
Technical Notes

Freescale MPC5xx & MPC8xx On-Chip Emulation

Contents

Contents.....	1
1 Introduction	2
2 Emulation Options.....	3
2.1 Hardware Options.....	3
2.2 Initialization Sequence	4
3 CPU Setup	6
3.1 General Options.....	6
3.2 Debugging Options.....	7
3.3 Advanced Options	9
3.4 Exceptions	10
4 Access Breakpoints	11
5 MPC56x Nexus Trace	12
5.1 Nexus Trace Configuration	13
5.1.1 Trigger Configuration.....	13
5.1.2 Qualifier Configuration	15
5.2 Trace Troubleshooting Scenarios	15
5.2.1 Nexus Trace.....	15
5.2.2 Nexus RTR Trace	22
6 Profiler.....	37
7 Execution Coverage.....	41
8 Getting Started.....	43
9 Troubleshooting.....	43

1 Introduction

The MPC5xx/8xx use a BDM style debug interface. A Nexus interface on the MPC56x provides faster development access and, more importantly, instruction and data trace.

The debug interface of the PowerArchitecture MPC56x core uses the BDM or Nexus development port, which is a dedicated port that needs none or minimal of the regular system interfaces. System activity can be controlled from the development port when the core is in debug mode. The development port is a relatively inexpensive interface that allows the development system to operate at a lower frequency than the core's frequency (except for Nexus that operates at the core's frequency). In debug mode, the core fetches all instructions from the development port. Data can be read from or written to the development port. This allows memory and registers to be read and modified by a development tool (emulator) connected to the development port.

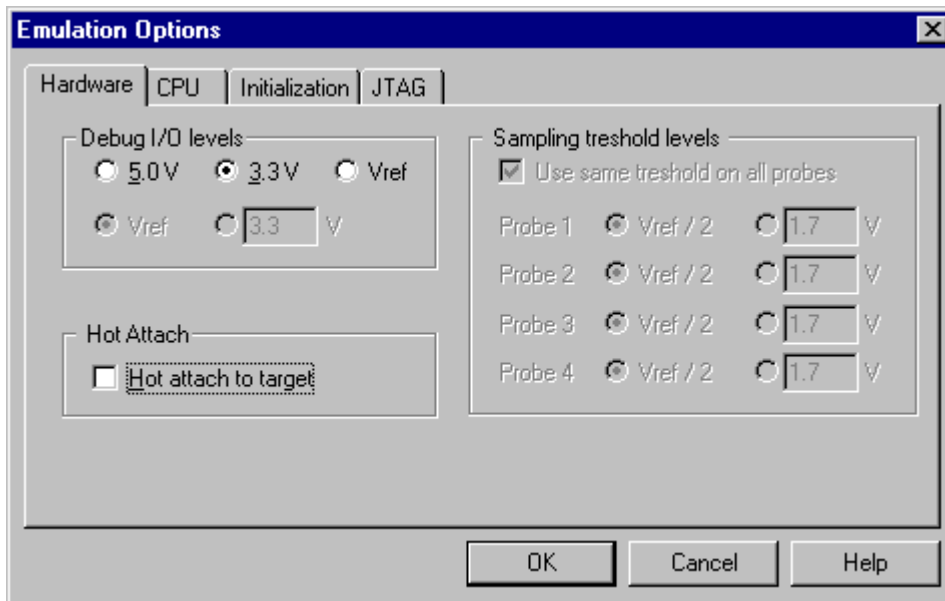
The Nexus/READI module interfaces to the MPC56x processor and internal buses to provide development support as per the IEEE-ISTO 5001 - 1999. The development features supported are program trace, data trace, watchpoint trace, ownership trace, run-time access to the MCU's internal memory map, and access to RCPU internal registers during halt, via the Nexus auxiliary port.

Debug Features

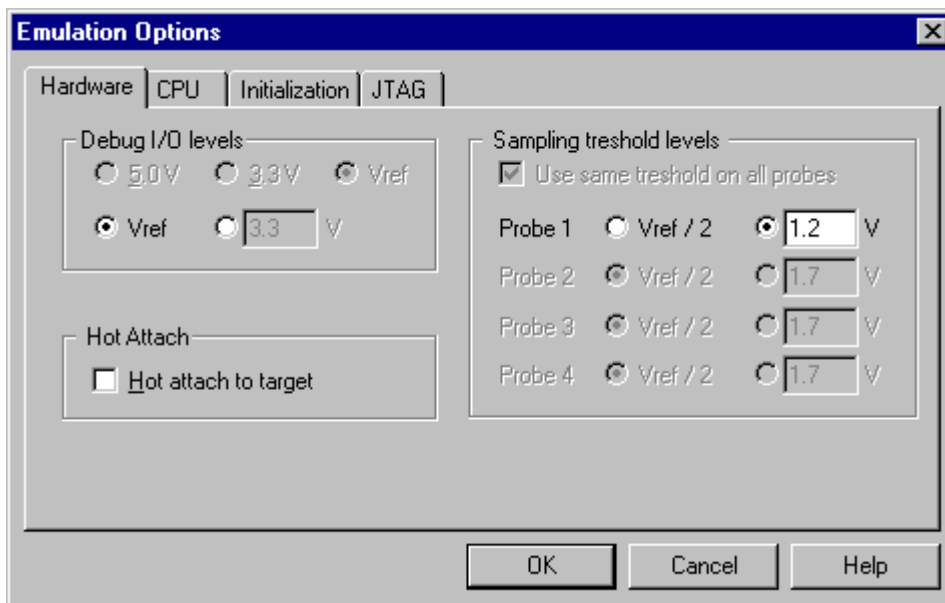
- 4 hardware breakpoints
- Access breakpoints
- Unlimited software breakpoints
- Fast flash programming
- Real-time memory access (MPC56x Nexus port only)
- Trace (MPC56x)
- Profiler (MPC56x)
- Execution Coverage (MPC56x)

2 Emulation Options

2.1 Hardware Options



Emulation options, Hardware pane (Debug iCARD)



Emulation options, Hardware pane (iTRACE, iTRACE PRO, iTRACE GT)

Debug I/O levels

The development system can be configured in a way that the debug BDM signals are driven at 3.3V, 5V or target voltage level (Vref).

When 'Vref' Debug I/O level is selected, a voltage applied to the belonging reference voltage pin on the target debug connector is used as a reference voltage for voltage follower, which powers buffers, driving the debug BDM signals. The user must ensure that the target power supply is connected to the Vref pin on the target BDM connector and that it is switched on before the debug session is started. If these two conditions are not meet, it is

highly probably that the initial debug connection will fail already. However in some cases it may succeed but then the system will behave abnormal.

Sampling threshold levels (iTRACE PRO/GT only)

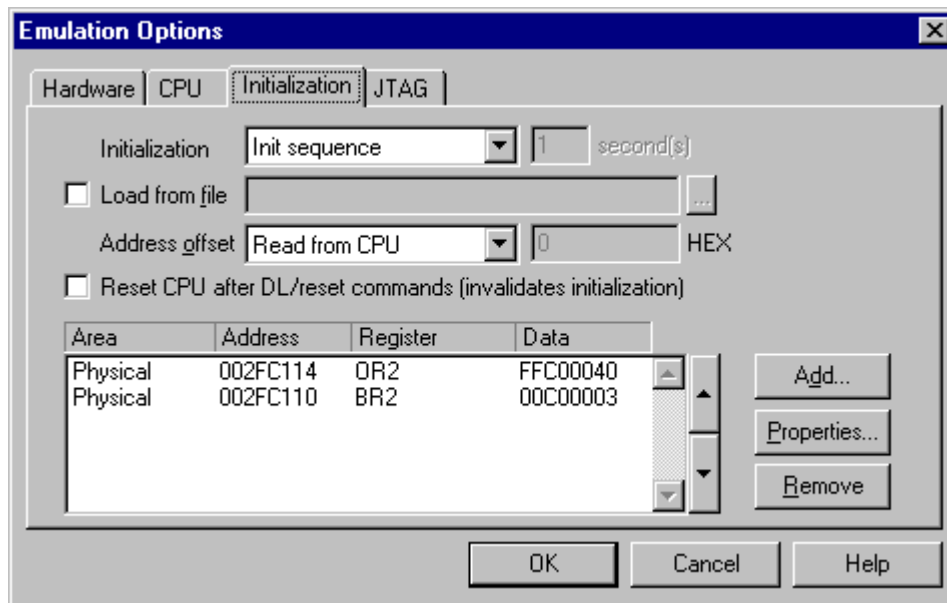
Voltage levels of the debug input and output signals are set according to the setting.

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.

The initialization sequence can be set up in two ways:

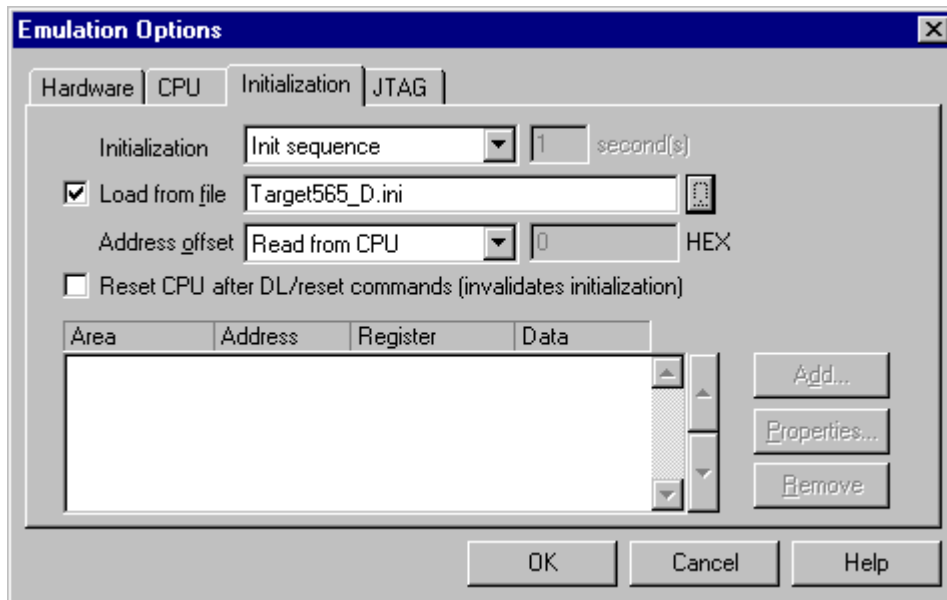
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



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.

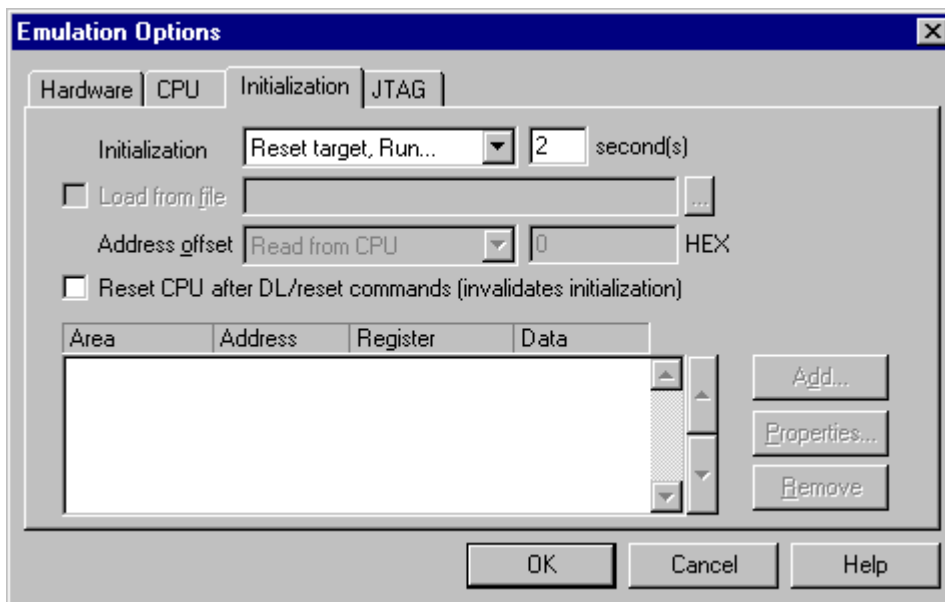
Excerpt from example Target565_D.ini file for a Freescale MPC565 based target:

```
S OR2 L FFC00040 // CS2 external FLASH, 4WS  
S BR2 L 00C00003 // 4MB, 32b, U13..14
```



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.

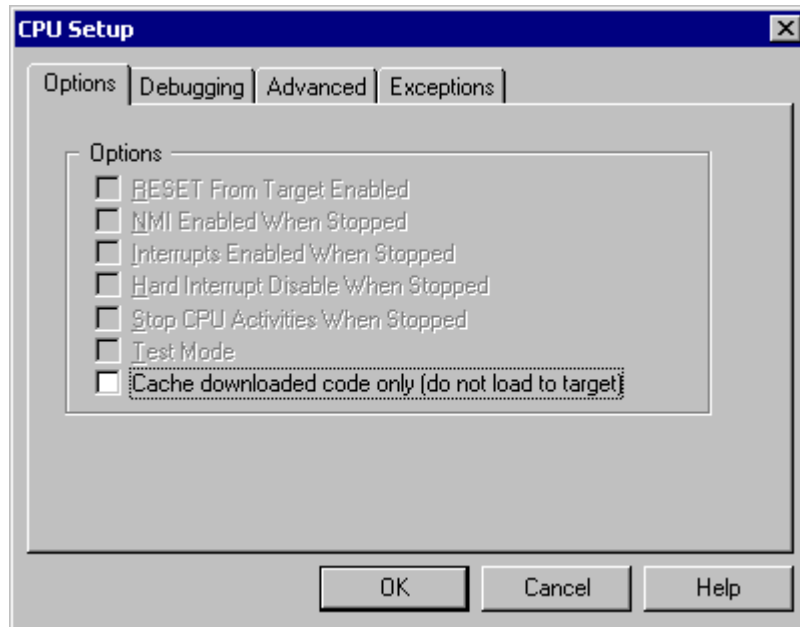
There is also a third method, which can be used too but it's not highly recommended for the start up. The user can initialize the CPU by executing part of the code in the target ROM for X seconds by using 'Reset and run for X sec' option.



3 CPU Setup

3.1 General Options

The CPU Setup, Options page provides some emulation settings, common to most CPU families and all emulation modes. Settings that are not valid for currently selected CPU or emulation mode are disabled. If none of these settings is valid, this page is not shown.



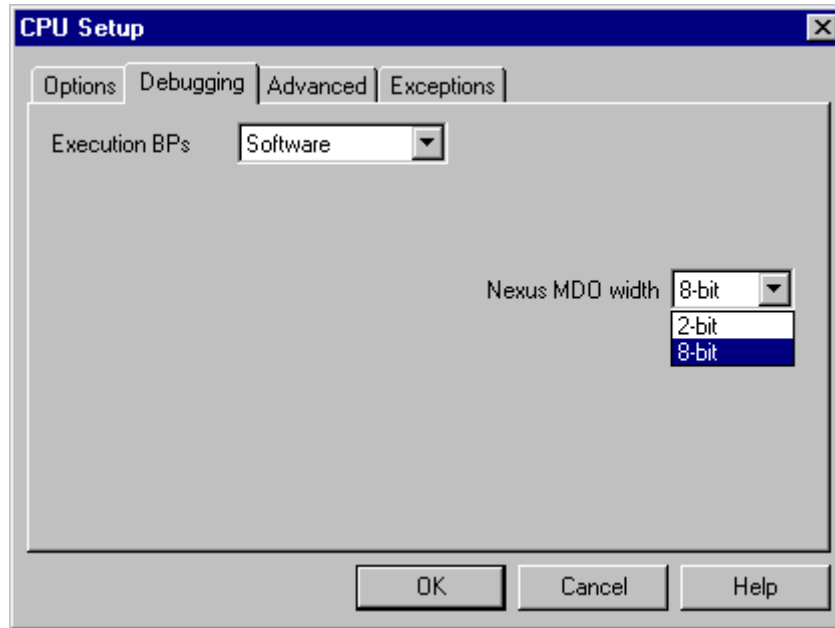
General Options

Cache downloaded code only (do not load to target)

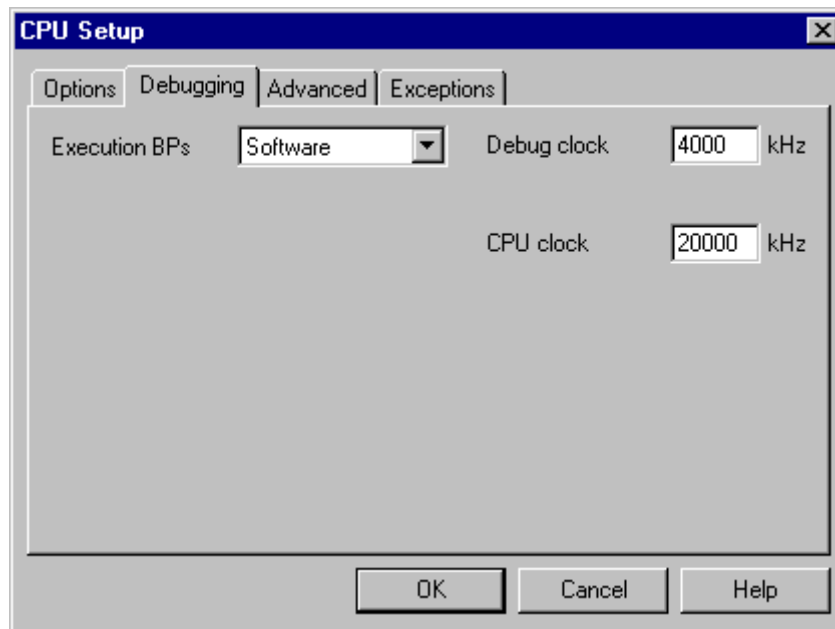
When this option is checked, the download files will not propagate to the target using standard debug download but the Target download files will.

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

3.2 Debugging Options



iTRACE PRO/GT Debugging Options



Debug iCARD Debugging Options

Execution Breakpoints

Hardware Breakpoints

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

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

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.

Nexus MDO width

This setting is available only for the development system supporting Nexus on-chip trace.

Nexus signals are located on the CPU pins, which can be configured for different alternate functions. The CPU allows configuring Nexus MDO port either as 2 or 8-bit port. An 8-bit MDO implementation ensures optimum Nexus operation. A 2-bit MDO implementation requires less CPU signals than the 8-bit MDO but the Nexus throughput is decreased, which is a crucial factor for correct trace operation. Note that the trace displays errors when the CPU doesn't manage to send out complete Nexus messages to the external development system. It's highly probably that 2-bit MDO port will result in trace errors while 8-bit MDO port will function flawlessly.

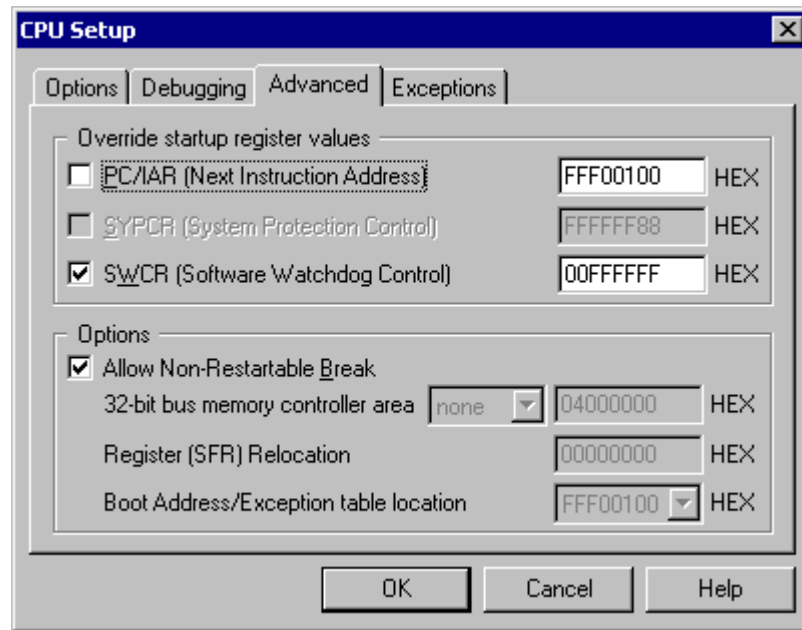
Debug Clock

Debug clock frequency must be selected between 4 and 16MHz. Note however, that the clock generation is using a discrete divider from a 48MHz source. This, for example, means that selected 10MHz clock will in fact be set to 9.6MHz. Choose the debug clock frequency wisely, especially at power-on. Typically, CPUs wake up at some lower system frequency that is then switched up with application software or via initialization script. In general, the debug clock frequency must be no greater than 1/3 of the system frequency. Note that MPC509 with 4MHz clock source wakes up at 3MHz system clock. Set the debug clock to 1MHz.

CPU Clock

This setting applies for MPC555 device only. A special divider is present on the CPU to ensure the correct programming frequency for the FLASH module, which is generated from the CPU clock. Since the CPU clock cannot be automatically detected, it must be set in this dialog. Note that this setting affects internal FLASH programming only.

3.3 Advanced Options



Advanced Options

Override Startup Register Values

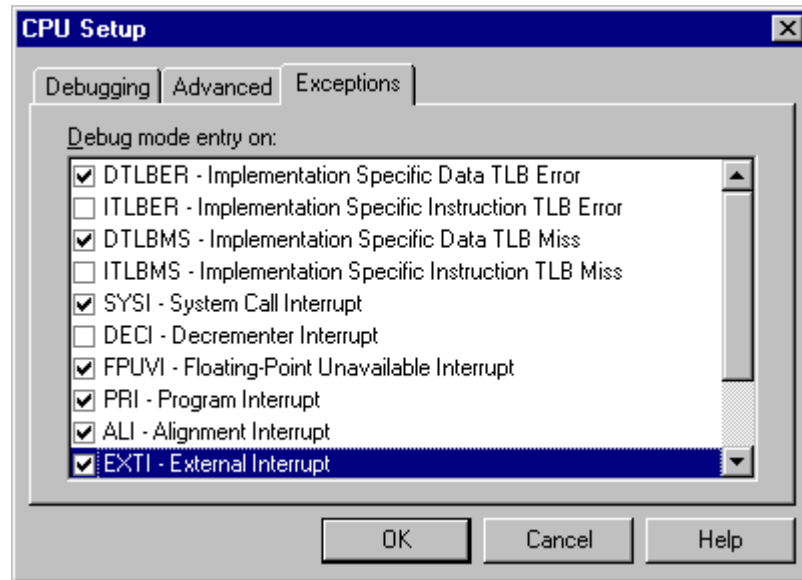
If required, the Emulator can change the checked register after the CPU is released from reset. These settings have typically to do with setting the default program starting point and disabling the watchdog.

Allow Non-Restartable Break

In general, stopping the processor is possible and breakpoints are recognized in the core only when the RI bit in the MSR register is set, which guarantees machine restartability after a breakpoint or stop. In this working mode, breakpoints are masked. There are times when it is preferable to enable breakpoints even when the RI bit is clear, even though there is a risk of causing a non-restartable machine state. Checking this option, for example, allows stopping the CPU from an endless loop, even if the RI bit is clear.

For more info on the RI bit, please see the User Manual of your processor.

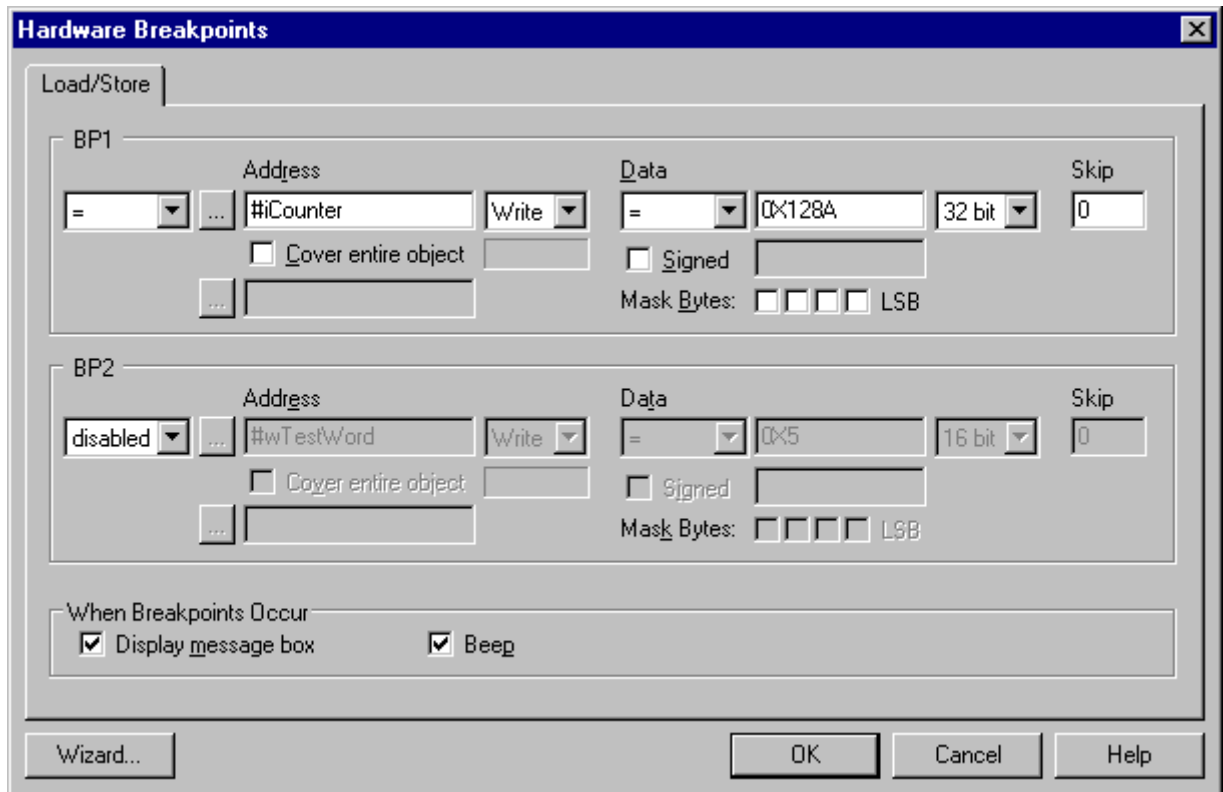
3.4 Exceptions



Exceptions menu

The 5xx/8xx BDM debug unit allows the debug mode (the processor is stopped) to be entered upon the selected CPU exceptions.

4 Access Breakpoints



Hardware Breakpoints

The debug interface of the MPC5xx/8xx families include two L-Address Comparators and two L-Data Comparators Supporting Equal, Not Equal, Greater Than, and Less Than. The comparators can be used in several different ways:

- 2 address-only compares
- 2 data-only compares
- 2 address & data compares
- 1 address range (inside or outside)
- 1 data range (inside or outside)
- 1 address & data ranges

The following is a brief walkthrough of the configurable items.

First, select a desired address comparison type. Supported are: disabled, =, <, >, !=, Inside, Outside. Select the address from the list (the '.' button) or type the number in directly. Use the 0x prefix for hexadecimal numbers. For arrays you can flag the Cover entire object option. The address comparison type will be automatically set to Inside. Next, select the access type, read, write, R/W for read or write.

For data, identical comparison types are supported: disabled, =, <, >, !=, Inside, Outside. Check the Signed checkbox to treat fixed-point numbers as signed values. Select the access width, 8, 16, or 32-bit. Check the Mask Bytes checkboxes to ignore selected bytes in a data comparison. The included two event counters (Skip) further extend above capabilities.

Please refer also to the Development Capabilities and Interface section of the respective CPU manual.

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

Wizard...

Use Wizard (button in the left bottom corner) in case of problems understanding and configuring the access breakpoints dialog. It helps setting simple a breakpoint on data access.

5 MPC56x Nexus Trace

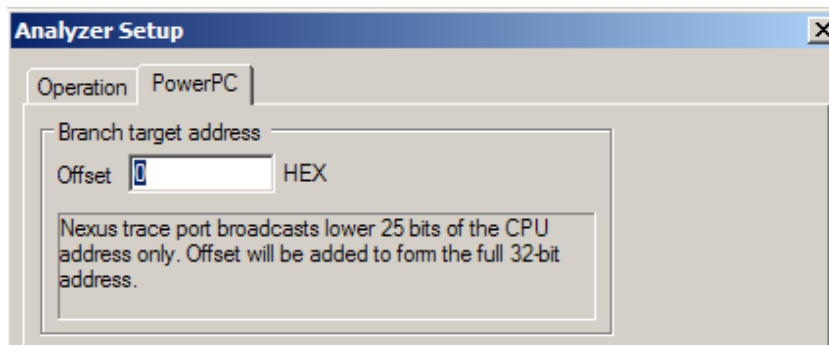
Nexus is a message based asynchronous protocol. In contrast to an In-Circuit Emulation where all buses are visible to the emulator and allow implementation of various analyzer features, a Nexus based tool is limited by the information provided through the Nexus port.

For tracking the sequence of program instructions, the Nexus port broadcasts only information related to instructions that cause a change to the normal sequential execution of instructions. With knowledge of the source code, which is programmed in the CPU flash, the debugger can reconstruct the path of execution through many instructions from the recorded change-of-flow information.

The MPC56x CPUs provide Nexus 2+ level on-chip trace.

Nexus Trace features (iTRACE PRO / GT):

- Compliant with Nexus standard
- External trace buffer
- Instruction, Data and OTM Trace
- On-chip trigger and qualifier
- Advanced external trigger and qualifier
- Time Stamps
- AUX inputs
- Profiler
- Execution Coverage



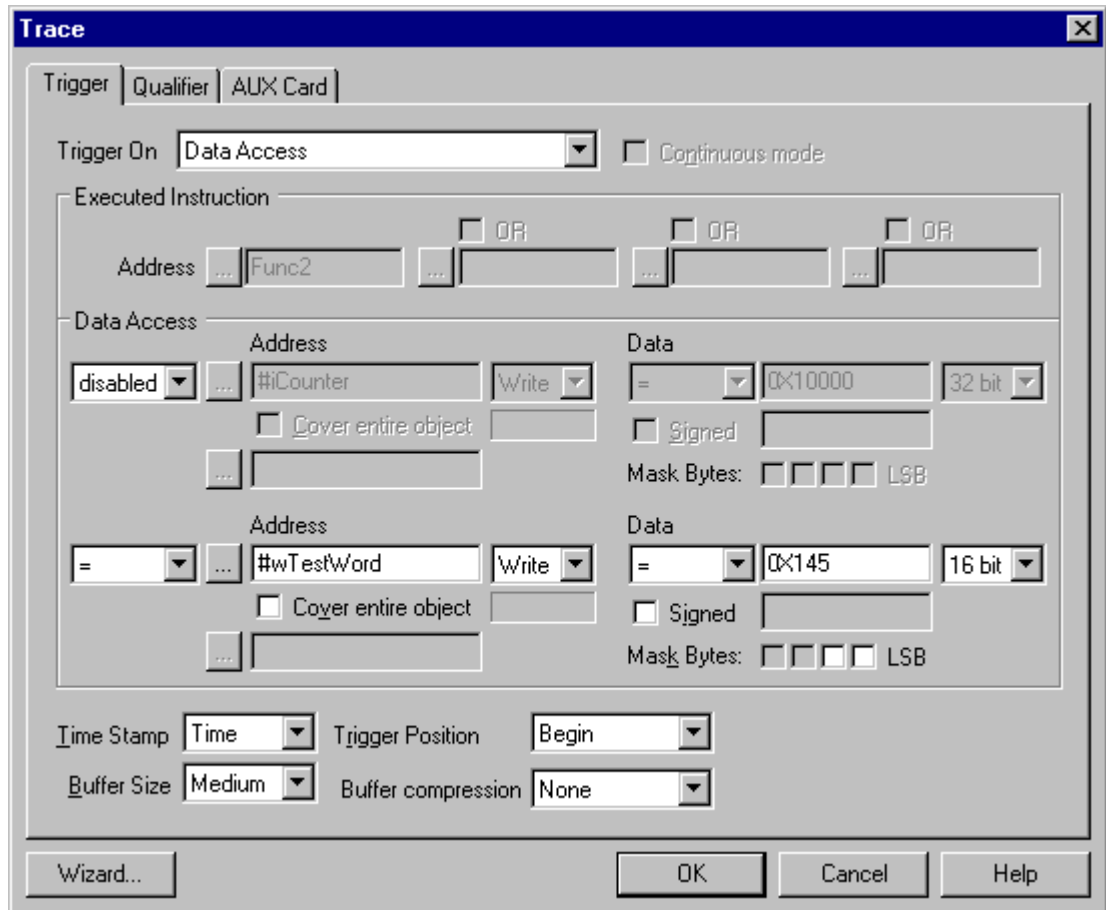
MPC56x Nexus doesn't report full 32-bit instruction address but only lower 25-bits. It is necessary to specify the address offset in the 'Hardware/Analyzer Setup/PowerPC' tab when the code runs at offset address exceeding the 25-bit address space. For instance, if the address offset is not set properly (e.g. set to 0), the profiler, trace

and execution coverage will work correctly while running at one address (e.g. 0x800000) and incorrectly when the code is running at e.g. 0xFFFF0000.

Nexus doesn't report the size of the data transfer. Instead it is guessed according to the number of bits transferred in the data field but this may not work always. As an example when the algorithm fails, 16-bit data access is displayed as 8-bit access.

5.1 Nexus Trace Configuration

5.1.1 Trigger Configuration



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.

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

With each 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 trigger are marked with negative values; samples after trigger are marked with positive values. If you select a clock cycle then the CPU cycle counter is recorded.

The Trigger condition can be simple or complex. Select 'Anything' if you want recording to begin immediately. When the trace buffer is filled the recording will stop.

In continuous mode the recording is stopped only after CPU has been stopped or the user has manually selected stop recording in the trace window.

It is not possible to set an arbitrary trigger condition. The MPC5xx BDM/Nexus has limited resources that are common to trigger and hardware breakpoints. This means that if too many resources have been used for the breakpoints less will be available in the trigger and vice versa.

Select 'Execute instruction' to trigger when CPU executes instruction from the address defined in the Address field.

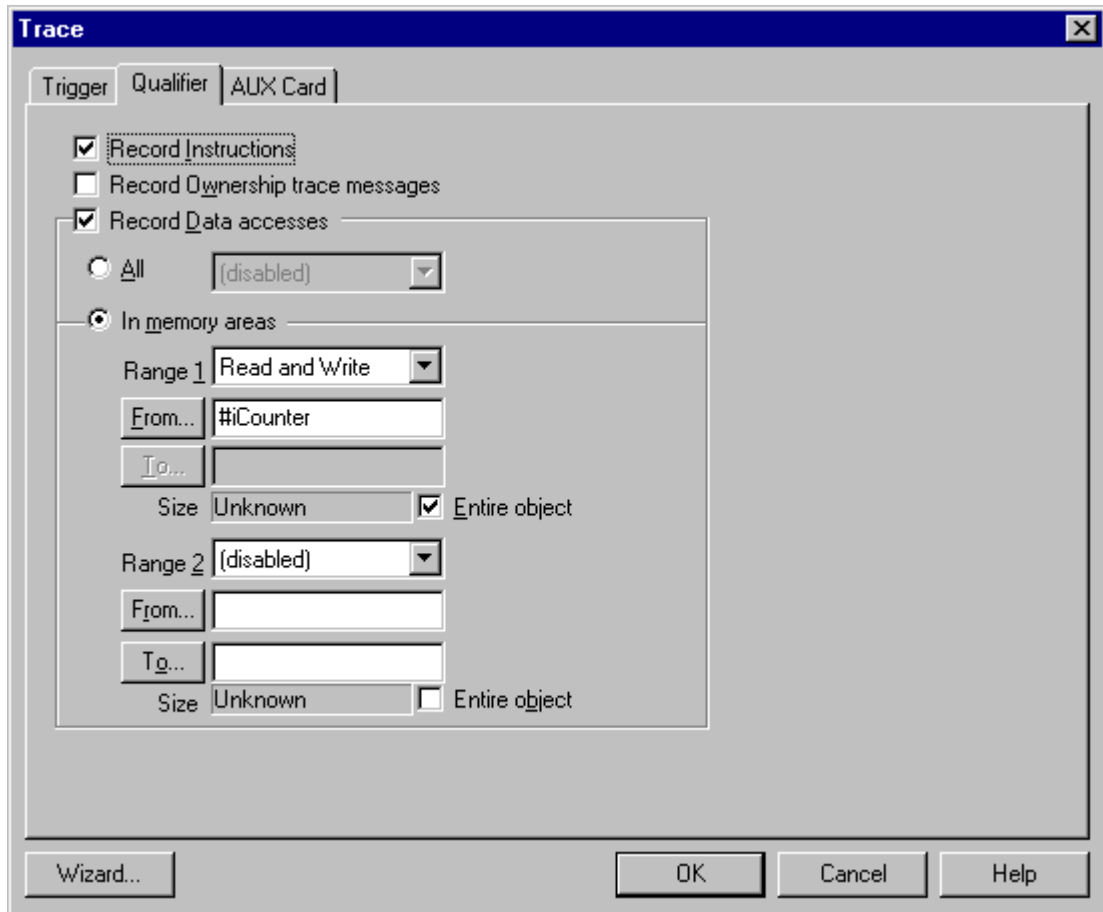
Trigger may also be set on a specific 'Data access'. The debug/trace interface of the MPC56x family include two L-Address Comparators and two L-Data Comparators Supporting Equal, Not Equal, Greater Than, and Less Than. The comparators can be used in several different ways:

- 2 address-only compares
- 2 data-only compares
- 2 address & data compares
- 1 address range (inside or outside)
- 1 data range (inside or outside)
- 1 address & data ranges

First, select a desired address comparison type. Supported are: disabled, =, <, >, !=, Inside, Outside. Select the address from the list (the '...' button) or type the number in directly. Use the 0x prefix for hexadecimal numbers. For arrays you can flag the Cover entire object option. The address comparison type will be automatically set to Inside. Next, select the access type, read, write, R/W for read or write.

For data, identical comparison types are supported: disabled, =, <, >, !=, Inside, Outside. Check the Signed checkbox to treat fixed-point numbers as signed values. Select the access width, 8, 16, or 32-bit. Check the Mask Bytes checkboxes to ignore selected bytes in a data comparison.

5.1.2 Qualifier Configuration



The qualifier may be set only for data accesses. It may be set to None, All or in selected memory areas.

Option 'None' effectively disables the Data Trace, thus only execution trace is enabled.

With 'All' all read-write, read-only or write-only data accesses are traced. Please use this option with caution, because Nexus Data trace FIFO inside the CPU may quickly overflow.

Using the third option 'In memory areas' it is possible to narrow down the data access recording by defining two memory areas. In the above example, one area for global variables, and the other for stack trace.

5.2 Trace Troubleshooting Scenarios

5.2.1 Nexus Trace

'Nexus' must be selected in the 'Hardware/Analyzer Setup' dialog.

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

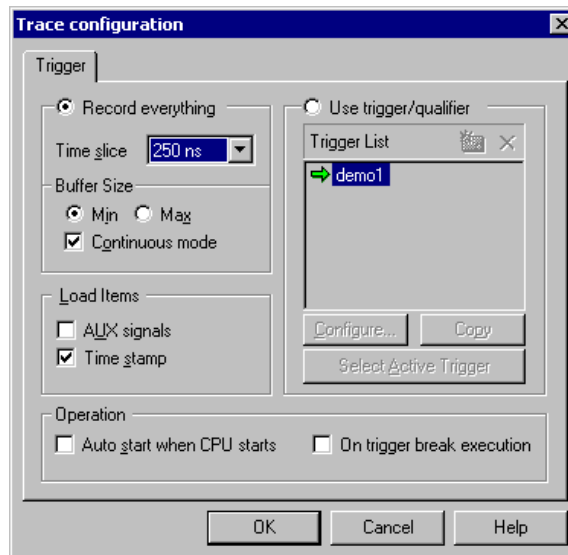
The trace can start recording on the initial program start from the reset or after resuming the program from the breakpoint location. The trace records and displays program flow from the start until the trace buffer fulfills.

As an alternative, the trace can stop recording on a program stop. 'Continuous mode' allows roll over of the trace buffer, which results in the trace recording up to the moment when the application stops. In practice, the trace

displays the program flow just before the program stops, for instance, due to a breakpoint hit or due to a stop debug command issued by the user.

Example: Trace records the CPU execution until the CPU is stopped by either the user or the program itself in case of a problem originating from the application.

Open the *Trace configuration* dialog by pressing *Hardware Configuration* toolbar in the trace window and select 'Record everything' operation type.



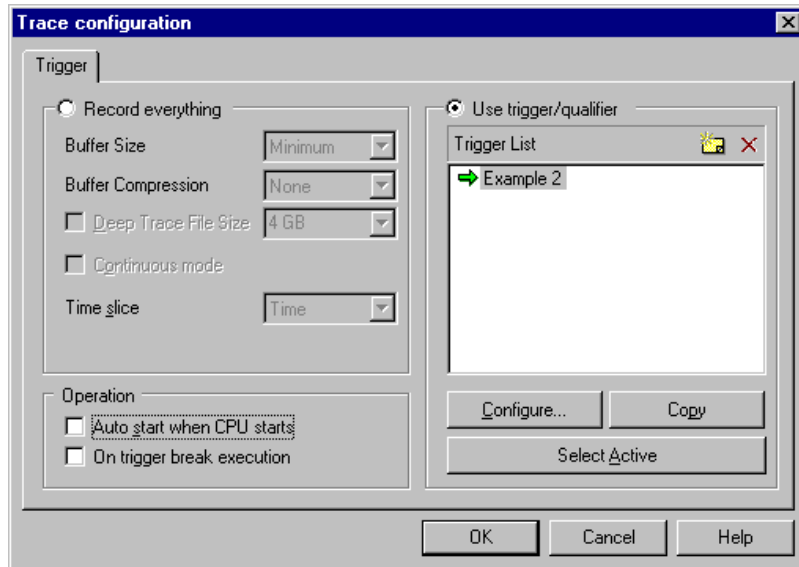
Select minimum buffer size and check continuous mode.

With these settings, the trace records program flow, while the user's program runs. After the user's program stops, trace stops recording and displays the program flow before the CPU was stopped.

Example: Trace triggers on a function Func3. Using trace, the program flow before the trigger event and after the trigger event can be inspected.

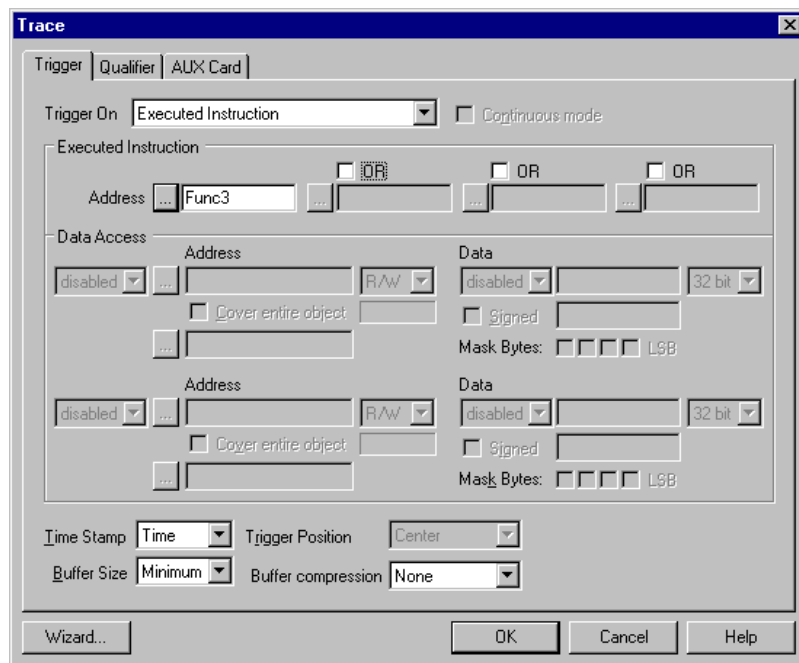
This is a very often used scenario. Trace triggers on a specific function or source line and the user can inspect the program execution before and after the trigger event.

Open *Trace configuration* dialog, select 'Use trigger/qualifier' operation type and define new trigger, named 'Example2'.

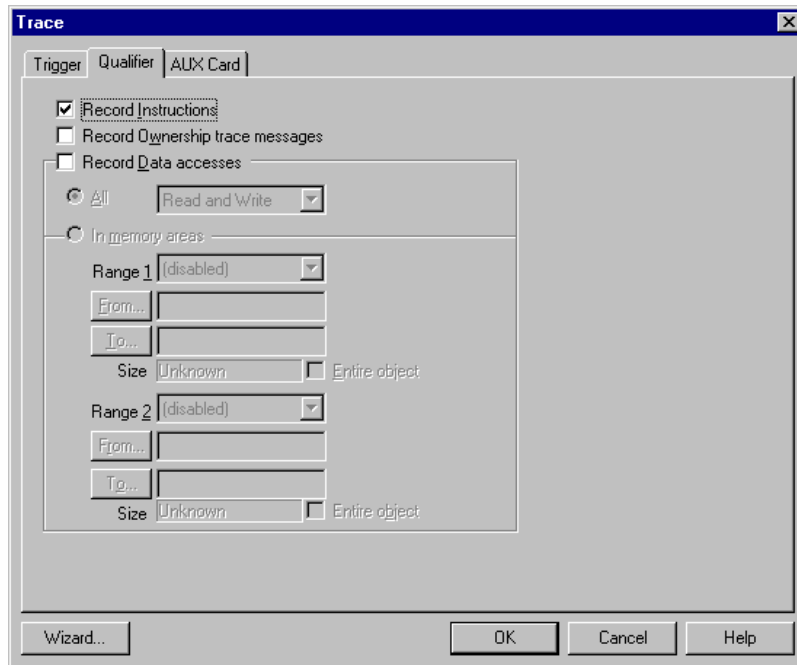


Open *Trigger* configuration dialog by pressing the ‘Configure...’ button and define a trigger event.

Select trigger on ‘Executed Instruction’ and enter the address.



Now, define a qualifier – condition, which defines the CPU cycles to be recorded by the trace. Select *Qualifier* tab and check ‘Record Instructions’. You may check ‘Record All Data accesses’ as well.



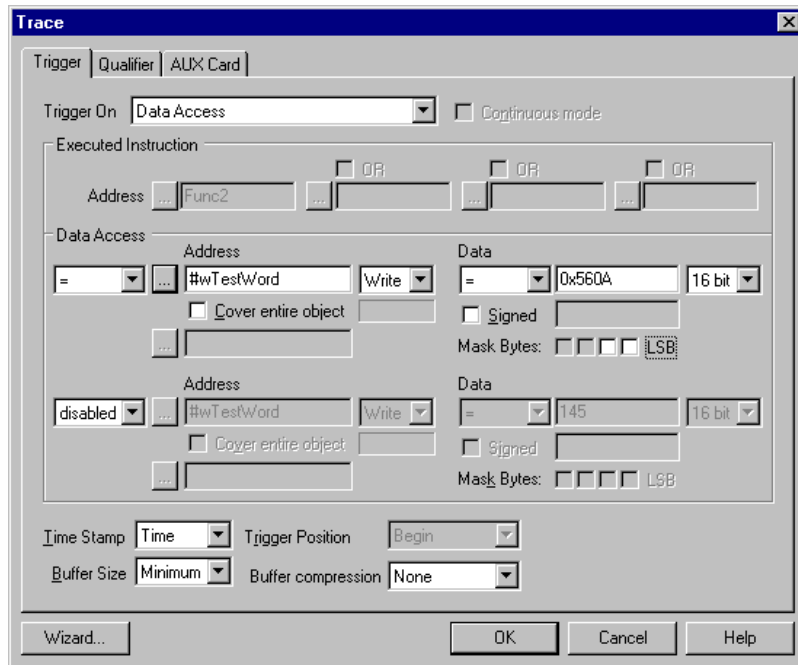
Now, execute *CPU reset*, start trace and run the program. After Func3 is executed for the first time, winIDEA™ displays program flow before and after the entry to function Func3.

Number	Address	Data	Content	Time
-4	00D00674	90610014	90610014 stw r3,14(r1)	-500 ns
-3	00D00678	39210014	pY=ey; 39210014 la r9,14(r1)	-500 ns
-2	00D0067C	61230000	Func3(pY); 61230000 mr r3,r9	-500 ns
-1	00D00680	4BFFFFA1	4BFFFFA1 bl Func3 (00D006	-500 ns
0	003FCFBC	00D00684		0 ns
1	003FCFCC	00000000		250 ns
2	00D00620	7C0802A6	{ Func3 7C0802A6 mflr r0	250 ns
3	00D00624	90010004	90010004 stw r0,04(r1)	250 ns
4	00D00628	39800000	*pY=0; 39800000 li r12,00	250 ns
5	00D0062C	91830000	91830000 stw r12,00(r3)	250 ns
6	00D00630	4E800020	} Func3_EXIT_ 4E800020 blr	250 ns
7	003F1004	00000004	iCounter	500 ns

Example: The user gets incorrect writes to a particular address/variable. This example demonstrates how to locate the code writing faulty value (0x560A) to a word variable wTestWord.

Open *Trigger* configuration dialog by pressing the ‘Configure...’ button and define a trigger event.

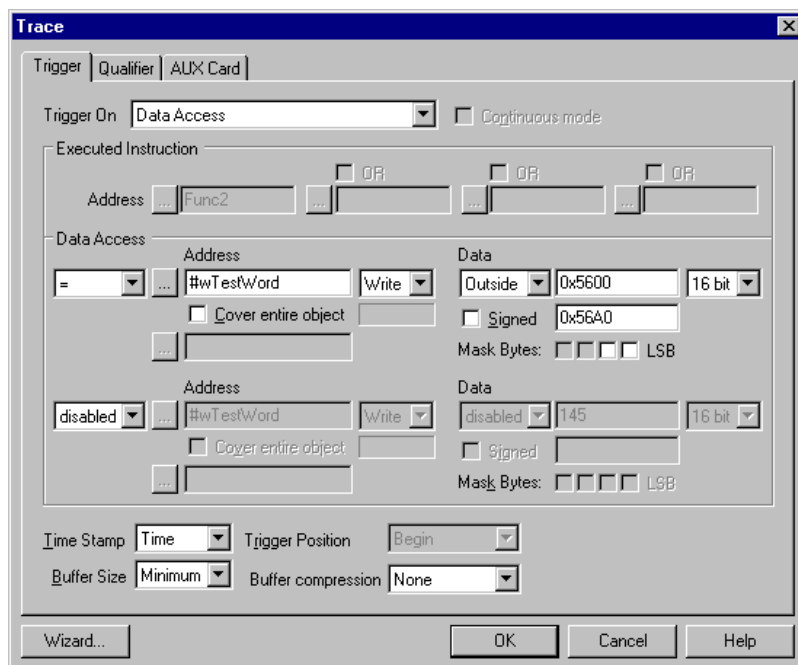
Select trigger on ‘Data Access’, enter the address, select write access type, enter 0x560A in the data field and select 16-bit data size. **Make sure you always set correct data size.** It must match with the variable size. Otherwise, trace may behave strangely.



Trigger set on a variable write “wTestWord==0x560A”

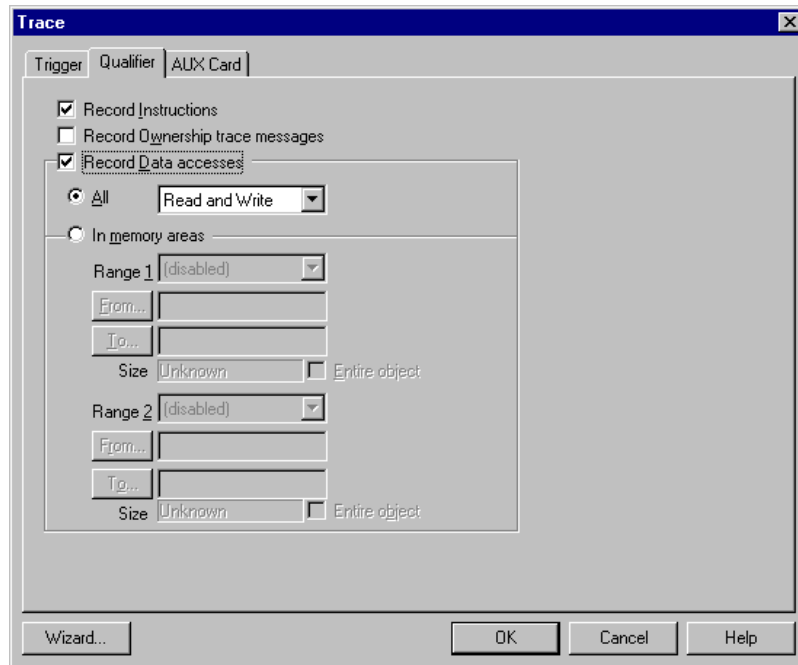
Below is a trigger configuration, where trace triggers on any write to the wTestWord, having different value than 0x5600 - 0x56A0.

Likewise, trigger can be set on any write to the wTestWord, having value within 0x5600 - 0x56A0 range by selecting ‘Inside’ item in the associated combo box.



Trigger set on a variable write “wTestWord<0x5600 or wTestWord>0x56A0 ”

Now, define a qualifier. Check ‘Record Instructions’ and ‘Record All Data accesses’.



Execute *CPU reset*, start the trace and run the program. When faulty write occurs, a trace trigger event occurs simultaneously. The trace buffer fills up and the program flow is reconstructed and displayed. Since pre-trigger history is visible, the user can easily locate the code writing 0x560A to the wTestWord variable and fix the problem.

Trigger event can be used as a breakpoint source as well. Check the 'On trigger break execution' option in the *Trace* configuration dialog. This may provide additional help to the user as he can inspect the target application after "problematic" trigger event, using debug windows (memory, watch, SFR, call stack, etc...).

Example: In this example, tracing of so-called Ownership trace messages will be explained. The MPC56x devices have a dedicated register, the Ownership Trace Register (OTR) located at address 0x38 002C. It is write-only register (by the application). The trace can then trace all write cycles to this specific register. In general the trace cannot trace accesses to any other internal CPU register.

Let's define OTR in our application as following:

```
#define OTR *(volatile unsigned long *)0x38002C
```

Now the user can write to the OTR anywhere in the application by simply adding following source line:

```
OTR = 0x5;
```

or

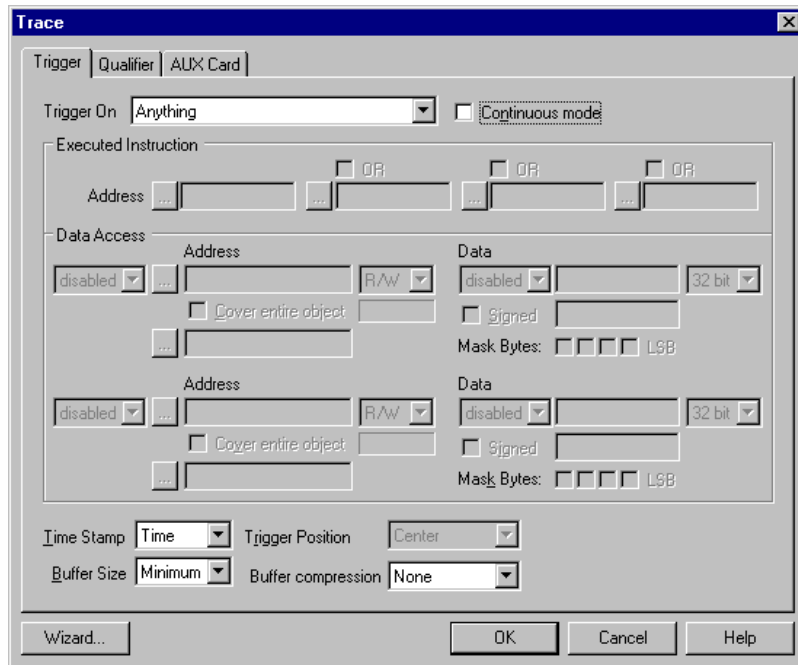
```
OTR = otrRecursionEntry;
```

Such add may help out the user while debugging his application without disturbing the program execution vitally. While the application writes various values to the OTR, the trace records and displays all OTR writes.

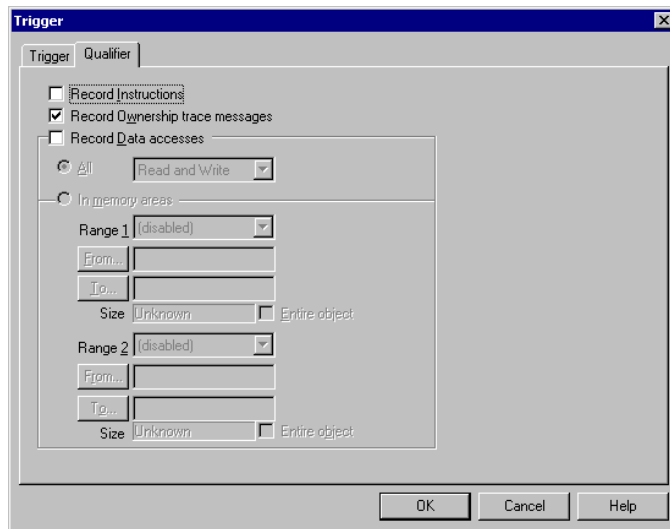
Open *Trace configuration* dialog, select 'Use trigger/qualifier' operation type and define new trigger, named 'Example4'.

Open *Trigger* configuration dialog by pressing the 'Configure...' button and define a trigger event.

Select trigger on 'Anything'.



Next, select 'Record Ownership trace messages' in the *Qualifier* tab. Additionally, the user may check 'Record Instructions' and/or 'Record Data accesses'.



Execute *CPU reset*, start the trace and run the program. All user's writes to the OTR are marked as OTM (Ownership Trace Messages) in the trace window. The address value at OTM cycles should be ignored.

Number	Address	Data	Content	Time
0	00000000	00000000	OTM	0 ns
1	00000000	00000001	OTM	5.00 us
2	00000000	00000002	OTM	6.50 us
3	00000000	00000003	OTM	15.00 us
4	00000000	00000004	OTM	16.00 us
5	00000000	00000005	OTM	66.00 us
6	00000000	00000008	OTM	131.00 us
7	00000000	00000009	OTM	141.25 us
8	00000000	00000001	OTM	143.25 us
9	00000000	00000002	OTM	144.75 us
10	00000000	00000003	OTM	153.00 us

P: 2.07725 ms D: -2.07725 m: M2: - M1: - IDLE

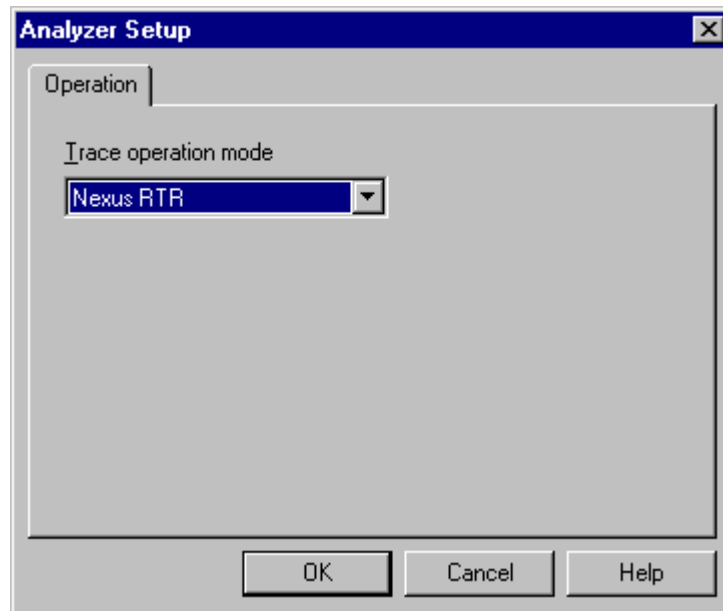
5.2.2 Nexus RTR Trace

The MPC56x development system offers advanced trace features, which are based on iSYSTEM Nexus RTR technology and restricted to the program execution bus:

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

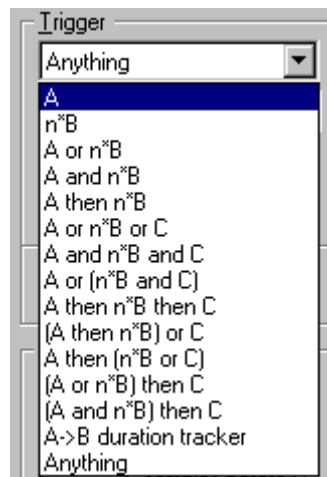
‘Nexus RTR’ must be selected in the Hardware/Analyzer Setup dialog to use these features.

Note: Nexus RTR is implemented on iTRACE GT development system only and for 8-bit MDO (CPU Nexus port) implementation only.



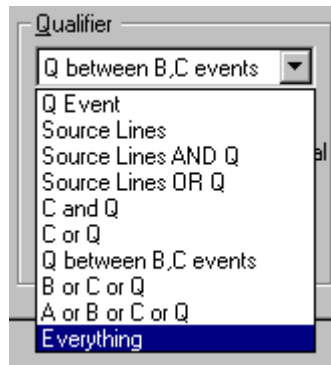
3-Level Trigger

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



Qualifier

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



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

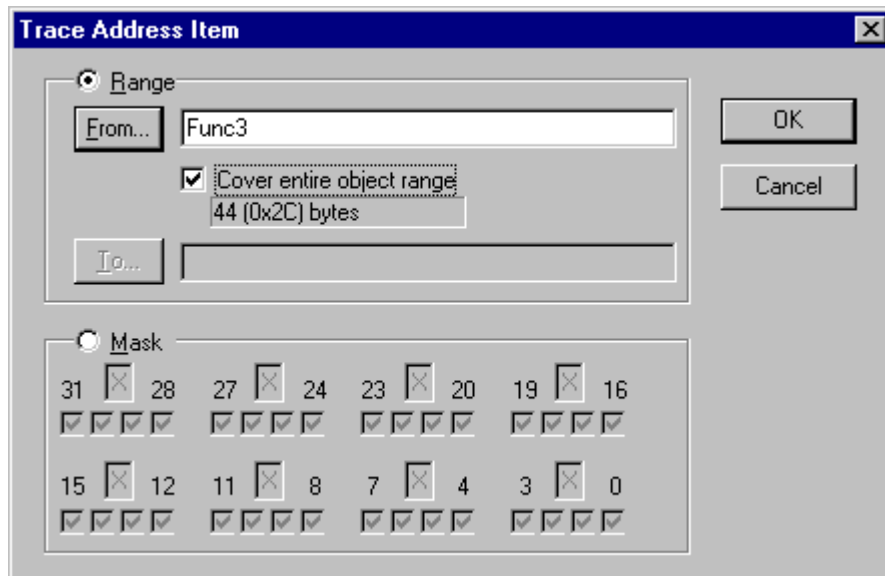


Thereby, the qualifier can be configured in a standard way and then additionally up to 8 CPU cycles can be recorded before and/or after the qualifier. For instance, this allows recording of a function or just its entry point and few instructions recorded before make possible to determine, which code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

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

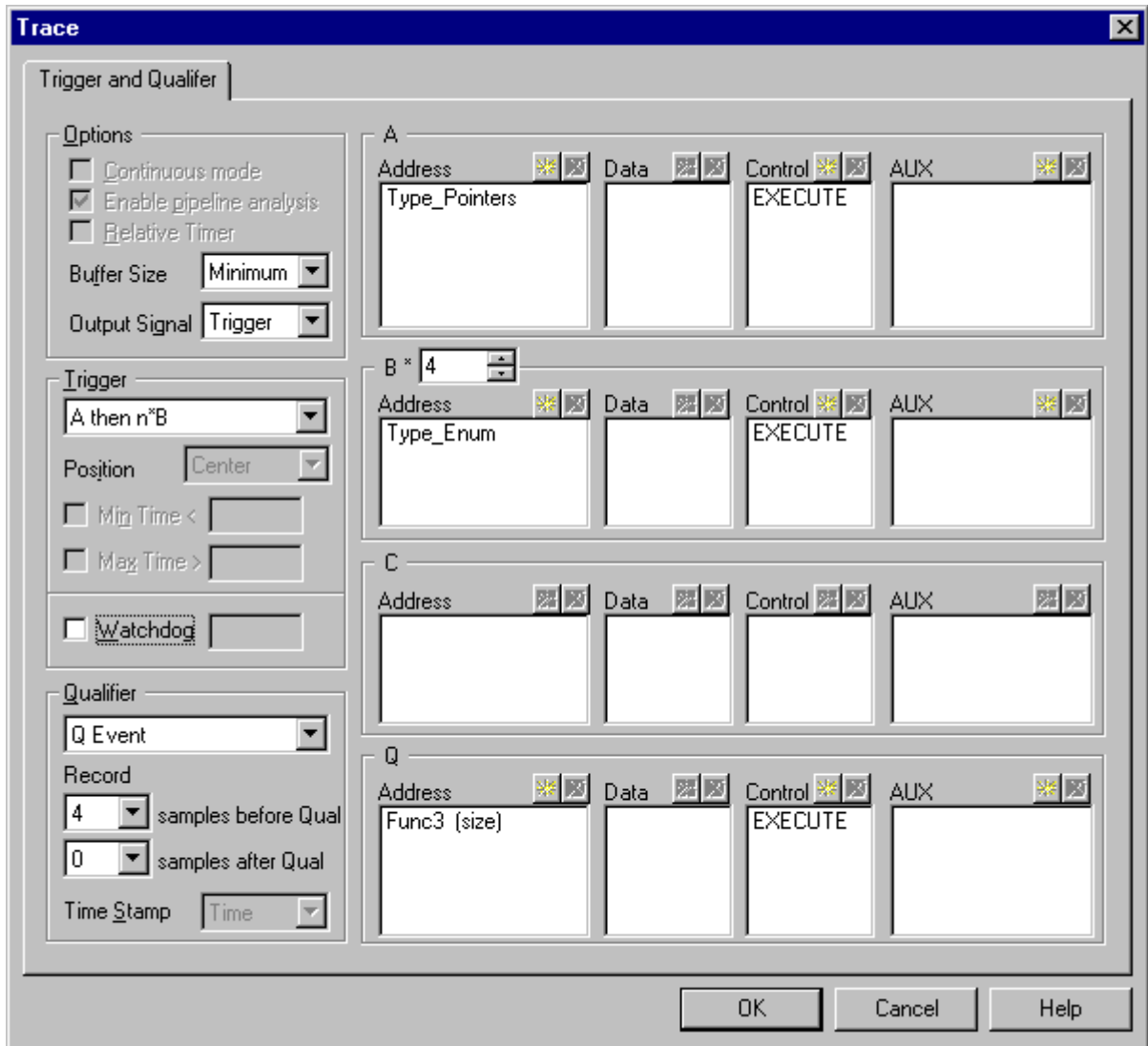
Example: Let's record Func3 execution after the Type_Pointers function is executed and then 4-times Type_Enum function is called.

- 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.

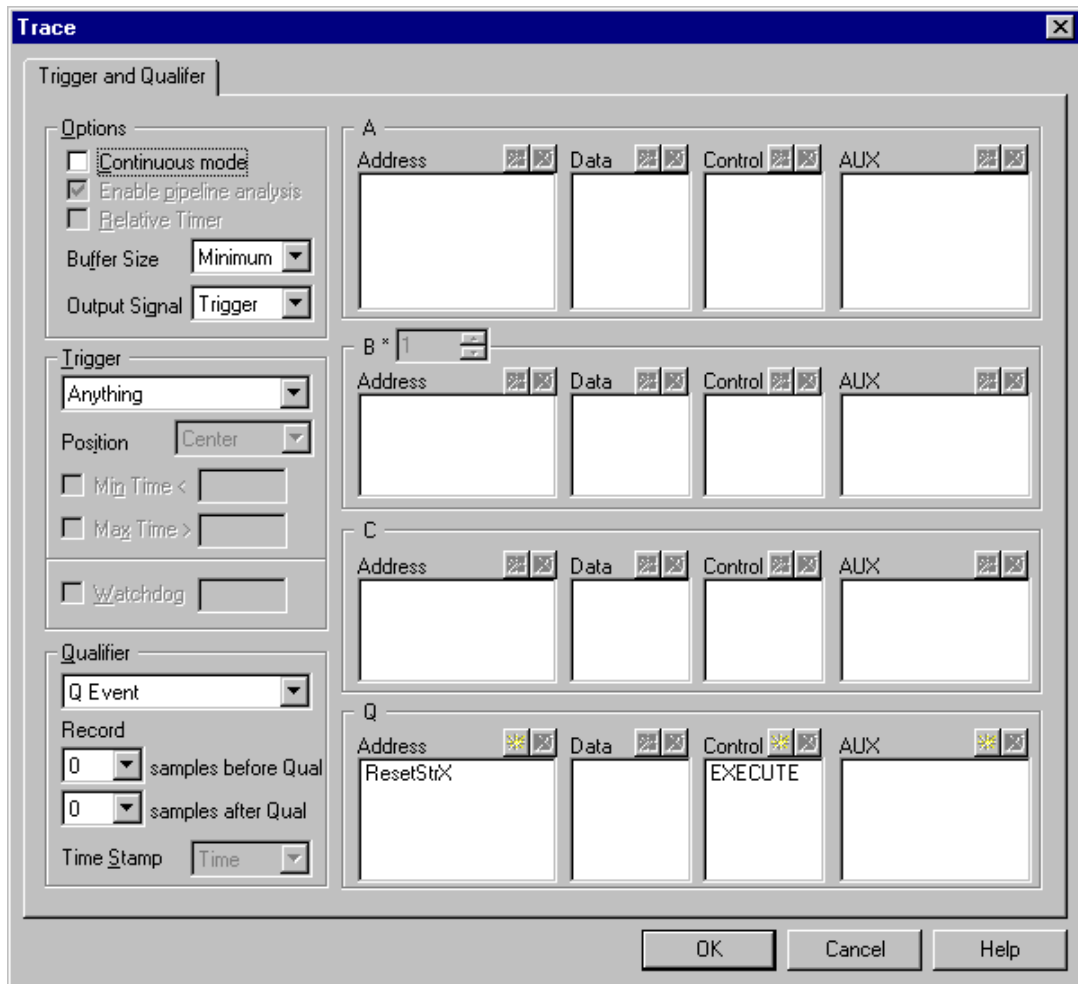
The user is able to determine which code actually called each Func3 function by clicking on any of four lines before Func3 entry point.

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 stw r1,-20(r1) Executed	18.910012 ms
50	00000D80	93E1001C	93E1001C stw r31,1C(r1) Executed	18.910025 ms
51	00000D84	7C3F0B78	7C3F0B78 mr r31,r1 Executed	18.910025 ms
52	00000D88	907F0008	907F0008 stw r3,08(r31) Executed	18.910037 ms
53	00000D8C	813F0008	*pY=0; 813F0008 lwz r9,08(r31) Executed	18.910037 ms
54	00000D90	38000000	38000000 li r0,00 Executed	18.910050 ms
55	00000D94	90090000	90090000 stw r0,00(r9) Executed	18.910050 ms
56	00000D98	81610000	} 81610000 lwz r11,00(r1) Executed	18.910062 ms
57	00000D9C	83EBFFFC	83EBFFFC lwz r31,-04(r11) Executed	18.910062 ms
58	00000DA0	7D615B78	7D615B78 mr r1,r11 Executed	18.910075 ms
59	00000DA4	4E800020	Func3_EXIT_ 4E800020 blr Executed	18.910075 ms

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

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Select 'Anything' for the trigger condition, specify Q Event for the Qualifier and then define the Q event.

Following picture depicts current trace settings.



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

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

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

Watchdog Trigger

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

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

Usage

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

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

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

Configuring Watchdog Trigger

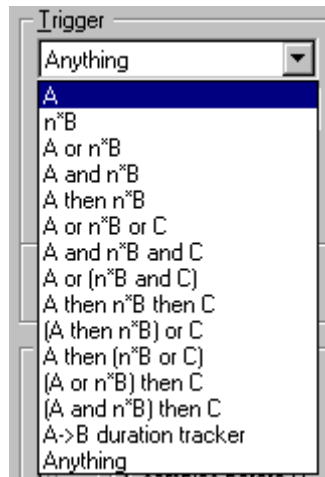
The user needs to enter the trace watchdog time-out period and define the “trace watchdog reset” condition, which can be logically combined from events A, B and C.

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



Trigger field

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



Trigger conditions

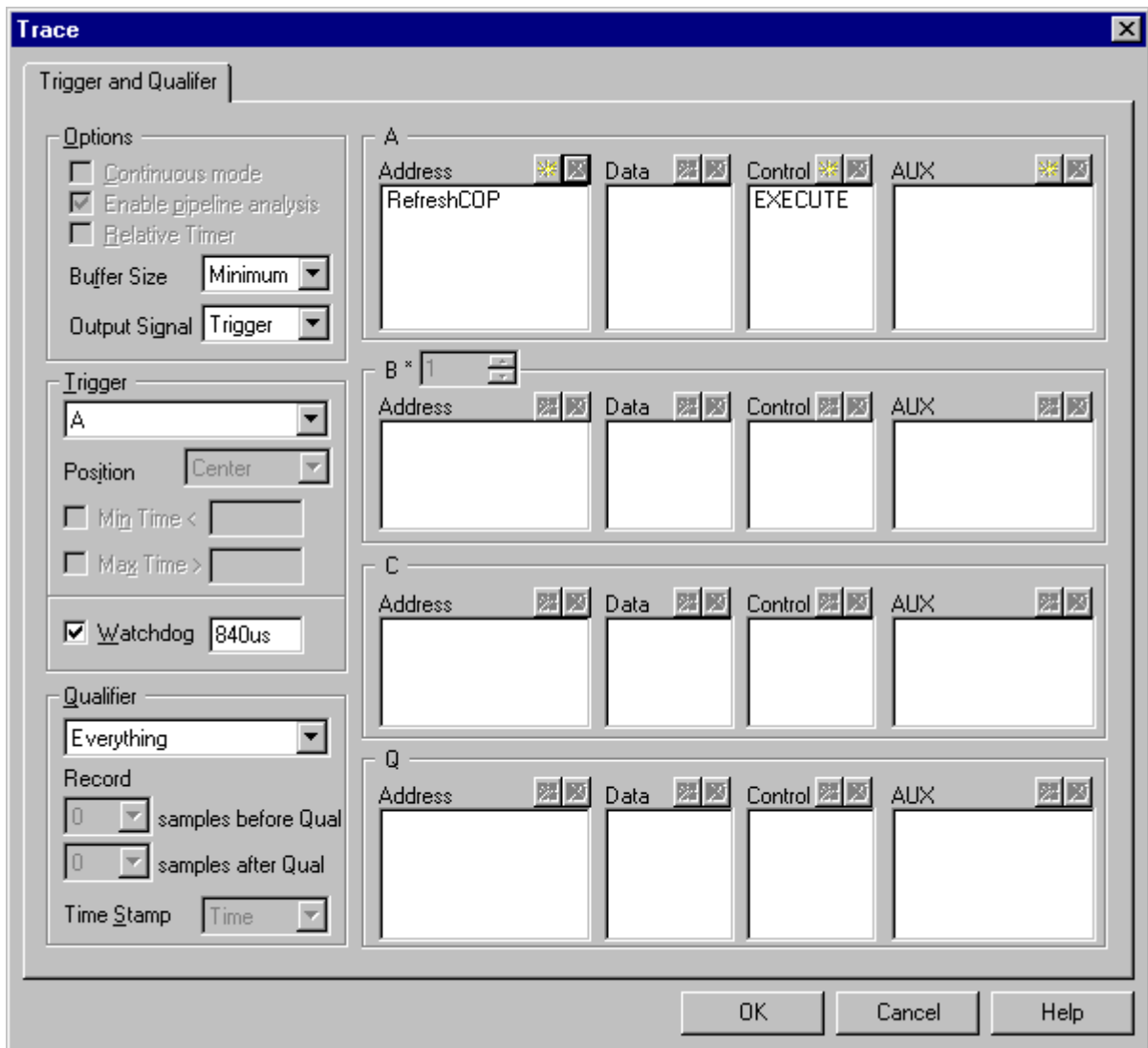
Example: Target application features on-chip COP watchdog, which enables the user to check that a program is running and sequencing properly. When the COP is being used, software is responsible for keeping a free running watchdog timer from timing out. If the watchdog timer times out it's an indication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

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

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

- 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.

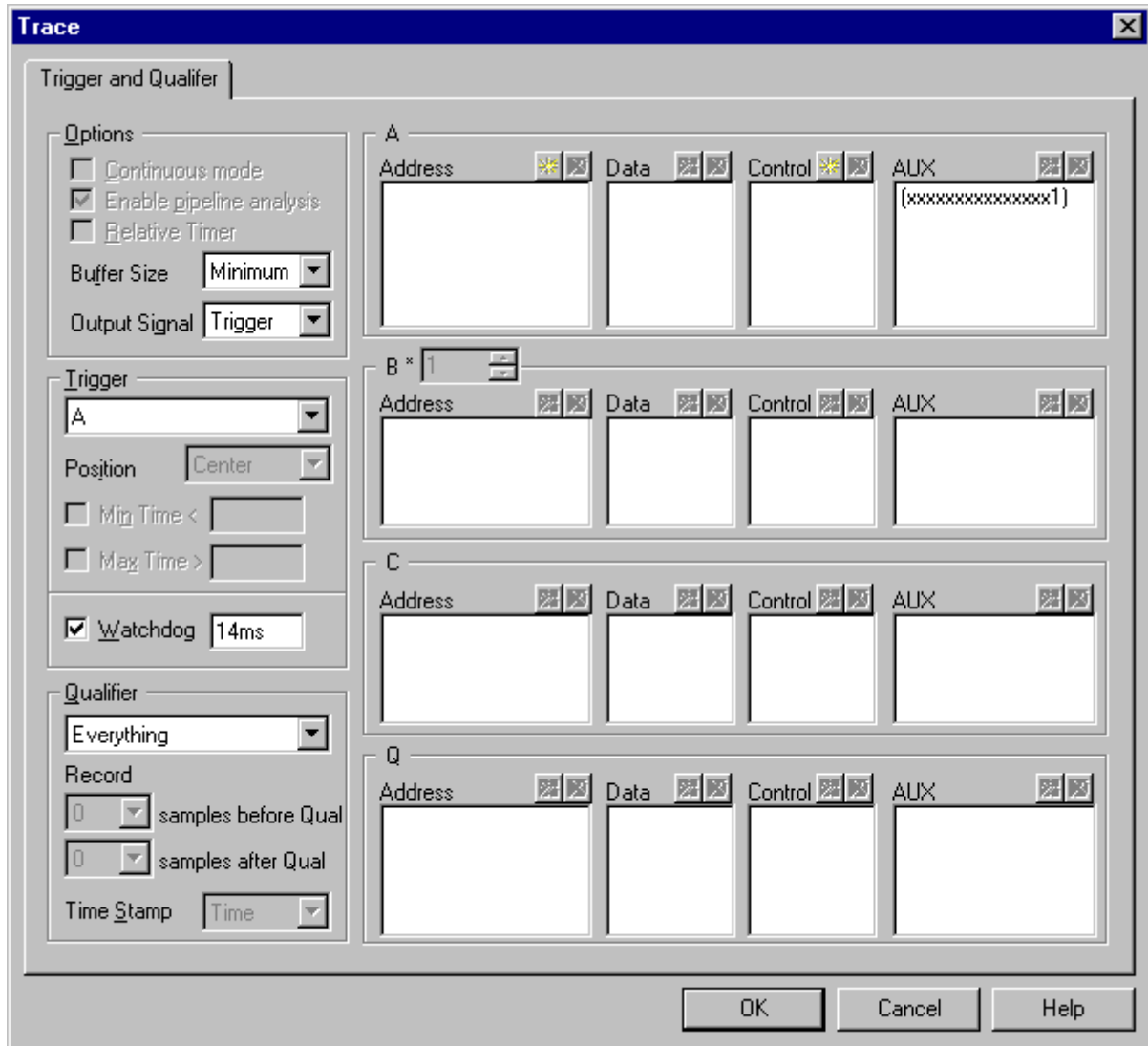
The trace needs to be configured to trap the target watchdog timer time out before it initiates a system reset. Then the user can find the code where the program misbehaves using the trace history.

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.



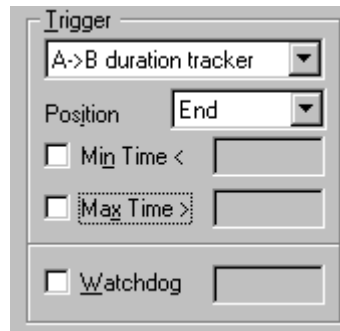
While the application operates correctly, the trace never triggers. The trace triggers when the application misbehaves, that is when RefreshCOP is no longer called within 840 μ s, and record the program behavior before that. The user can find out why the watchdog wasn't serviced by analyzing the trace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

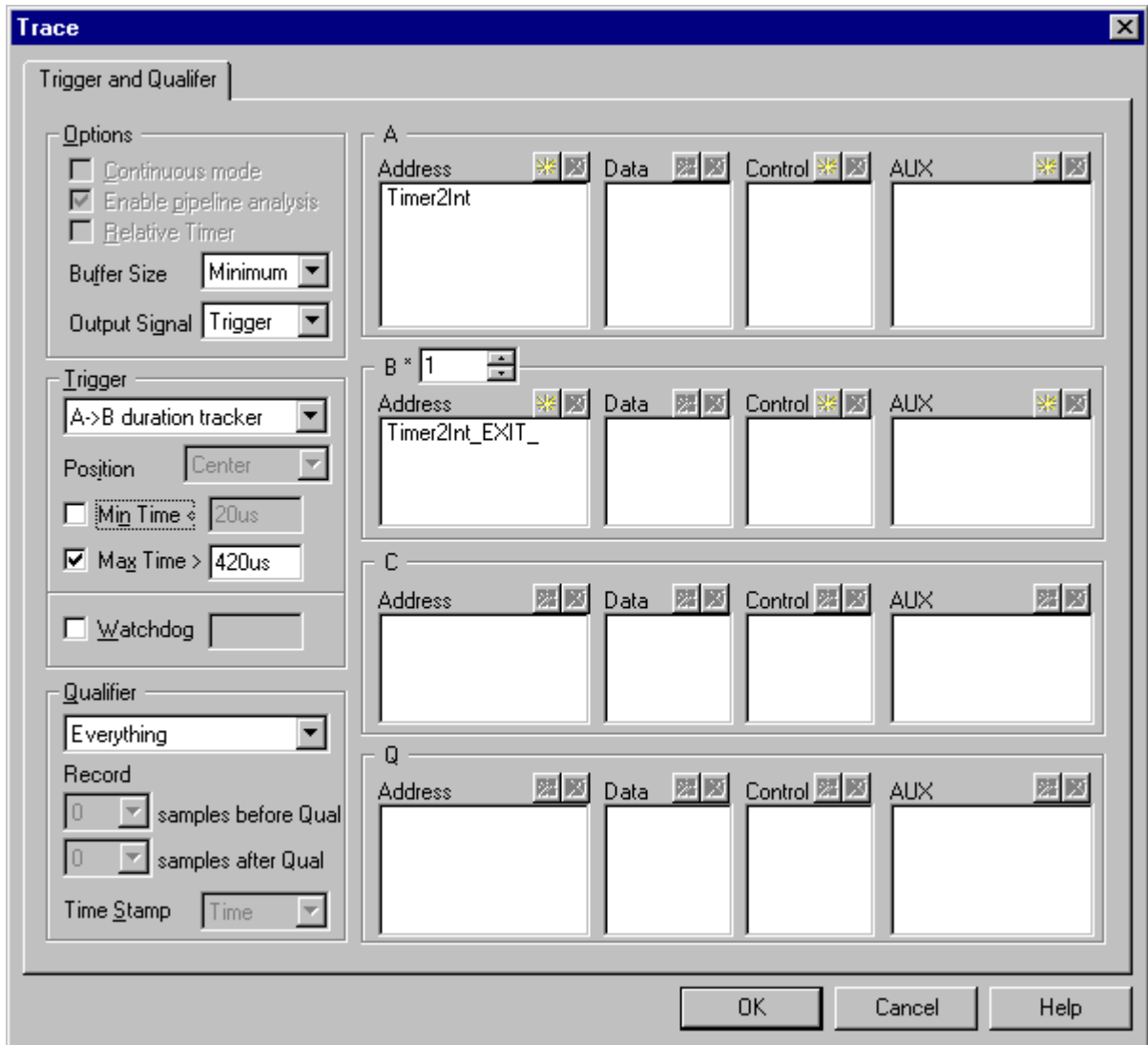
Based on the trace history, the user can easily find why the program executed out of the normal limits. Trace results can be additionally filtered out by using the qualifier.

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

- 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_` exit point for the event B. Make sure you select 'Fetch' access type for the control bus for both events since the object of our interest is the code.
- Check the 'Max Timer >' option and enter 420 μ s for the limit.

Below picture depicts current trace settings.



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



Duration Tracker Status Bar toolbar

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

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

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

Trace Window results

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

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

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

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

Duration Tracker Status Bar

Duration Tracker Status Bar reports:

Timer2Int minimum execution time was 54.95 μ s

Timer2Int average execution time was 235.90 μ s

Timer2Int maximum and current execution time was 416.85 μ s

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

Timer2Int routine completed 27 times.

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

The duration tracker ran for 27.88 ms.

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

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

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

6 Profiler

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

- **Functions Profiler**

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

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

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

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

- **Data Profiler**

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

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

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

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

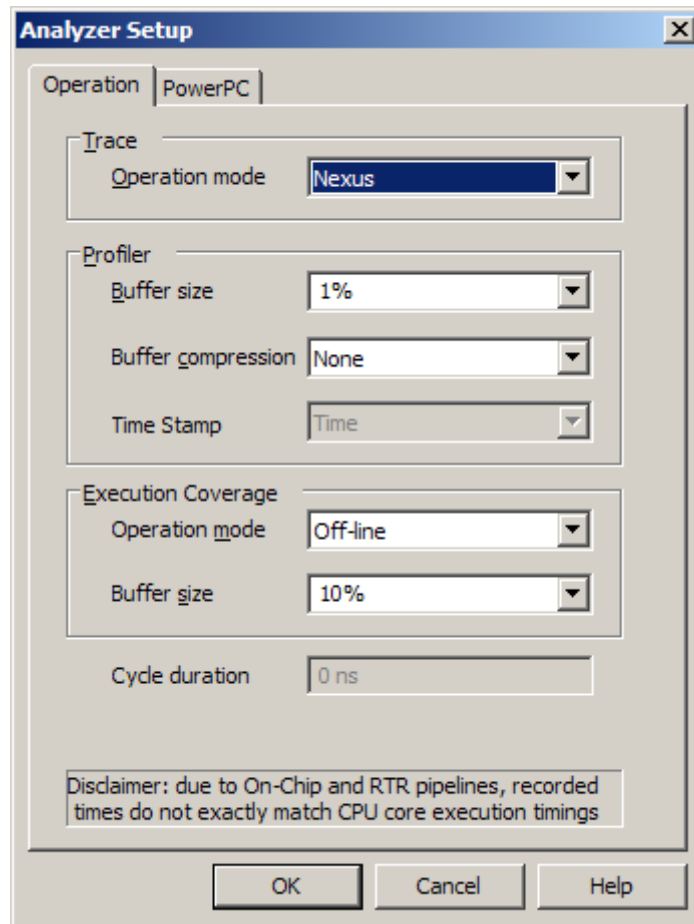
Nexus RTR is implemented on iTRACE GT for 8-bit MDO Nexus port only.

Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace. Single functionality can be used at the time only.

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

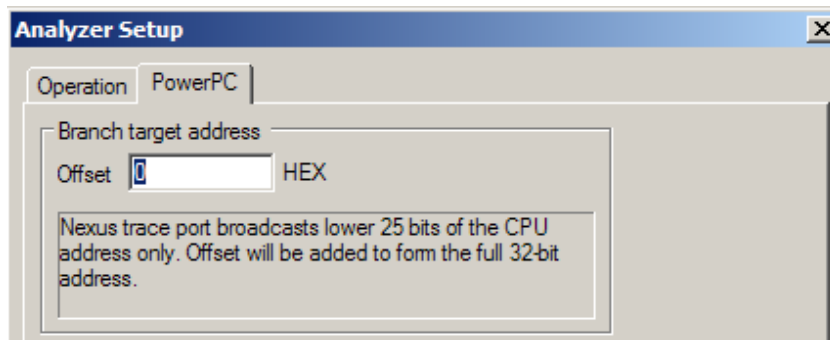
Typical Use

To use off-line profiler, select 'Nexus' trace operation mode and working profiler buffer size in the 'Hardware/Analyzer Setup' dialog. Keep 'Buffer Compression' set to None as this will provide the most accurate results.



For real-time profiler use, select 'Nexus RTR' trace operation mode and working profiler buffer size in the 'Hardware/Analyzer Setup' dialog.

Nexus doesn't report full 32-bit instruction address but only lower 25-bits. It is necessary to specify the address offset in the 'Hardware/Analyzer Setup/PowerPC' tab when the code runs at offset address exceeding the 25-bit address space. For instance, if the address offset is not set properly (e.g. set to 0), the profiler, trace and execution coverage will work correctly while running at one address (e.g. 0x800000) and incorrectly when the code is running at e.g. 0xFFFF0000.



Next, select 'Profiler' window from the View menu and configure profiler settings (see next figure). Select 'Functions' option in the 'Profile' field when profiling functions. In order to profile a data variable, 'Data'

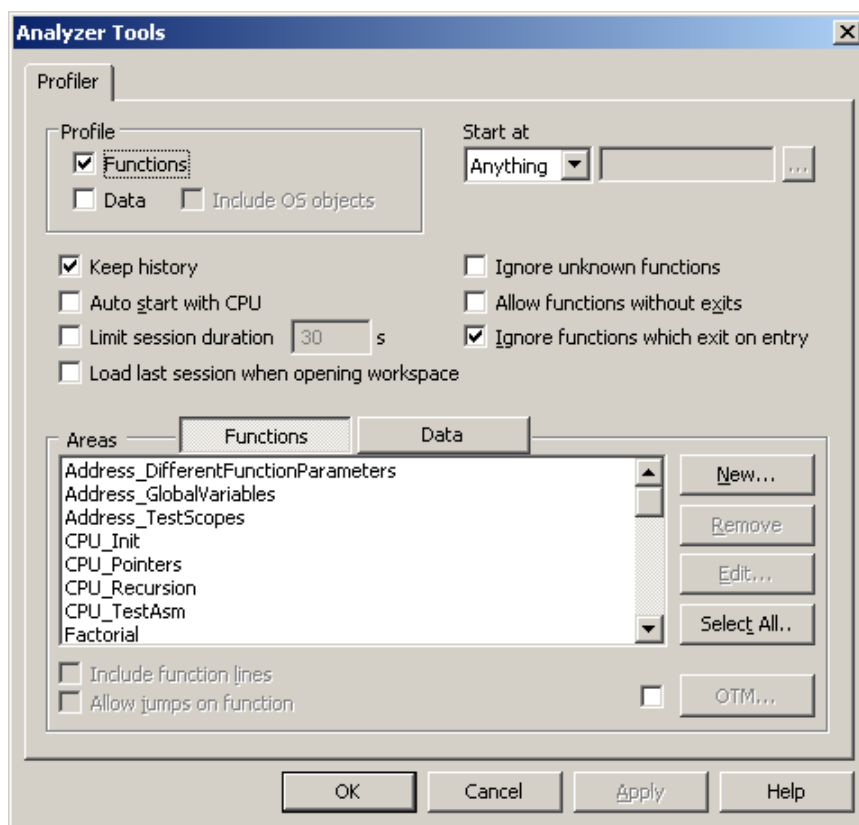
should be checked instead. For instance, Data Profiler can be used as a Task Profiler, when the operating system writes a unique task ID to a particular global variable at every task switch. The Profiler is then configured to profile that particular variable.

When using functions profiler in an application with operating system, the task switch variable ABSOLUTELY & UNCONDITIONALLY MUST be profiled too! Data profiler must be used to profile task switches.

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

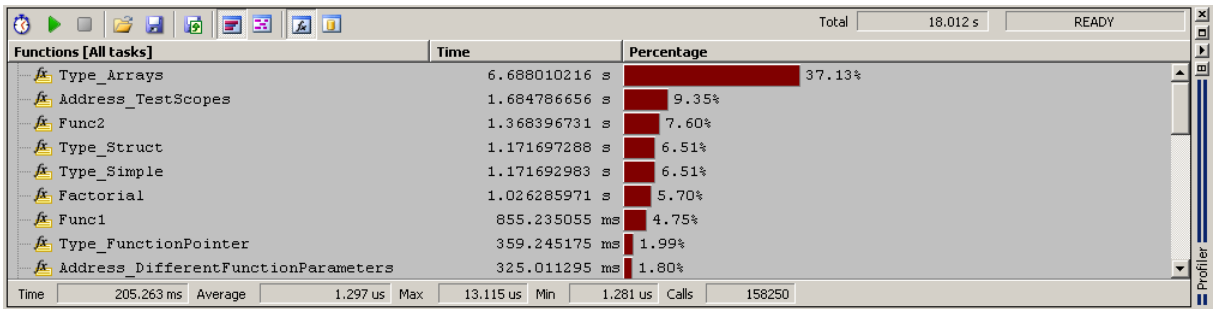
Finally, profiled functions are selected by pressing 'New...' button. It's recommended that 'All Functions' option is selected for the beginning.

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

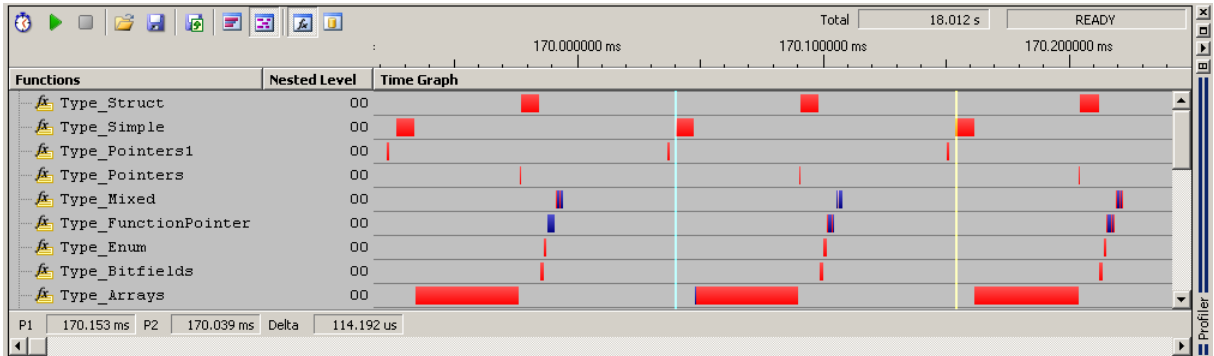


Profiler configuration settings

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



History view



Statistics view

7 Execution Coverage

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

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

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

Real-time execution coverage is based on a hardware logic, which in real-time registers all executed program addresses. It features statement coverage but no decision coverage since it keeps the information on all executed program addresses but without any history, which would tell which address was executed when and in what order.

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

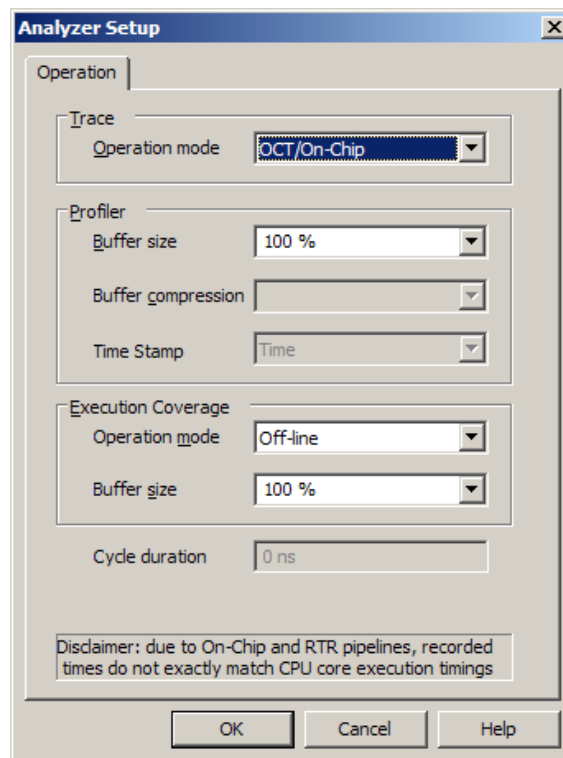
Refer to a separate Execution Coverage User's Guide for more details on execution coverage configuration and use.

Nexus RTR is implemented on iTRACE GT for 8-bit MDO Nexus port only.

Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace. Single functionality can be used at the time only

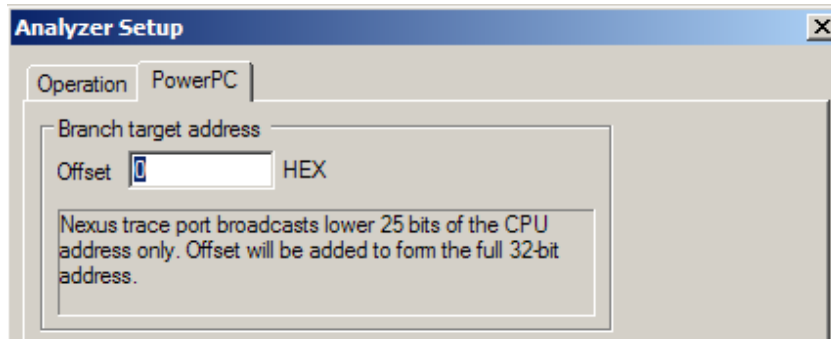
Typical Use

To use off-line execution coverage, select 'Nexus' trace operation mode and working execution coverage buffer size in the 'Hardware/Analyzer Setup' dialog.

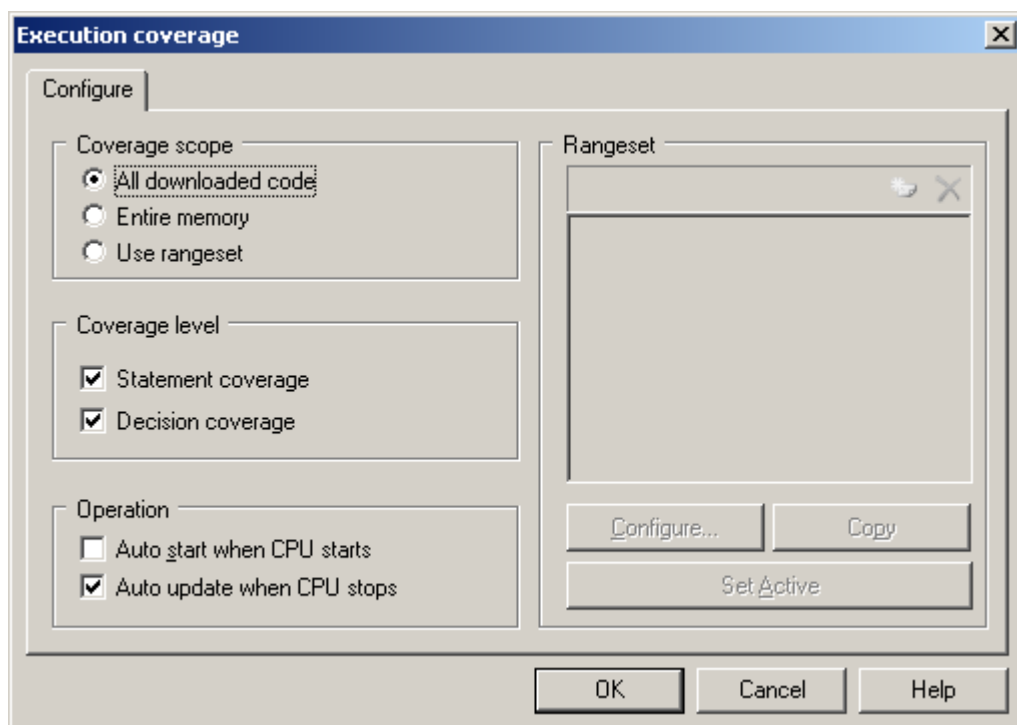


For real-time execution coverage use, select 'Nexus RTR' trace operation mode. Buffer size setting is not applicable for this mode.

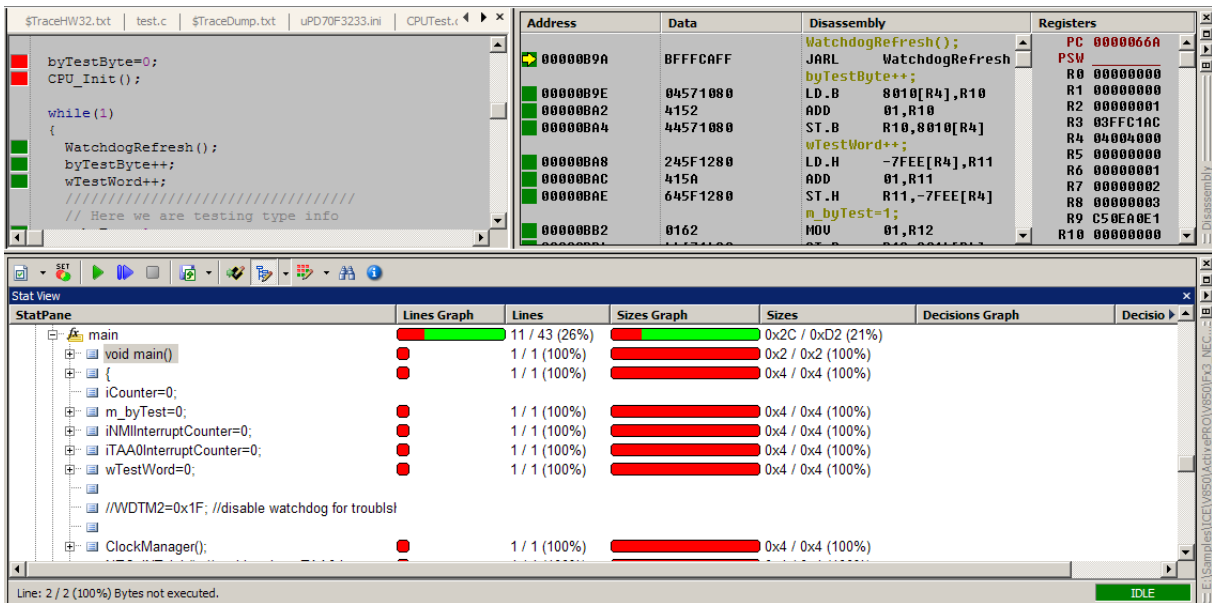
Nexus doesn't report full 32-bit instruction address but only lower 25-bits. It is necessary to specify the address offset in the 'Hardware/Analyzer Setup/PowerPC' tab when the code runs at offset address exceeding the 25-bit address space. For instance, if the address offset is not set properly (e.g. set to 0), the profiler, trace and execution coverage will work correctly while running at one address (e.g. 0x800000) and incorrectly when the code is running at e.g. 0xFFFF0000.



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



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



Execution Coverage results

8 Getting Started

Before powering on check the Power supply setting in Emulation Options/Hardware. Default is 3.3V, note however that certain newer devices operate at lower voltages and may no longer be 3.3V-tolerant. Use 'Target' setting in such case. Target board should provide its I/O reference voltage on the debug connector.

Disable watchdog with the option described above. Usually the watchdog timer will timeout after just a few seconds after reset is released, leaving very little time for user intervention. At the same time, don't forget to disable or adequately service the watchdog timer in your software.

To enable code download user should provide some minimum memory access configuration. This can be trivial by merely setting the appropriate chip-select registers for flash or SRAM, or complicated and lengthy for DRAM. Note that there's no need for that with the 5xx/8xx that have internal RAM sufficient for small test code.

See notes above for setting debug clock and reset delay.

In case the debugger won't start, disable initialization and all configurable options except disabling the watchdog. Check if the CPU stops at reset vector 0xFFFF00100, or 0x100 if MSR_{IP} is 0.

It is recommended that CPU clock PLL is not modified in the initialization sequence as this may cause CPU debug port to fail. Instead, user code should do this in any case.

Under Emulation Options/CPU Setup check all exceptions that don't have a handler in your code. This is a good means to detect early when your code strays away.

9 Troubleshooting

- Q: We noticed that our MPC565 application is running faster when the debugger is connected than when it's running in stand-alone. We are using iTRACE PRO/GT Nexus development system. What could be the reason for this behavior?

A: Most probably your application doesn't set the ISCT_SER field of the ICTRL (debug) special purpose register. After reset, the microcontroller runs fully serialized and show cycles are performed for all fetched instructions (ISCT_SER = b000). When the debugger is connected, following ISCT_SER values are set:

- 5 to facilitate 565 Nexus trace operation, before chip revision D
- 6 to facilitate 565 Nexus trace operation, with and after revision D

- 7 for all MPC5xx/8xx BDM and Nexus without trace

In your case, the application (e.g. startup code) should set ISCT_SER to 7. Refer to the Development Support chapter in the 565 Reference Manual for more details on the ISCT_SER field.

- When attaching an emulator to the MPC5xx/8xx, some specific considerations are called for:
 1. Actively drive the DBGC (D9,D10) and DBPC (D11,D12), Hard Reset Configuration Word bits onto the bus to select the Multifunction I/O pin selection for target and Development Port. The values required to be driven onto these pins are based on the signals that have been routed to the 10-pin BDM connector. This doesn't apply to MPC505/509.
 2. There are two BDM 'Freeze' signal schemes. Option A is recommended for Trace option. But, Option B may also be used as an alternative when target design uses PCMCIA Port B. Option B requires that the bit SIUMCR.FRC=0. For BDM operation alone it is irrelevant what option is used. With the exception of MPC505/509 the emulator uses BDM software command to establish CPU's run status.
 3. Physically locate the 10-pin connector as close as possible to the processor to minimize trace length and crosstalk onto the BDM signals. Do not run high-speed clocks or signals adjacent to the BDM communication signals.
 4. Ensure that the emulator /HRESET connector pin is connected directly to the /HRESET signal of the processor. This will provide the ability for the emulator to drive and sense the status of /HRESET. The target design should only drive the /HRESET with open collector or open drain type devices.
 5. /HRESET should not be tied to /PORESET. The emulator drives the /HRESET and DSCK to enable BDM operation.
 6. To guarantee that the JTAG Mode is not accidentally invoked, connect a pull down (~10K) resistor on the /TRST signal and a pull-up (~10K) on the TMS signal. When the MPC's development port (BDM) is used, JTAG functionality is disabled. Designs that require both, should have a Reset configuration scheme to support the two modes. We use 10k pulldown on a '245 buffer that drives DBPC=00 and enables BDM functionality. And if a JTAG tester is connected it overcomes the pulldown to a logic high and the DBPC=11 enabling JTAG pins.

Known Issues

- Problem: Download fails with verify errors.

Solution: For example, program segments A, B and C are linked together in a download file. But the B segment is mapped to a non-existent memory region. This in turn, causes failure in writing the first 8 bytes of the segment C. Of course, the solution is to always check the Load Map and make sure that the program is linked to valid memory regions.

- Problem: I'm trying to inspect the floating-point registers. How do I do that?

Solution: Open the SFR window and expand the Floating Point Register group. If the MSR.FP floating point enable bit is zero, all floating-point registers will display zero.

- Problem: Nexus trace doesn't display properly 16-bit data write access in terms of data size.

Explanation: Nexus doesn't report the size of the data transfer. Instead it is guessed according to the number of bits transferred in the data field but this may not work always. As an example when the algorithm fails, 16-bit data access is displayed as 8-bit access.

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.