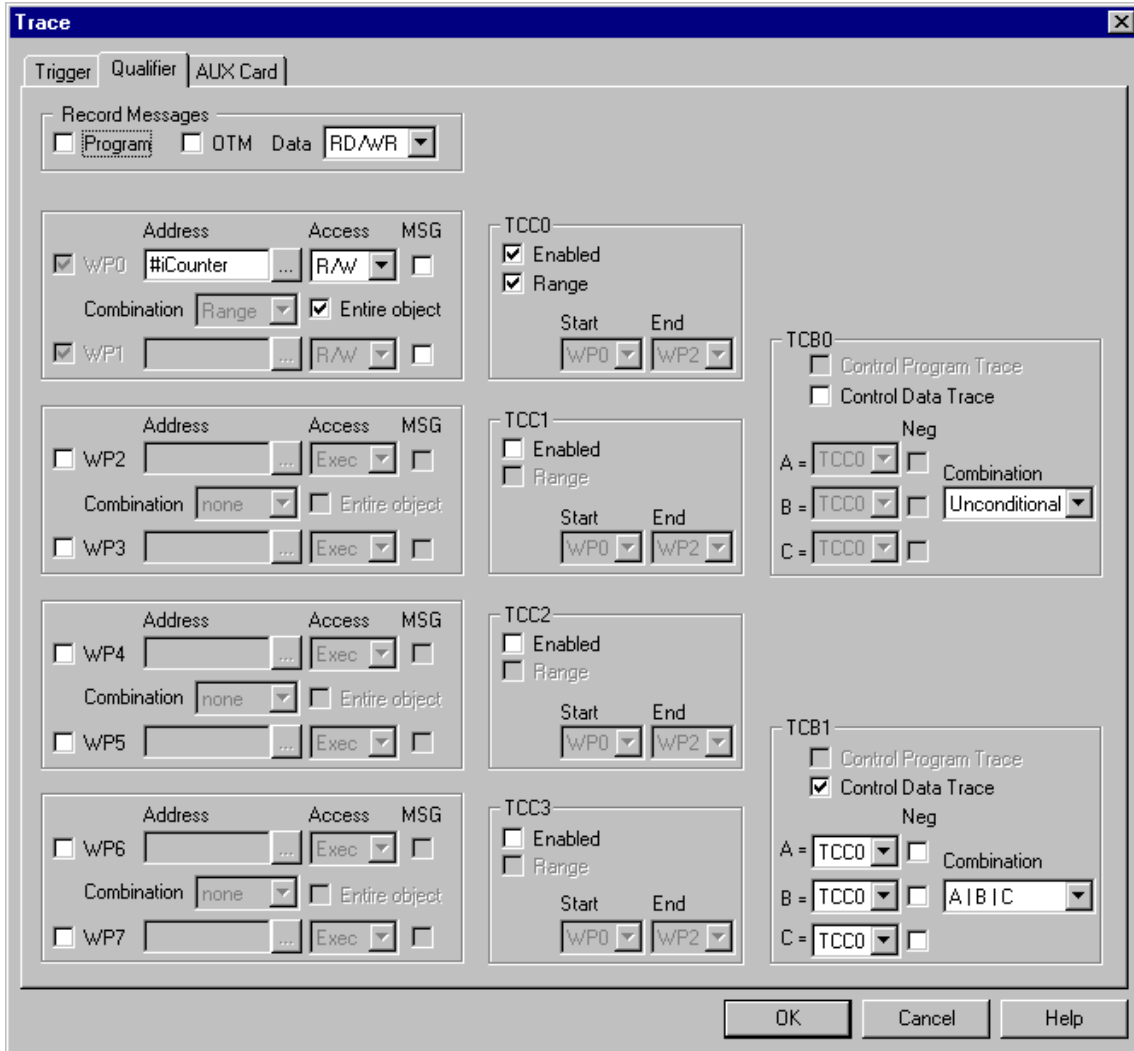
- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Set 'Anything' for the trigger condition in the Trace Trigger dialog.



Trace Trigger dialog

- Next, configure qualifier in the Trace Qualifier dialog. First, enable data trace only in the Record Messages field.
- Let's use TCB1 to control data trace. Make sure that TCB0 doesn't control program nor data trace.
- Select 'AIBC' condition for TCB1 Combination and select TCC0 event for events A, B and C.
- Configure watchpoint WP0. Set `iCounter` for the address; set Combination to 'Range' and Access to 'R/W'. Check the 'Entire object' option.
- Finally enable TCC0 event and check the 'Range' option.

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



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

	Number	Address	Data	Content	Time
1	123	01030000	0000003A	iCounter iCounter+0 iCounter+0 iCounter+0 Write	12.518375 ms
	124	01030000	0000003A	iCounter iCounter+0 iCounter+0 iCounter+0 Read	12.996300 ms
2	125	01030000	0000003B	iCounter iCounter+0 iCounter+0 iCounter+0 Write	12.997963 ms
	126	01030000	0000003B	iCounter iCounter+0 iCounter+0 iCounter+0 Read	13.130613 ms
	127	01030000	0000003C	iCounter iCounter+0 iCounter+0	13.132938 ms

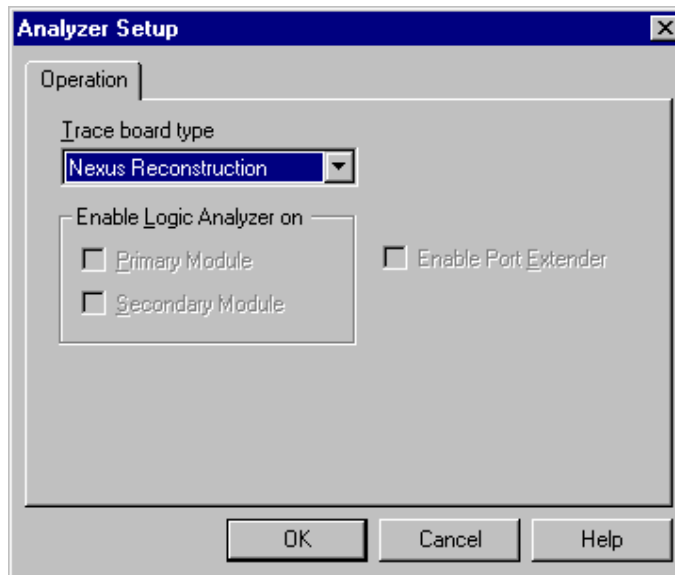
Time difference between two consecutive data write accesses can be measured using markers. In this particular case, the time difference is 479,588 us.

7.3.3 Advanced Trigger

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

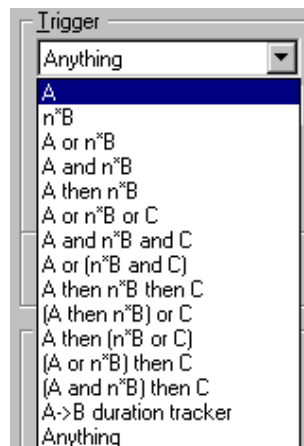
- 3-Level Trigger
- Qualifier
- Watchdog Trigger
- Duration Tracker

'Nexus Reconstruction' must be selected in the Hardware/Analyzer Setup dialog to use these extra features.



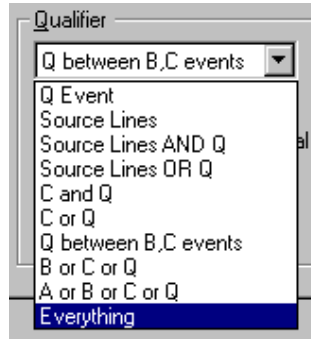
3-Level Trigger

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.

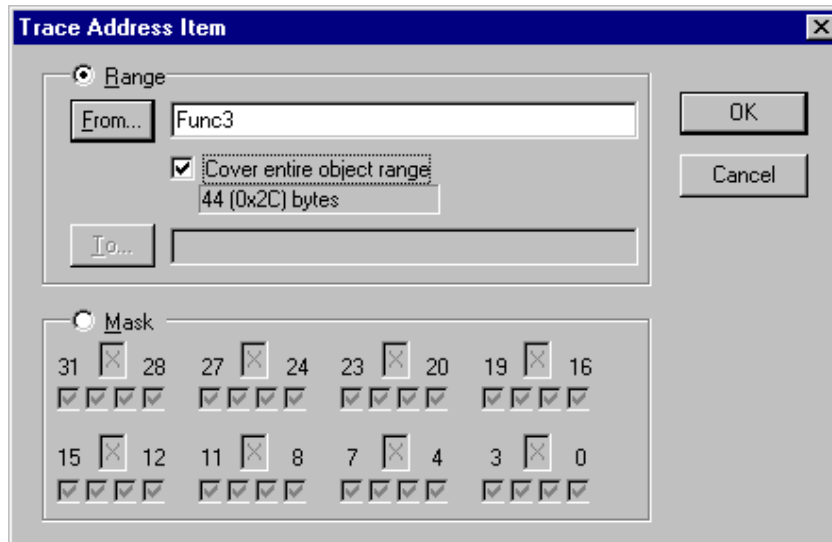


Thereby, the qualifier can be configured in a standard way and then additionally up to 8 CPU cycles can be recorded before and/or after the qualifier. For instance, this allows recording of a function or just its entry point and few instructions recorded before make possible to determine, which code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

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

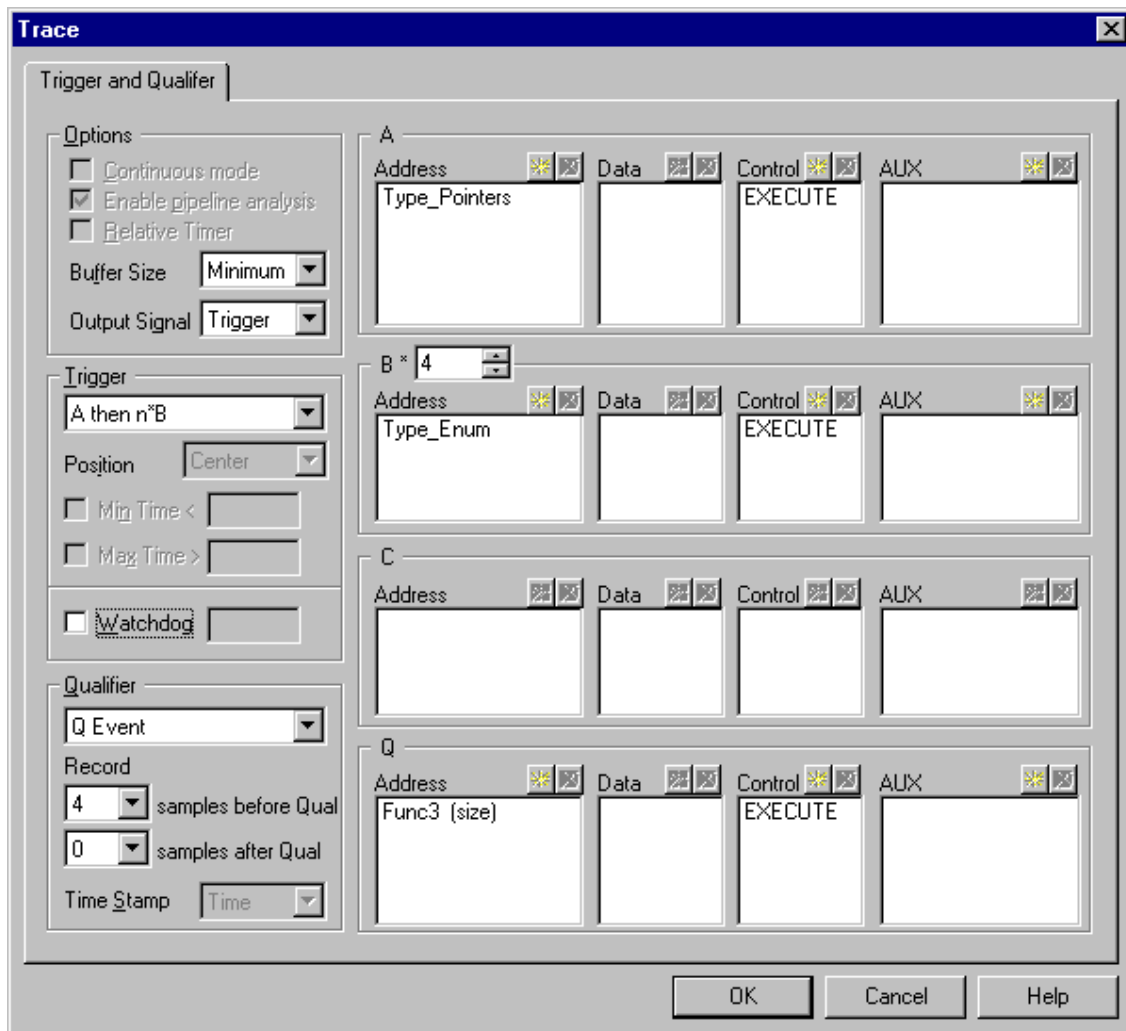
Example: Let's record `Func3` execution after the `Type_Pointers` function is executed and then 4-times `Type_Enum` function is called.

- 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 field.

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 stwa 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

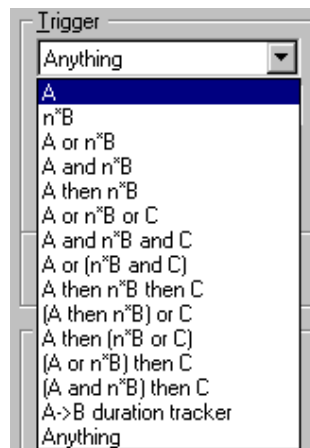
The user needs to enter the trace watchdog time-out period and define the “trace watchdog reset” condition, which can be logically combined from events A, B and C.

- 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.

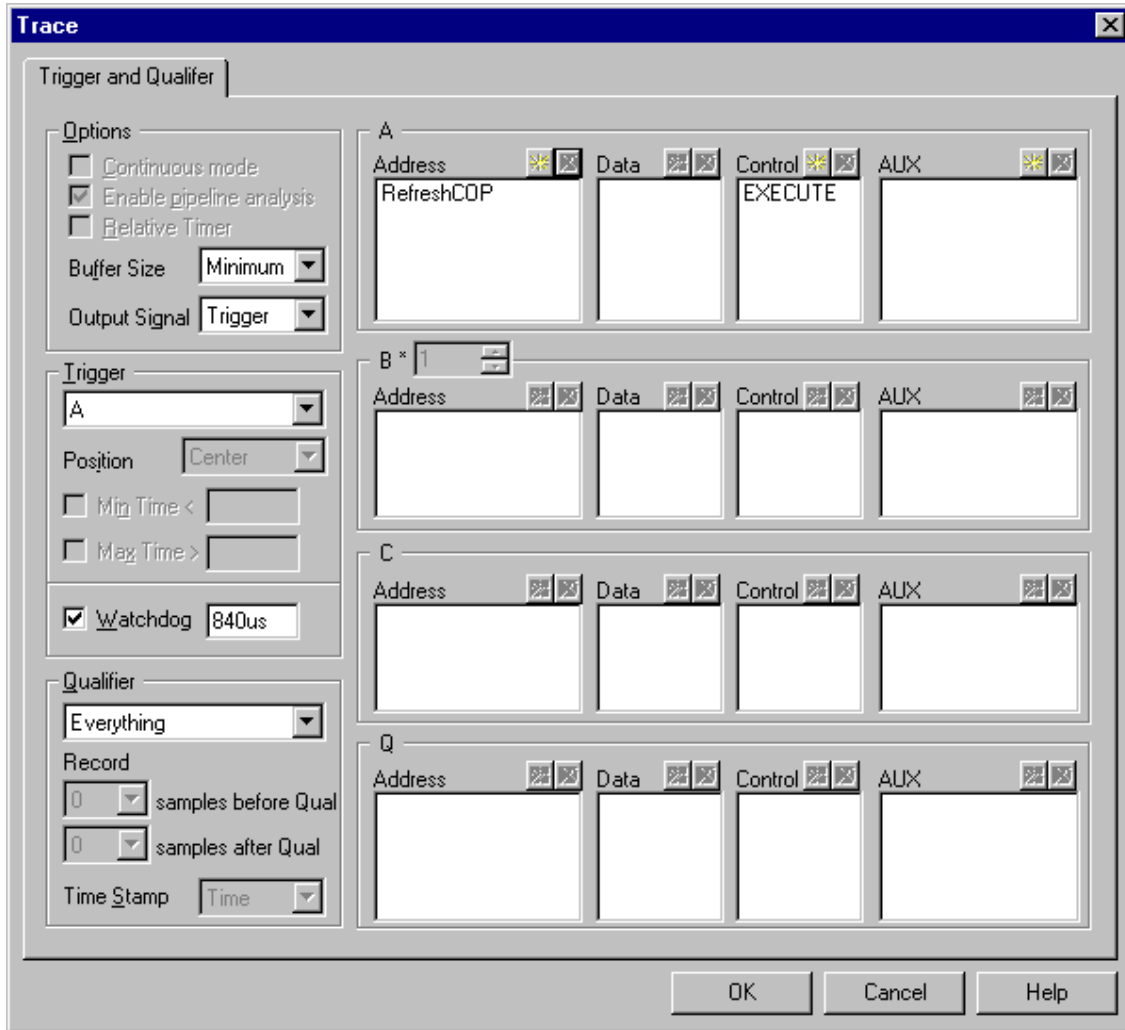
When COP is enabled, the program must call `RefreshCOP` routine during the selected time-out period. Once this is done, the internal COP counter resets to the start of a new time-out period. If the program fails to do this, the part will reset. The COP timer time-out period is 890 μ s in this particular 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.

- 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.

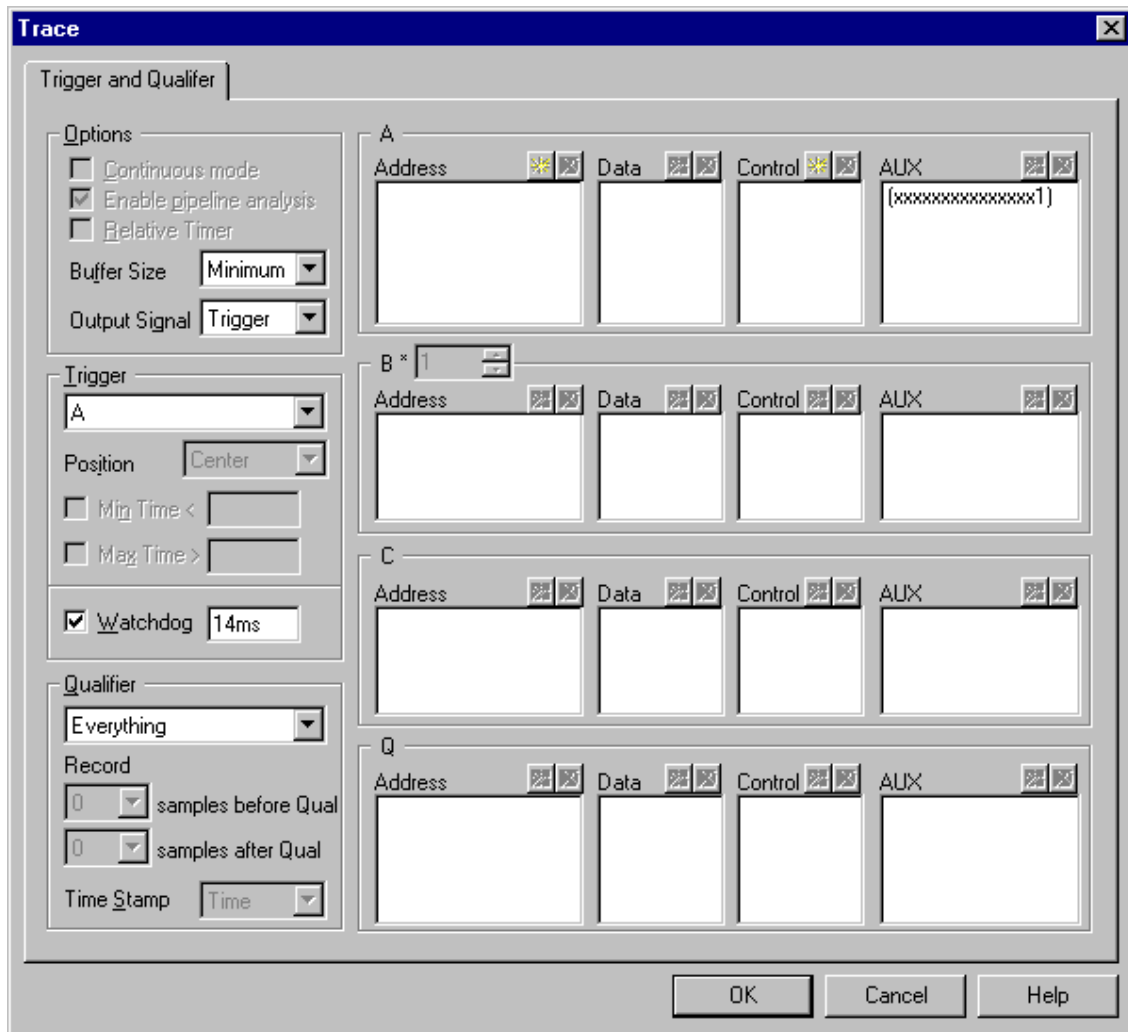
Example: The application features (external) target watchdog timer, which is normally periodically reset every 15 ms by the WDT_RESET target signal.

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.

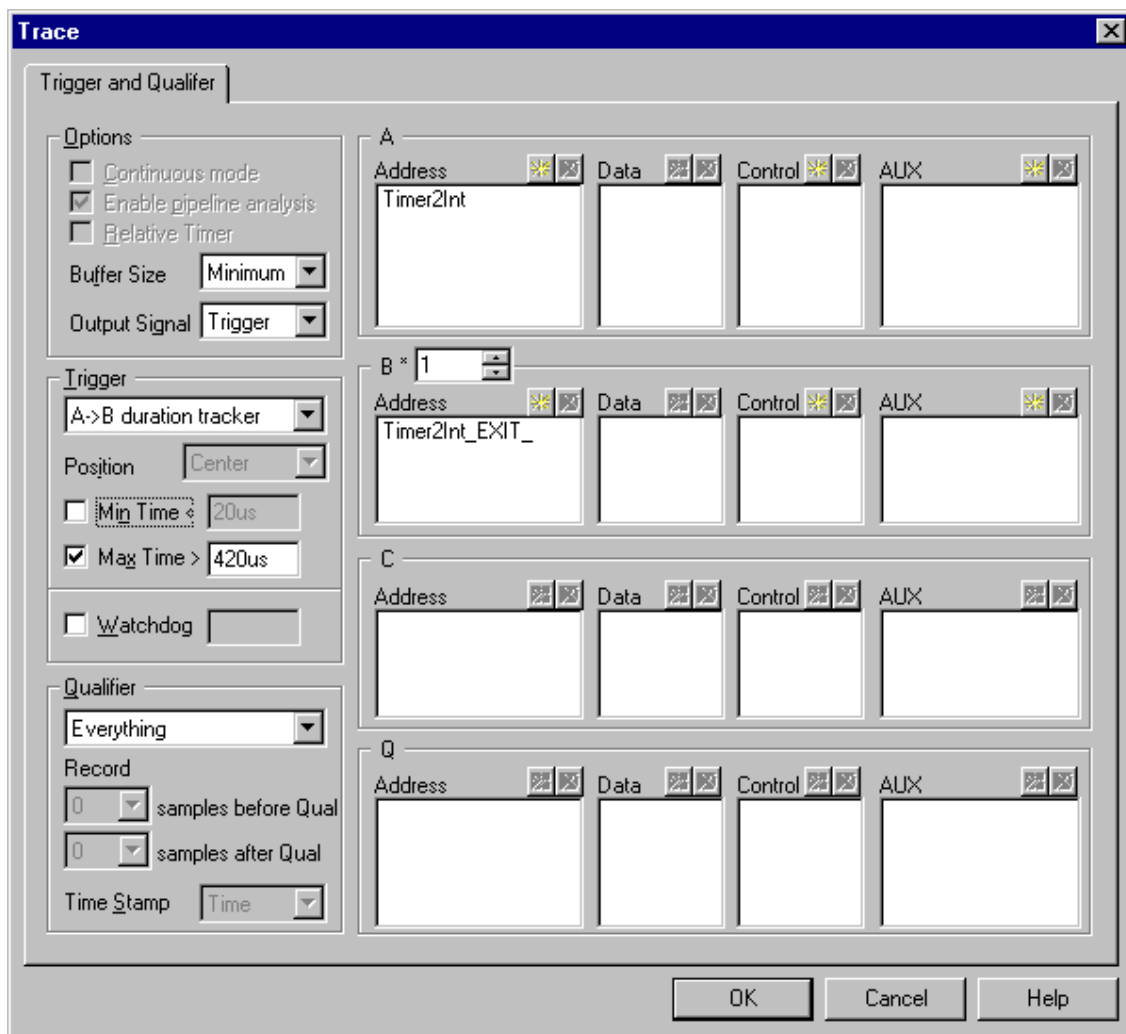
Min Time is compared with the Current Time as soon as the event A is detected or simply, Current Time is compared against Min Time as soon as the program enters the object being tracked.

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
T	0	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.

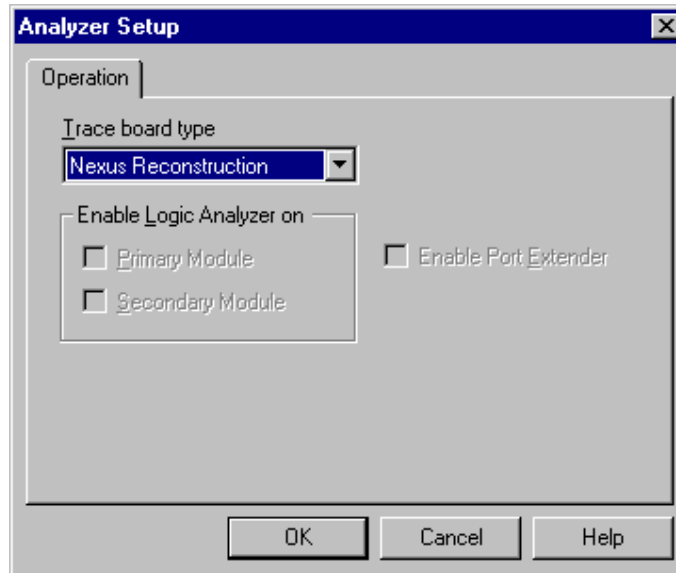
Note 2: Duration Tracker can be used in a way in which it works like the execution profiler (one of the analyzer operation modes) on a single object (e.g. function/routine) profiting two things, the results can be uploaded on the fly while the CPU is running and the object can be tracked over a long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

8 RTR Execution Coverage

RTR 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.

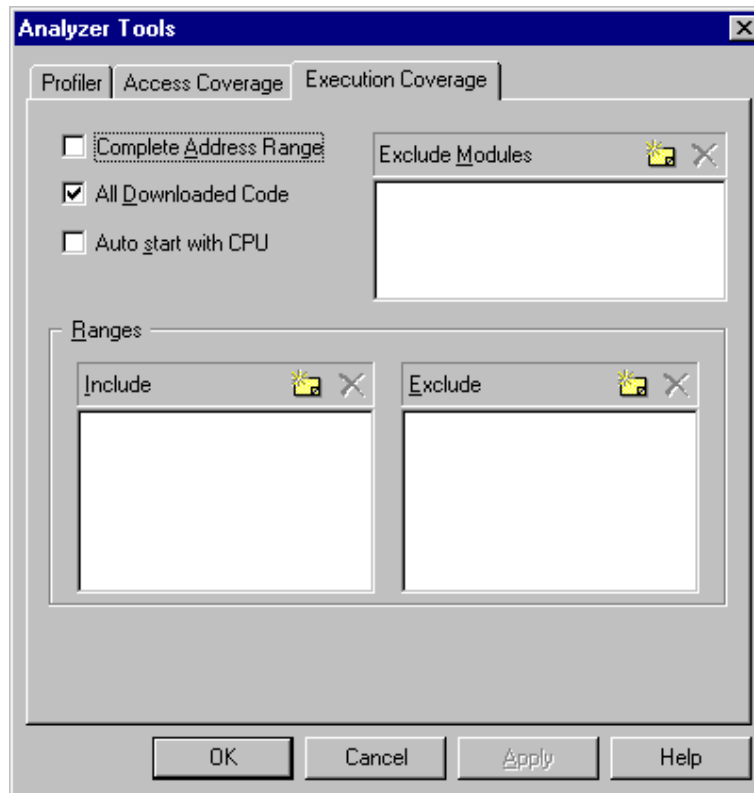
RTR Execution Coverage can cover 4MB (iTRACE PRO) CPU address space, which should cover all existing CRX applications. It can run infinite time, which means in practice that the application can run for days and then the execution coverage results can be analyzed.

Select ‘Nexus 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

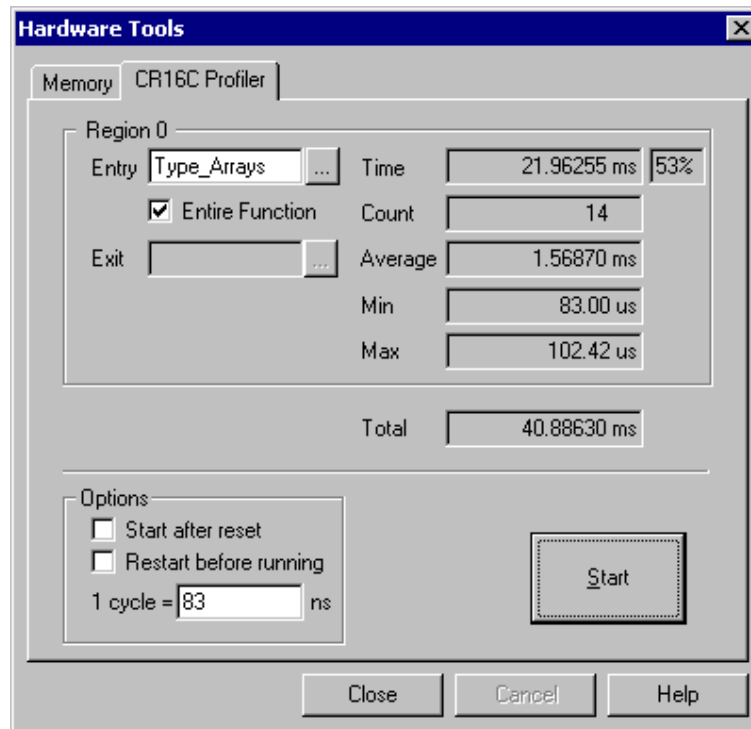
9 Profiler

The JTAG debugger supports on-chip profiler, which is restricted to a single memory area (e.g. function) and the development system supporting Nexus trace port features RTR Execution Profiler, which can profile unlimited number of application functions.

9.1 On-Chip Profiler

The On-Chip Profiler is invoked by the Hardware/Tools menu. Since this is a special on-chip feature of the CRX devices, it can not be shown in the regular Trace/Coverage/Profiler window.

On-chip profiler counts CPU cycles while the CPU executes the code within defined Region 0 area. The user needs to define a period of one CPU cycle in ns according to his target application. Note that profiler results may not be accurate since the profiler results rely on the user defined CPU cycle period. RTR Execution Profiler based on Nexus trace provides precise results.



CRX Profiler

Entry

Defines the address of region entry

Exit

Defines the address of region exit

Entire Function

If checked, the region Exit is set to function exit point.

Time

Shows the time spent in the region.

Count

Shows the number of entries in the region.

Average

Shows the average region execution time.

Min

Shows the minimum region execution time.

Max

Shows the maximum region execution time.

Total

Shows the total session run time.

Start after reset

If checked, the profiler is started automatically after the CPU is released from reset.

Restart before running

If checked, the profiler is restarted prior to any advancement of the execution point (run, step...)

1 cycle=

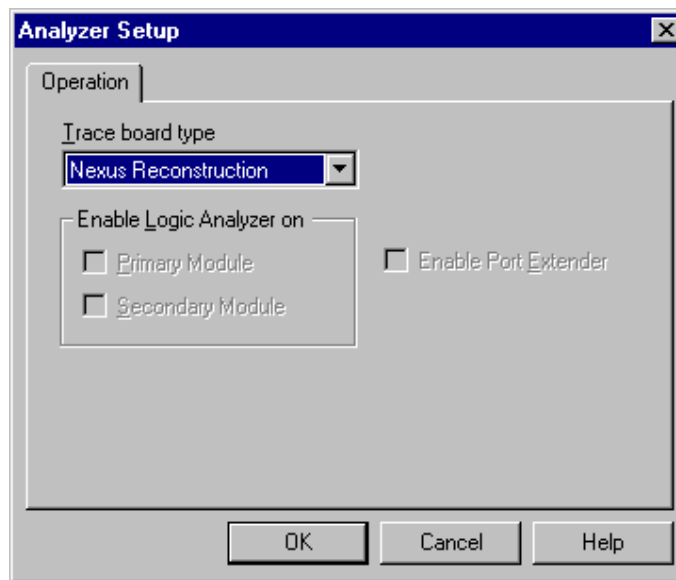
Defines the duration of one system clock cycle. If set to zero, the number of recorded system clocks is displayed.

9.2 RTR Execution 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.

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



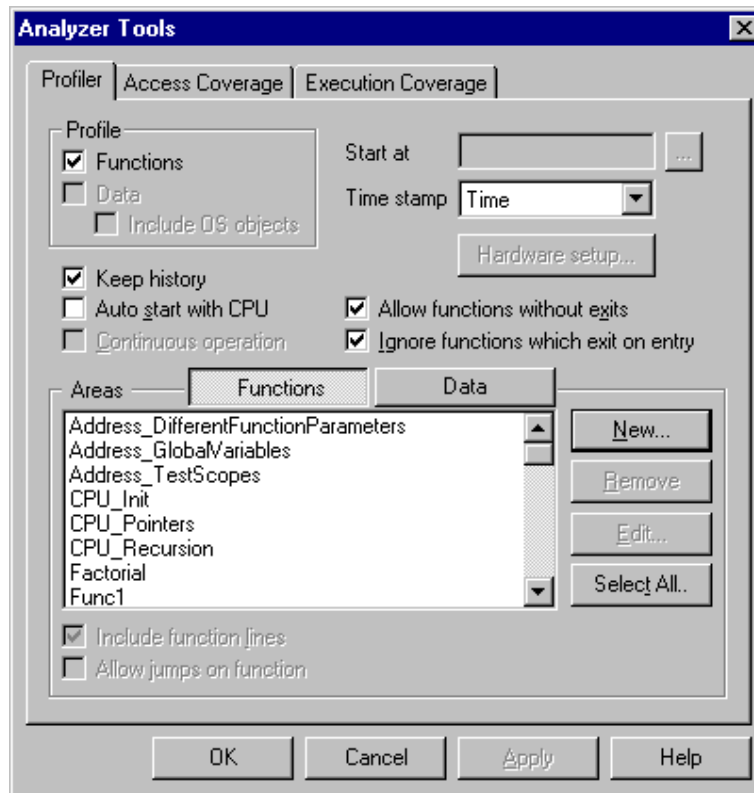
Next, select 'Profiler' window from the View menu and configure Profiler settings. Select 'Functions' option in the 'Profile' field.

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 software user's guide for more details on configuring Profiler and its use.

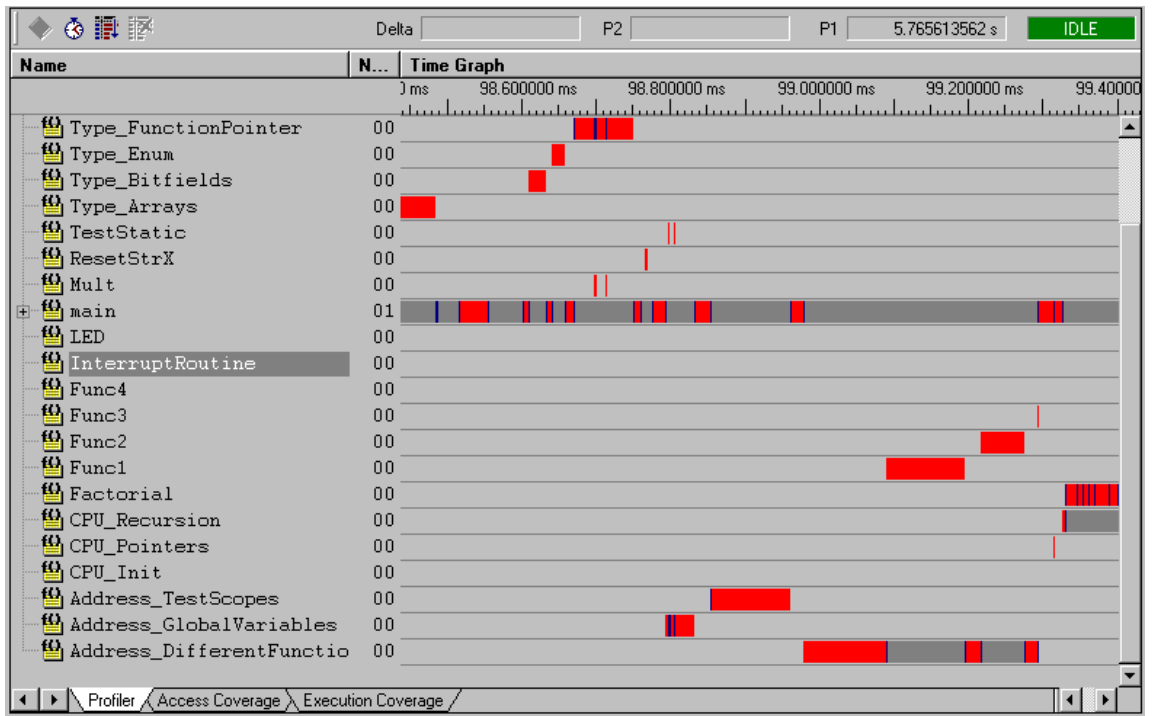


Profiler configuration settings

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

Name [All tasks]	Time	Percentage
Address_DifferentFunctionParameters	260.973462 ms	2.87%
Address_GlobalVariables	70.862445 ms	0.78%
Address_TestScopes	175.584016 ms	1.93%
CPU_Init	38 ns	0.00%
CPU_Pointers	127.875 us	0.00%
CPU_Recursion	24.584540 ms	0.27%
Factorial	147.500515 ms	1.62%
Func1	176.757068 ms	1.94%
Func2	109.125957 ms	1.20%
Func3	106.546 us	0.00%
Func4	0 ns	0.00%
InterruptRoutine	0 ns	0.00%
LED	208.957317 ms	2.30%
main	414.469779 ms	4.56%
Mult	383.625 us	0.00%
ResetStrX	468.880 us	0.01%
TestStatic	213.134 us	0.00%
Type_Arrays	717.613499 ms	7.89%
Type_Bitfields	46.236125 ms	0.51%
Type_Enum	34.167973 ms	0.38%
Type_FunctionPointer	144.078974 ms	1.58%
Type_Mixed	27.356283 ms	0.30%
Type_Pointers	58.063438 ms	0.64%
Type_Simple	6.394883106 s	70.35%
Type_Struct	78.169955 ms	0.86%

Profiler results – Code Statistics view



Profiler results – Code Statistics view

10 Getting Started

- 1) Connect the system
- 2) Make sure that the target debug connector pinout matches with the one requested by a debug tool. If it doesn't, make some adaptation to comply with the standard connector otherwise the target or the debug tool may be damaged.
- 3) Power up the emulator and then power up the target.
- 4) Execute debug reset
- 5) The CPU should stop on location to which the reset vector points
- 6) Open memory window at internal CPU RAM location and check whether you are able to modify its content.
- 7) If you passed all 6 steps successfully, the debugger is operational and you may proceed to download the code.

11 Troubleshooting

When performing any kind of checksum, remove all software breakpoints since they may impact the checksum result.

Make sure that the power supply is applied to the target JTAG connector when 'Target VCC' is selected for Debug I/O levels in the Hardware/Emulator Options/Hardware tab, otherwise emulation fails or may behave unpredictably.

Try 'Slow' JTAG Scan speed if the debugger cannot connect to the CPU.

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.