

Trace – Getting Started

V8.02



1. Introduction

This paper helps the user to entirely exploit the trace and troubleshoot most often situations that the developer is confronted with while debugging the application.

Trace is just one of the functionalities supported by the analyzer module. Beside the trace, the analyzer features data and execution profiler, execution coverage and access coverage. Refer to other documents for more details on these functionalities.

This document assumes that the user is familiar with winIDEA. If this is not the case, refer to other documents, like Software User's Guide and Hardware User's Guide delivered with the development tool, to get familiar with the IDE and the development system.

2. Facts And Restrictions

Described trace features and functionalities apply to iSYSTEM HC(S)12 ActivePODs supporting Motorola 68HC12 and HCS12 families.

Trace depth: Minimum buffer 32K frames, maximum buffer 128K frames

Time Stamp: 20ns

Time Reach: Unlimited

There are restrictions arising from the CPU itself when recording accesses to the internal CPU memory resources (RAM, EEPROM, registers).

No internal RD and WR cycles are visible when the CPU operates in 8-bit mode. Thereby, no trigger or qualifier can be set on these cycles.

All internal RD and WR cycles are visible but data bus is not active during internal RD cycle when the CPU operates in 16-bit mode. Respect this restriction when setting trigger or qualifier.

Above restrictions apply to 68HC12 and HCS12 CPUs. Other CPUs might impose different restrictions.

3. Trace Toolbar

Basic toolbar allows basic navigation through the trace record and setting up all the hardware-based settings like setting trace buffer size, time stamp resolution and trigger position, selecting trace mode and configuring trigger and qualifier events.

Trace record can be exported in a text format, various filters set and custom logical and group signals created using the advanced toolbar.



Figure 1. Basic toolbar

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



Figure 2. Advanced toolbar

Trace Toolbar:

1. Hardware Configuration
2. Trigger Configuration
3. Begin
4. Stop
5. Function Tree
6. Options
7. Duration Tracker Status Bar
8. Timing view
9. State View
10. Swap Views (active only when both, Timing and State View are selected)
11. Configure Signals
12. Insert Logical Signal
13. Insert Group Signal
14. Configure States and Filters
15. Collapse Toolbar
16. Export

4. Trace Window – Display Options

After the trace recording completes, the trace buffer is uploaded and analyzed, and the results displayed in the trace window based on the display options.

Display options are set either by using the Function Tree toolbar, where pull-down menu is available or Options toolbar, which opens the 'Options' dialog.



Figure 3. Function Tree toolbar pull-down menu

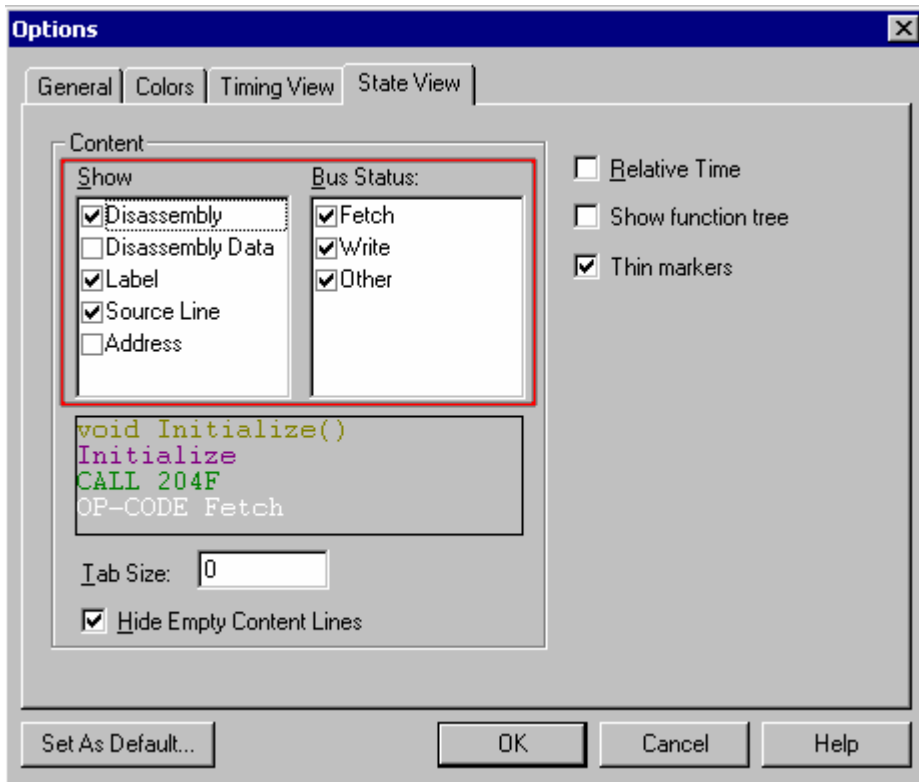


Figure 4. Options dialog

Low level info on CPU bus activity is displayed when 'Fetch', 'Write' and 'Other' options only are checked in the 'Bus Status' field. Only access type separates recorded CPU cycles.

	9	F008	8E31	OP-CODE Fetch	300 ns	0000
P	10	F00A	3A20	Code Read	333 ns	0000
	11	F00C	F925	OP-CODE Fetch	366 ns	0000
	12	F00E	3FCF	Code Read	416 ns	0000
	13	F010	16FF	Code Read	450 ns	0000
	14	F006	6C02	OP-CODE Fetch	500 ns	0000
	15	F008	8E31	OP-CODE Fetch	533 ns	0000
	16	F00A	3A20	Code Read	566 ns	0000
	17	F00C	F925	OP-CODE Fetch	616 ns	0000
	18	2008	0000	Memory Write	650 ns	0000
	19	F00E	3FCF	Code Read	700 ns	0000

Disassembly
Disassembly Data
Label
Source Line
Address

Fetch
 Write
 Other

Figure 5. CPU accesses displayed

Check the 'Disassembly' option to display the disassembly in the trace window.

	9	F008	8E31	CPX #203A OP-CODE Fetch	300 ns	0000
P	10	F00A	3A20	Code Read	333 ns	0000
	11	F00C	F925	BCS F007 OP-CODE Fetch	366 ns	0000
	12	F00E	3FCF	Code Read	416 ns	0000
	13	F010	16FF	Code Read	450 ns	0000
	14	F006	6C02	STD 2,X+ OP-CODE Fetch	500 ns	0000
	15	F008	8E31	CPX #203A OP-CODE Fetch	533 ns	0000
	16	F00A	3A20	Code Read	566 ns	0000
	17	F00C	F925	BCS F007 OP-CODE Fetch	616 ns	0000
	18	2008	0000	Memory Write	650 ns	0000
	19	F00E	3FCF	Code Read	700 ns	0000

Disassembly
Disassembly Data
Label
Source Line
Address

Fetch
 Write
 Other

Figure 6. CPU accesses and disassembly displayed

In Figure 6, it's not clearly visible where the disassembled code starts and which bytes from the data bus belong to each disassembled instruction. For example, it's not clear whether CPX #203A instruction (frame 9) is located at 0xF008 or 0xF009 and how many bytes does it occupy. Check the 'Disassembly Data' and 'Address' options to get more detailed disassembly. In Figure 7, we can see that CPX #203A starts at 0xF009 and occupies 3 bytes of code.

	9	F008	8E31	F009 8E203A CPX OP-CODE Fetch	300 ns	0000
P	10	F00A	3A20	Code Read	333 ns	0000
	11	F00C	F925	F00C 25F9 BCS FO OP-CODE Fetch	366 ns	0000
	12	F00E	3FCF	Code Read	416 ns	0000
	13	F010	16FF	Code Read	450 ns	0000
	14	F006	6C02	F007 6C31 STD 2, OP-CODE Fetch	500 ns	0000
	15	F008	8E31	F009 8E203A CPX OP-CODE Fetch	533 ns	0000
	16	F00A	3A20	Code Read	566 ns	0000
	17	F00C	F925	F00C 25F9 BCS FO OP-CODE Fetch	616 ns	0000
	18	2008	0000	Memory Write	650 ns	0000
	19	F00E	3FCF	Code Read	700 ns	0000

Figure 7. CPU accesses and disassembly with address and data displayed

Check the 'Source Line' and 'Label' options to display source lines and labels in the trace.

	225	F088	872E	iDataChangeVar=0; CLRA OP-CODE Fetch	8.933 us	0000
	226	F08A	7CC7	CLRB STD iDataChangeV OP-CODE Fetch	8.966 us	0000
	227	F08C	0C20	Code Read	9.016 us	0000
	228	F08E	7C42	iDownCounter=0x100; INCA STD iDownCounter OP-CODE Fetch	9.050 us	0000
	229	F08E	7C42	Code Read	9.100 us	0000
	230	F090	0A20	Code Read	9.133 us	0000
	231	200C	0000	iDataChangeVar iDataChangeVar+1 Memory Write	9.166 us	0000

Figure 8. Source lines, disassembly and labels displayed

5. Handling Trace Files

WinIDEA supports use of more trace files simultaneously. Figure 9 shows winIDEA window layout with the trace window opened. WinIDEA names the default trace file opened in the trace window based on the project name.

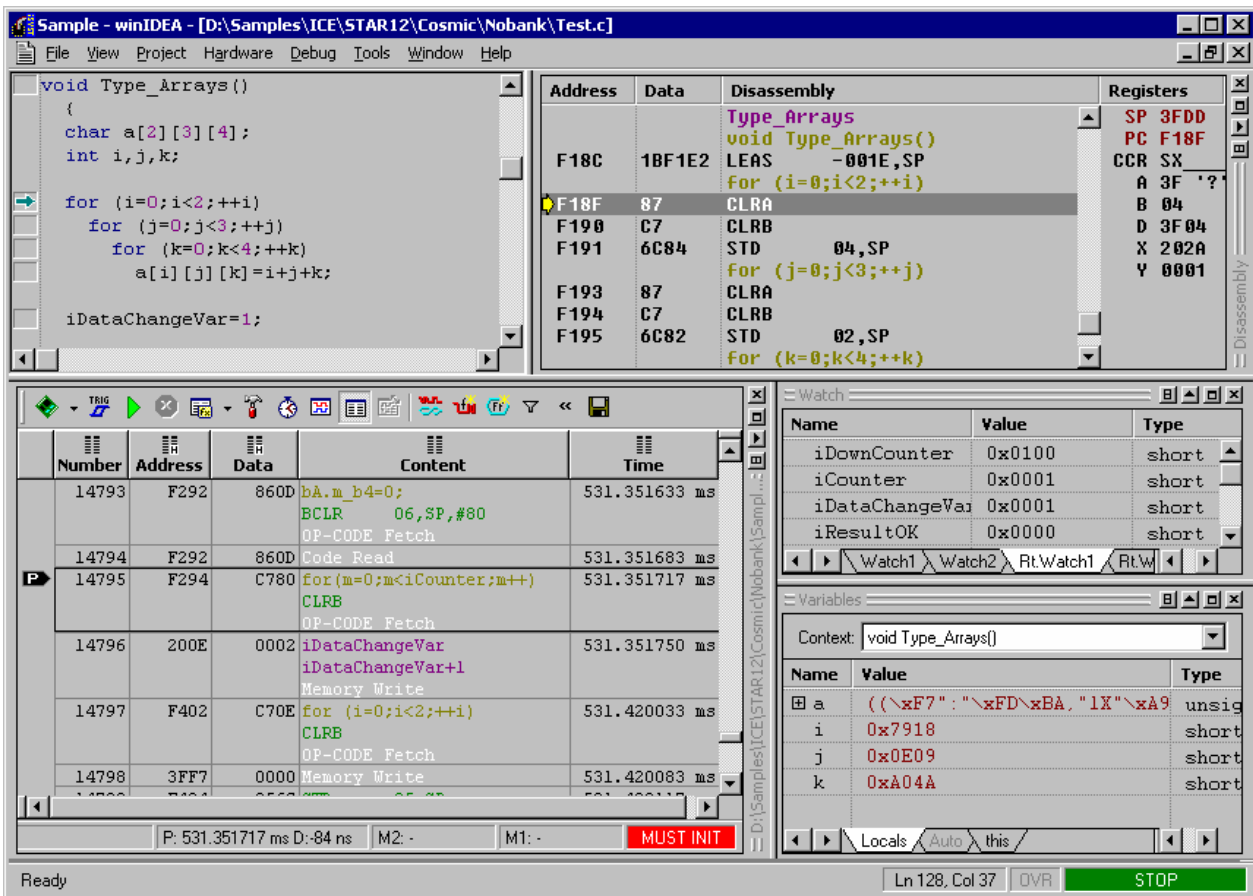


Figure 9. winIDEA Window Layout

Note that the editor window is 'MDI' type and the trace window 'Dock' type. Modify them if necessary using 'Window Type' option from the local menu.

To save the trace record as a document file:

- Make sure that the trace window is focused.
- Select 'Save As...' from the File menu and save the trace file. Trace file is saved with the .trd extension.

Assume that we have recorded three trace files: Example1.trd, Example2.trd and Example3.trd. Trace window can display only a single trace file. More trace files can be opened in the editor window. Open each file by selecting 'Open' from the File menu. Trace files are opened in the editor window, probably overlapping each other (MDI window type). The user may switch among them by using Ctrl-tab shortcut, however, this may not be very convenient.

Select 'Document Bar' from the View menu. Document Bar displays all the files being opened and allows quick navigation through the files (Figure 10).



Figure 10. Document Bar

Project Workspace window use is an alternative to the Document Bar. Close the Document Bar, open Project Workspace window by selecting 'Project' from the View menu and add a group called 'Trace files' using the local menu. Next, click on a newly created group, select 'Add files' from the local menu and add

all three trace files. Now, the user is able to navigate through the trace files by simply clicking the files (Figure 11).

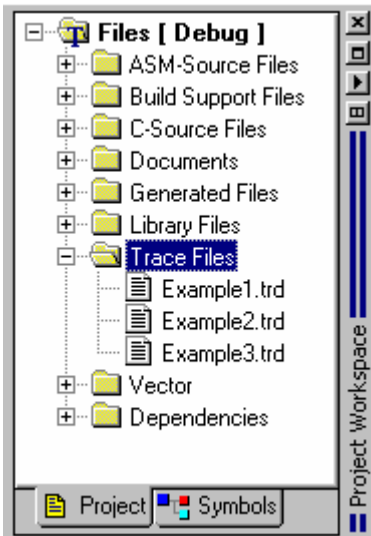


Figure 11. Project Workspace Window

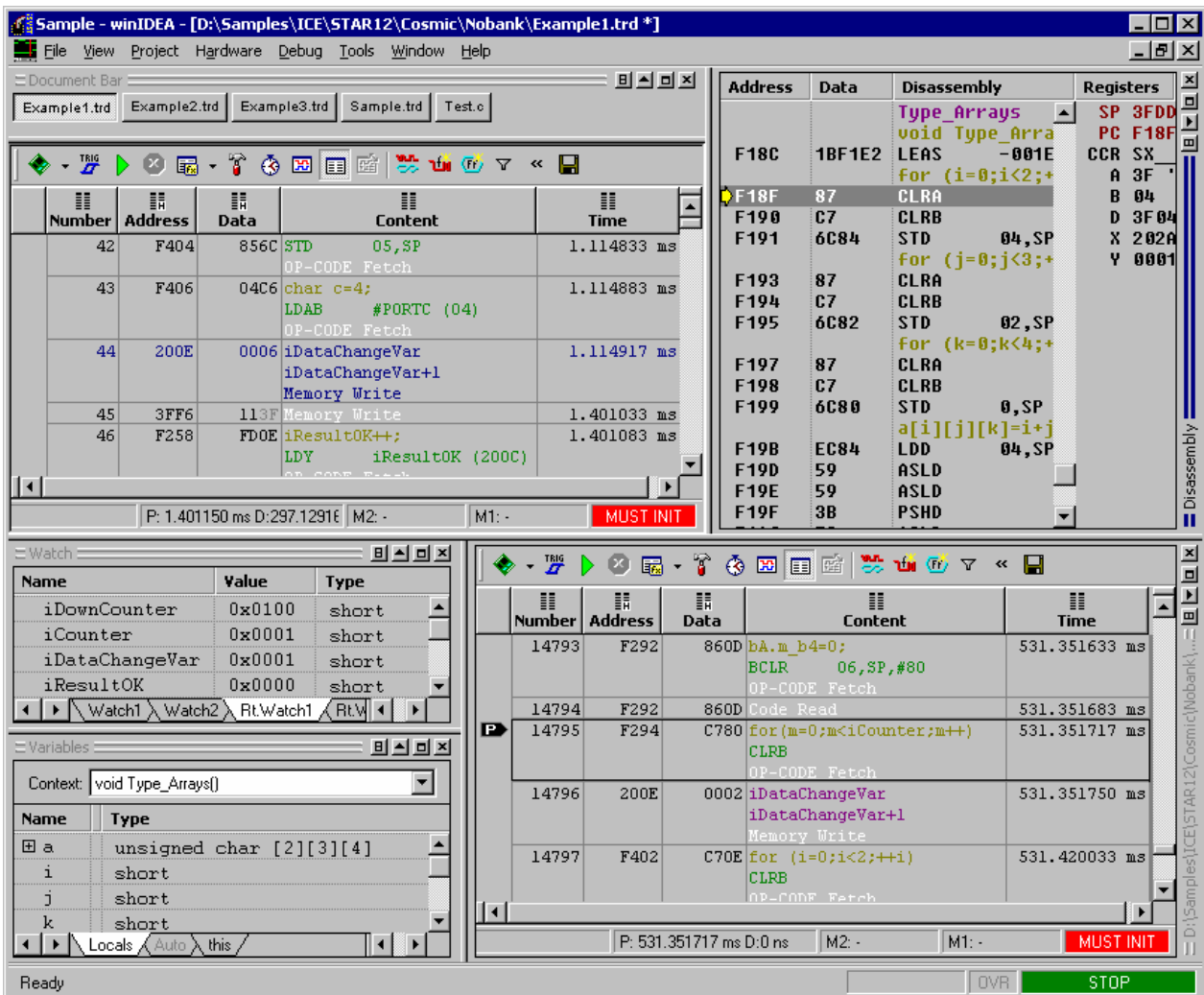


Figure 12. Two trace files

Troubleshooting Scenarios

5.1. 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' use 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 the breakpoint hit, due to the erratic state of the CPU, which yields the program stop, or due to the stop debug command issued by the user.

Example: The application enters unexpectedly in a halted mode (sleep mode, power down mode) during normal program execution. Such cases can be investigated only by using the trace since the CPU is no longer under the debugger's control while in halted mode. Using the trace, the course of program before the halted mode entry is recorded and the source of the unexpected entry can be found.

- 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 program execution 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.

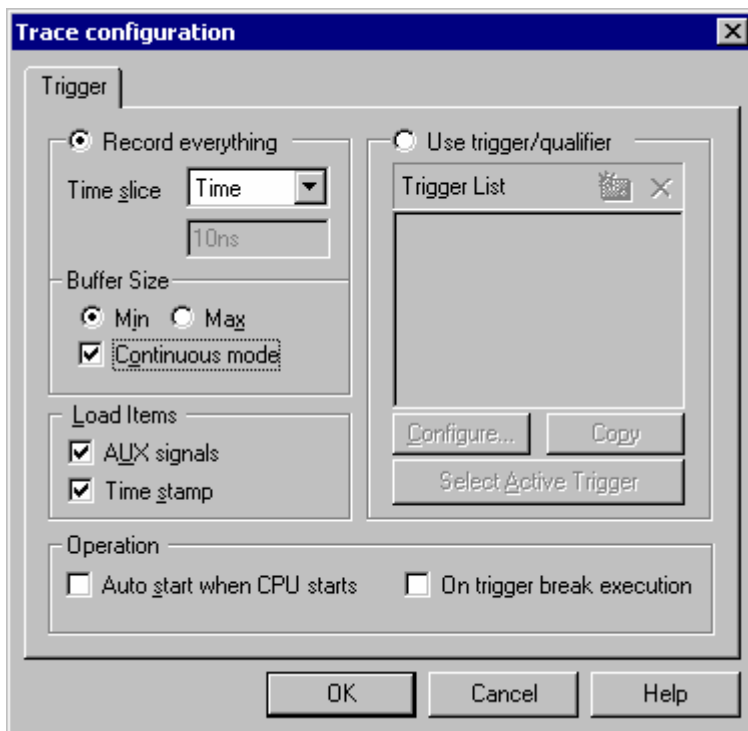


Figure 13. Trace Configuration dialog

With these settings (Figure 13), the trace records program flow while it's running. As soon as the program enters halted mode or is stopped due to any other reason, the trace stops recording. The source of the problem can be analyzed easily since the program flow before the program stop is recorded.

5.2. Plain Trigger/Qualifier Configuration

Configuring simple triggers set on a variable being accessed or an instruction being executed are described in this section. Additionally, simple qualifiers are configured.

If it's required to stop the program on a trigger event, check the 'On trigger break execution' option in the 'Trace Configuration' dialog.

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

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add and name a new trigger (e.g. `AddressTrigger`), and open 'Trigger and Qualifier Configuration' dialog.

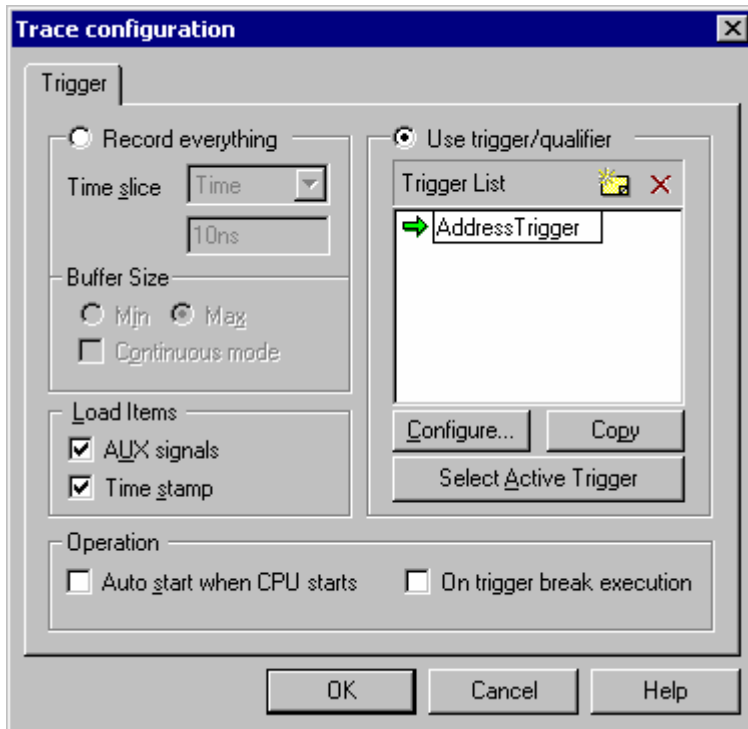


Figure 14. Trace Configuration dialog

- Select 'n*B' for the trigger condition. When the trigger on a first occurrence is required, 'A' can be opted instead of 'n*B' in the trigger condition field.
- Set `Type_Struct` entry point for the event B and set 5 in the counter field.
- When configuring an event on a code, select 'OP-CODE Fetch' access type for the Control Item to obtain correct results, which may otherwise be distorted due to the CPU pipeline queue. Select either 'Memory Write' or 'Data Read' access type when configuring an event on a data.

If the minimum buffer size is selected (default selection) and the trace buffer was filled completely, then a first half of the trace buffer contains program executed before the trigger and a second half contains program executed after the trigger.

When a maximum buffer is selected, the trigger position can be set at the beginning, center or at end of the trace buffer. Set the position based on your particular case since the code being of interest can be executed after or before the trigger event.

The trace is configured. Figure 15 depicts current trace settings. Initialize the complete system, start the trace and run the program.

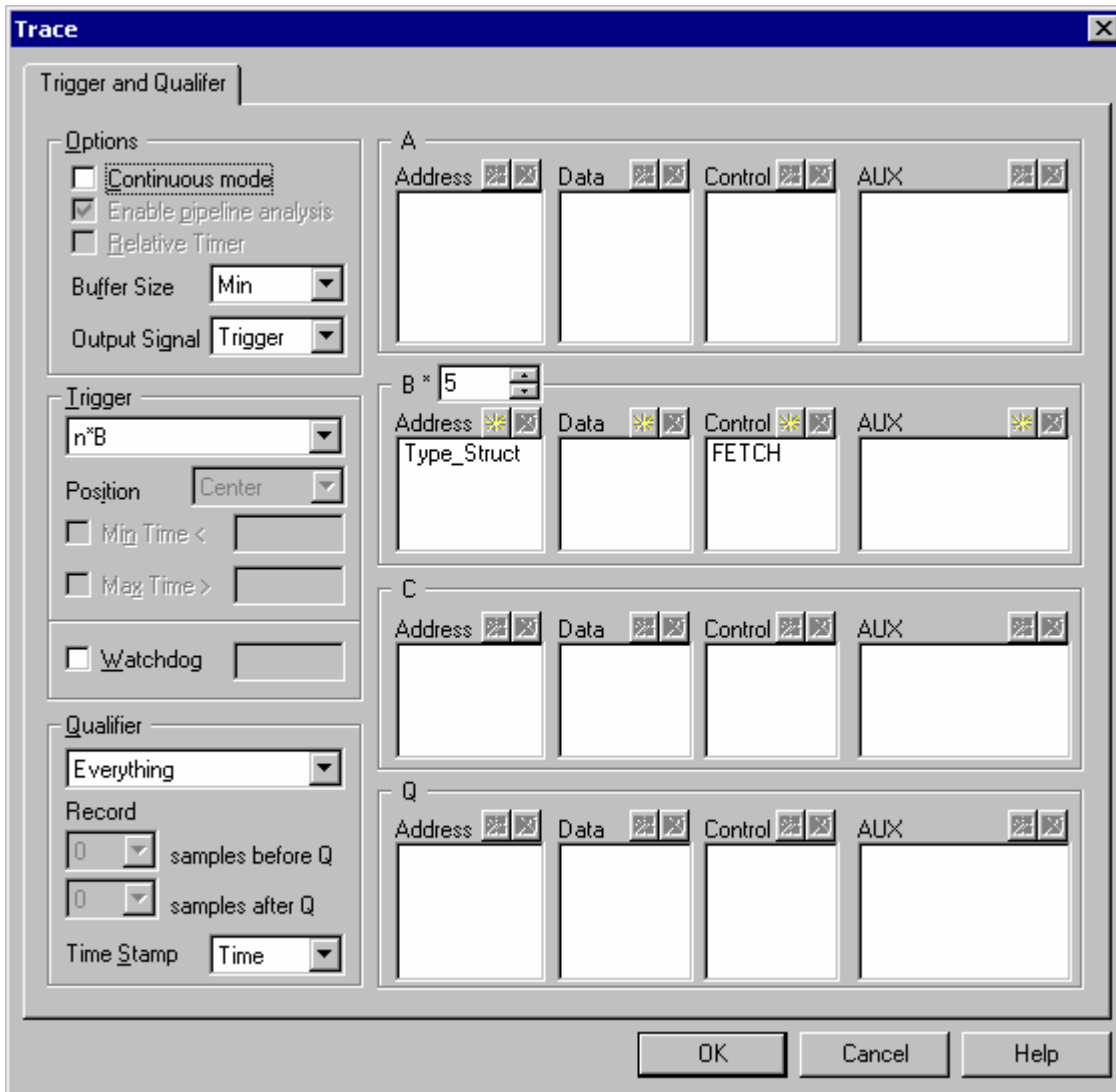


Figure 15. Trigger and Qualifier Configuration dialog

Let's inspect the results (Figure 16). Trigger point (frame 0) holds fifth `Type_Struct` function entry.

Number	Address	Data	Content	Time
-7	3FFB	0000	Data Read	-233 ns
-6	3FFB	0000	Data Read	-185 ns
-5	F0A6	1628	Type_Struct(); //2 JSR Type_Struct (F213) OP-CODE Fetch	-153 ns
-3	F0A8	13F2	Code Read	-121 ns
-2	F0AA	F316	Code Read	-65 ns
-1	3FFB	F0AA	Memory Write	-33 ns
0	F212	1B3D	void Type_Struct() Type_Struct LEAS -0011,SP OP-CODE Fetch	0 ns
1	F214	EFF1	Code Read	33 ns
2	F216	20FC	if(iResultOK==0x7CDA) LDD iResultOK (200C) OP-CODE Fetch	66 ns
3	F218	8C0C	CPD #7CDA OP-CODE Fetch	116 ns
4	F21A	DA7C	Code Read	150 ns

Figure 16. Trace Window results

Example: The code writing 0x166 to the `iCounter` variable needs to be analyzed. The same trace configuration can be used to verify that the variable never gets written 0x166 as for instance expected from the application. In such case, the trigger would never occur normally.

- 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.
- Configure 0x166 memory write to the `iCounter` variable as the event A. Set `iCounter` address in the 'Address Item' dialog and check 'Cover entire object range' option.

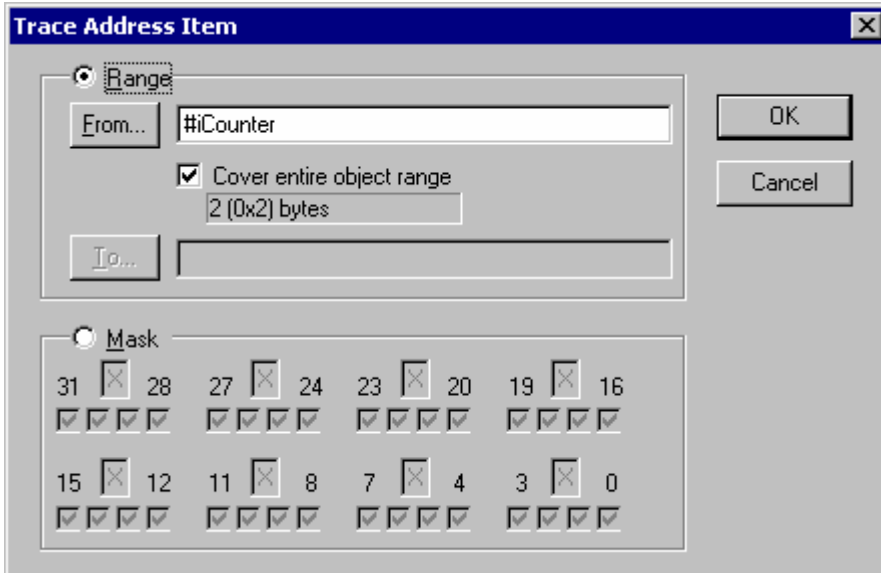


Figure 17. Event A Address Item dialog

- Next write a value in the 'Data Item' dialog and select proper data size. Maximum 'Data Item' size is limited by the CPU architecture. Only 8 and 16-bit values are valid and can be monitored properly in case of 16-bit CPU. Correct triggering on 32-bit variable requires code arming.

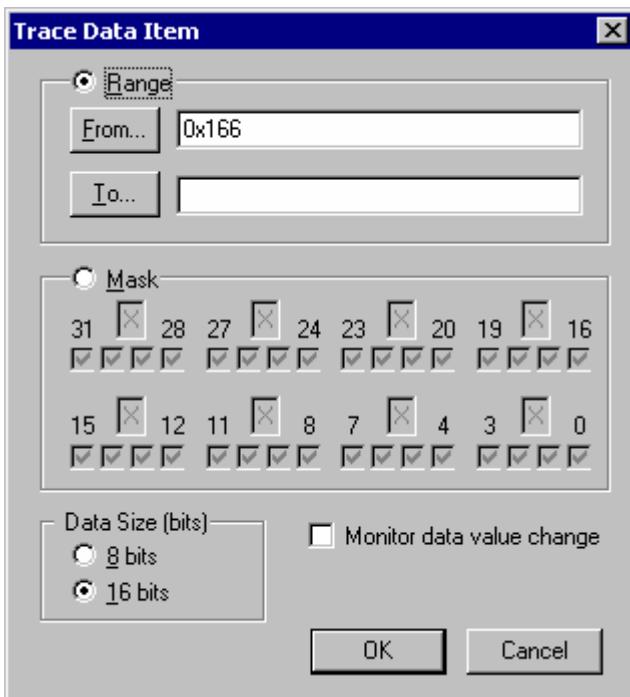


Figure 18. Event A Data Item dialog

- Set 'Memory Write' cycle type in the 'Control Item' dialog.

The trace is configured. Figure 19 depicts current trace settings. Initialize the complete system, start the trace and run the program.

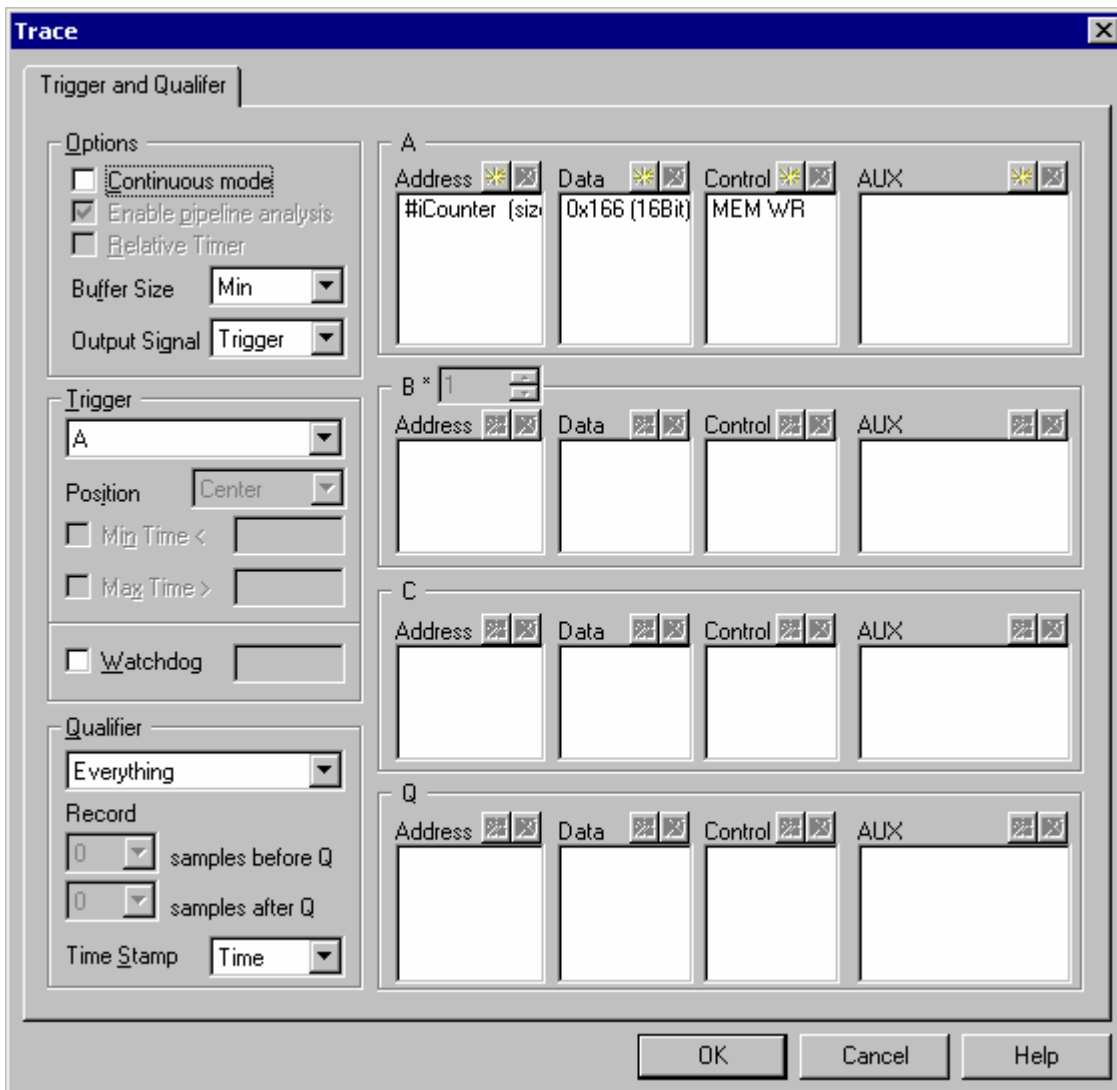


Figure 19. Trigger and Qualifier Configuration dialog

If a 0x166 write to the `iCounter` variable occurs, the trace triggers and displays the code that caused the write (Figure 20).

Number	Address	Data	Content	Time	AUX
-6	3FF9	0078	Memory Write	-250 ns	0000
-5	2000	0000	iCounter iCounter+1 Data Read	-200 ns	0000
-4	F06E	3100	} PULY OP-CODE Fetch	-167 ns	0000
-3	F06E	3100	Code Read	-133 ns	0000
-2	F070	373D	CPU_Recursion_EXIT_ RTS OP-CODE Fetch	-83 ns	0000
-1	F072	05C6	Code Read	-50 ns	0000
0	2000	0166	iCounter iCounter+1 Memory Write	0 ns	0000
1	2000	0000	iCounter iCounter+1 Data Read	33 ns	0000

Figure 20. Trace Window results

Example: Timer2Int routine only is recorded.

- 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 and 'Q event' for the Qualifier.
- Define Timer2Int routine for the event Q. When configuring Address Item for the event Q, select Timer2Int routine from the Symbol Browser and check 'Cover entire object range' option. By doing so, the Address Item is configured on a range bounded by Timer2Int entry point and Timer2Int exit point. A range size is displayed below the option, being 150 bytes for Timer2Int routine.

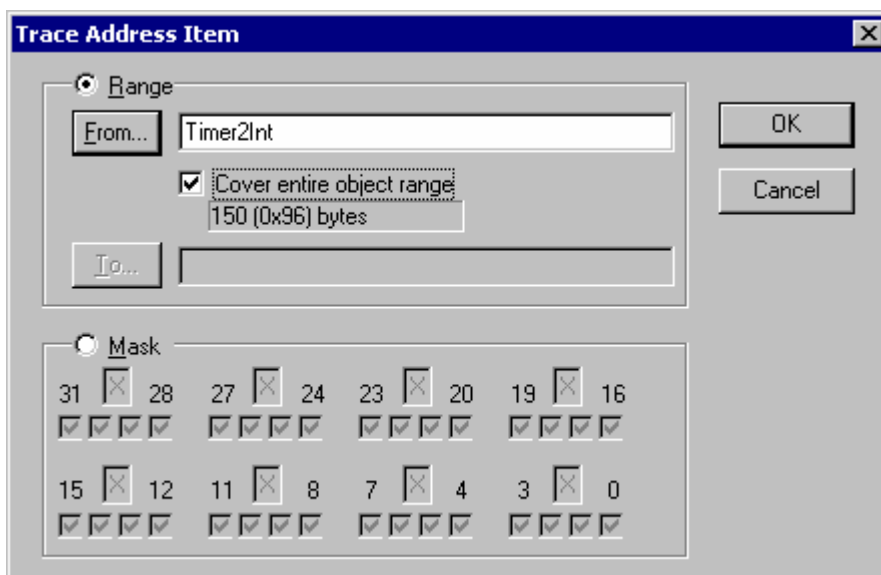


Figure 21. Event Q Address Item dialog

The trace is configured. Figure 22 depicts current trace settings.

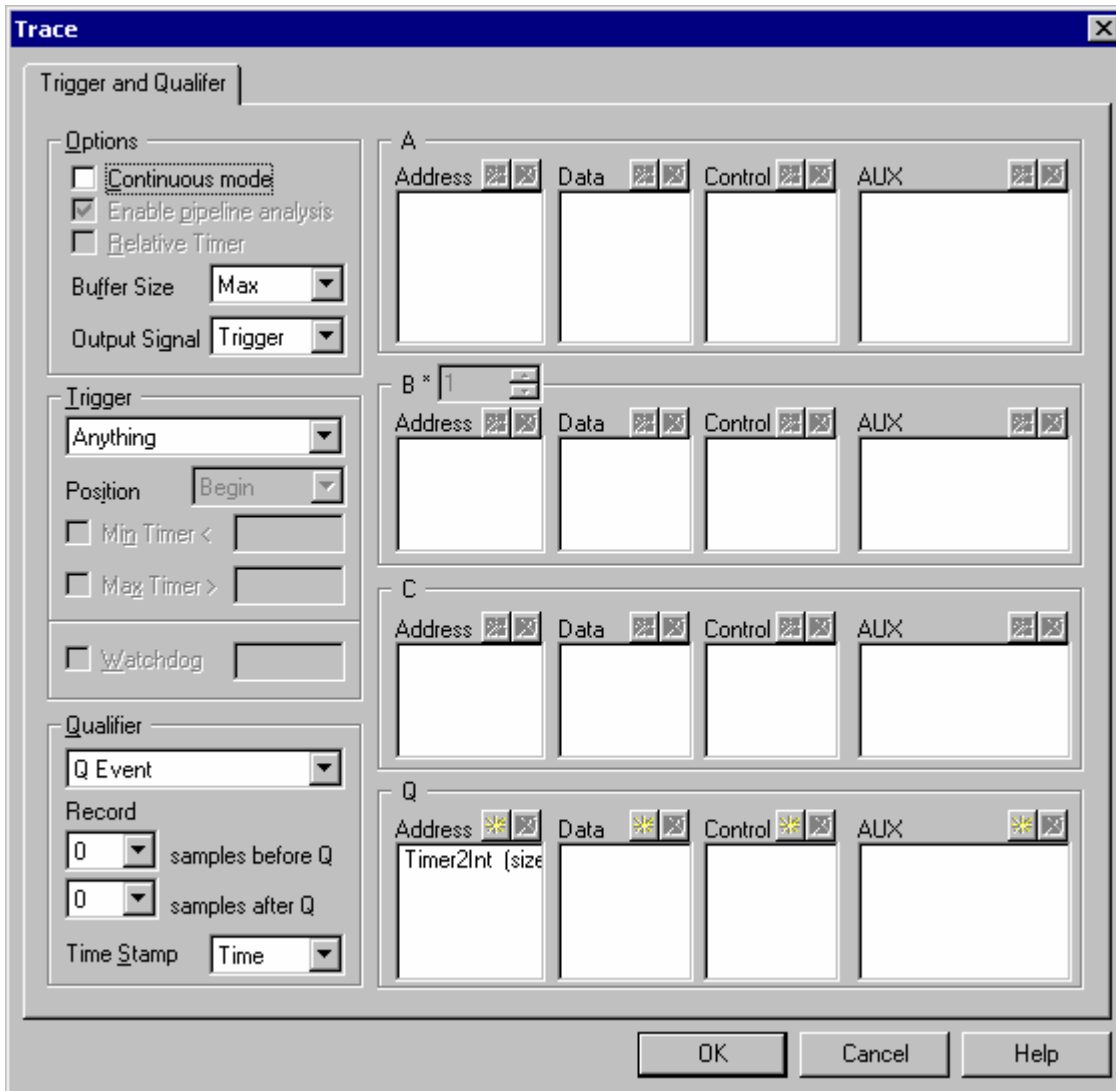


Figure 22. Trigger and Qualifier Configuration dialog

Initialize the complete system, start the trace and run the program. The trace records CPU activity only while `Timer2Int` routine is executed.

Example: Trace monitors the value of `iCounter` variable 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.
- Select 'Anything' for the trigger, 'Q event' for the Qualifier.
- Configure write to `iCounter` variable as the event Q. Set `iCounter` address in the 'Address Item' dialog and check 'Cover entire object range' option.
- Next, set 'Memory Write' for the bus cycle type in the 'Control Item' dialog.

The trace is configured. Figure 23 depicts current trace settings.

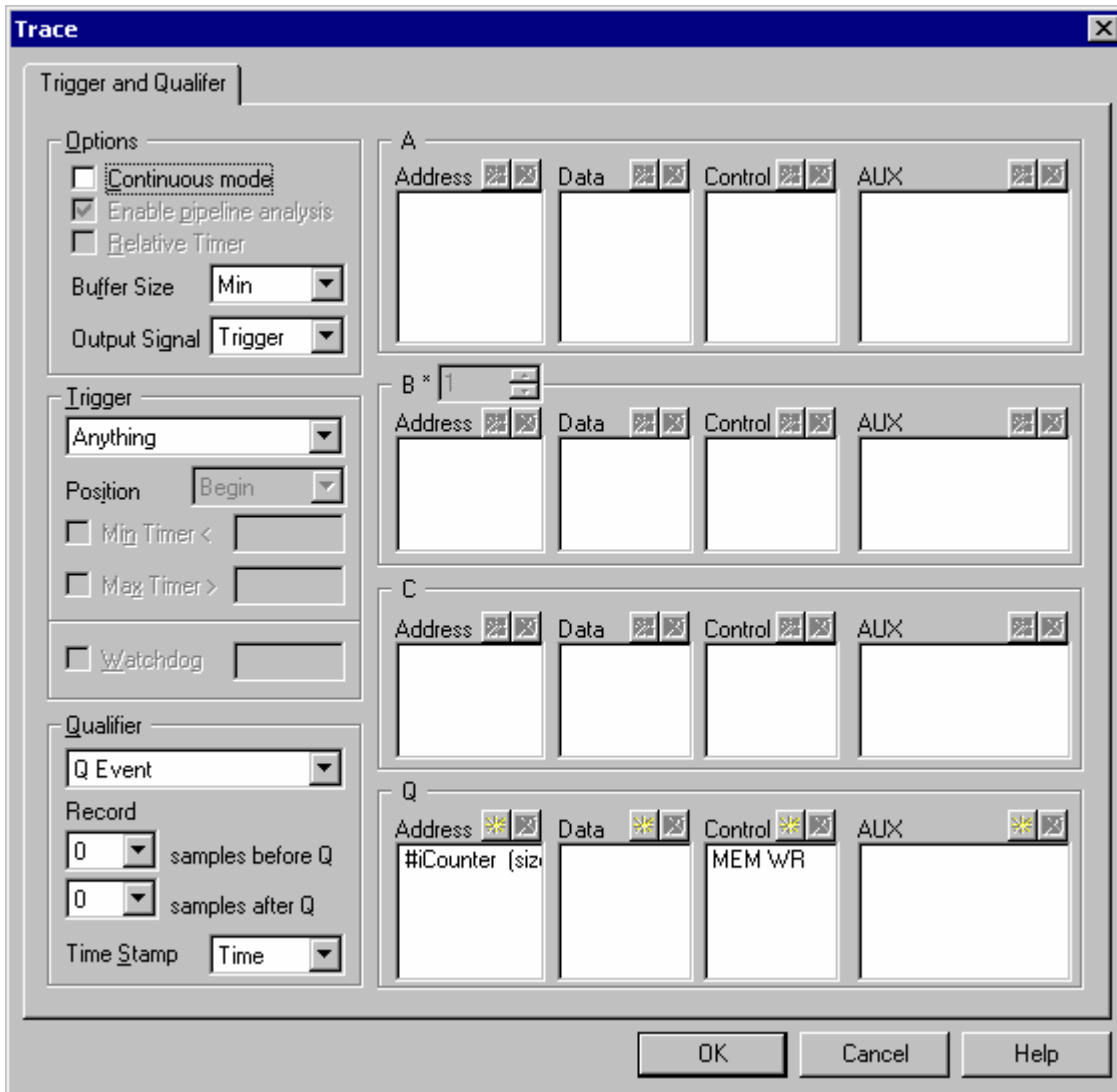


Figure 23. Trigger and Qualifier Configuration dialog

Initialize the complete system, start the trace and run the program. The trace records all writes to `iCounter` variable (Figure 24).

40	2000	005E	<code>iCounter</code> <code>iCounter+1</code> Memory Write	79.313000 ms
41	2000	005F	<code>iCounter</code> <code>iCounter+1</code> Memory Write	79.453752 ms
42	2000	0060	<code>iCounter</code> <code>iCounter+1</code> Memory Write	79.505888 ms
43	2000	0061	<code>iCounter</code> <code>iCounter+1</code> Memory Write	79.670320 ms

Figure 24. Trace Window results

5.3. 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.

Typical use:

- 1) 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. Typically, a "trace watchdog reset" condition is configured as a memory write to a specific memory location (e.g. watchdog register), which resets the watchdog timer to the start of a new time-out period. In case of the external watchdog timer being reset by the target signal, the external trace (AUX) input, where the signal must be connected, is configured, instead of a memory write. Time-out period of the trace watchdog timer must be set less than the period of the application watchdog so the trace can stop the application just before the application watchdog times out and resets the system.
- 2) If the application itself doesn't feature a watchdog mechanism, the trace watchdog trigger can be used to supervise the application. The user needs to determine a part of the code that is executed periodically and its execution period, while the program behaves expectedly. The code, resetting the trace watchdog timer, can be, for instance, a function or data write/read access. The execution period is set for the trace watchdog timer time-out period. When the application misbehaves, the "watchdog reset" code is no longer executed within the normal execution period, the trace watchdog timer times out and the trace triggers.
- 3) The code being executed a definite time after the program start can be analyzed.

Configuring Watchdog Trigger

The user needs to enter the period of the trace watchdog timer and define the "trace watchdog reset" condition, which can be logically combined from events A, B and C.

- 1) Check the 'Watchdog' option and specify the time-out period in the 'Trigger' field in the 'Trigger and Qualifier Configuration' dialog.

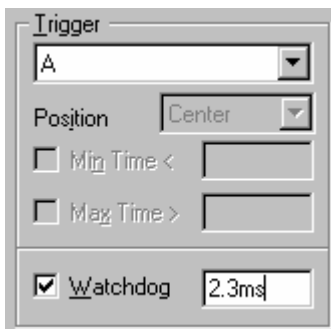


Figure 25. Trigger field

- 2) Next, define the "trace watchdog reset" condition. Typically, only the event A is selected for the "trace watchdog reset" condition and then e.g. a CPU memory write or 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.

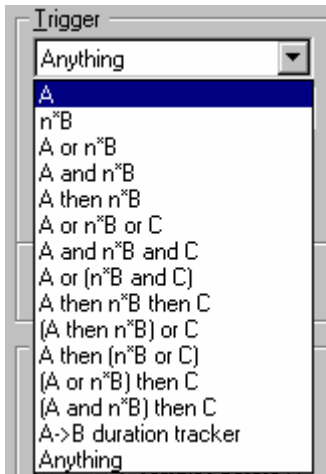


Figure 26. Trigger conditions

Example: Motorola HCS12 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 write 0x55 and 0xAA (in this order) to the ARMCOP register (0xE000) 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. Also if any value other than 0x55 and 0xAA is written, the part is immediately reset.

The COP timer time-out period is 21 ms in this particular example. It may vary between the applications since it's configurable. The watchdog timer is reset within 18 ms during the normal program flow.

The trace is going to be configured to trap COP time out and stop the program before it initiates the 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 then n*B' for the trigger condition, check the 'Watchdog' option and enter 20 ms for the trace watchdog timer time-out period.
- Next, configure the event A (reset sequence – first part) as memory write 0x55 to the address 0xE000 (ARMCOP register) and the event B (reset sequence – second part) as memory write 0xAA to the address 0xE000 (ARMCOP register).

Figure 27 depicts current trace settings.

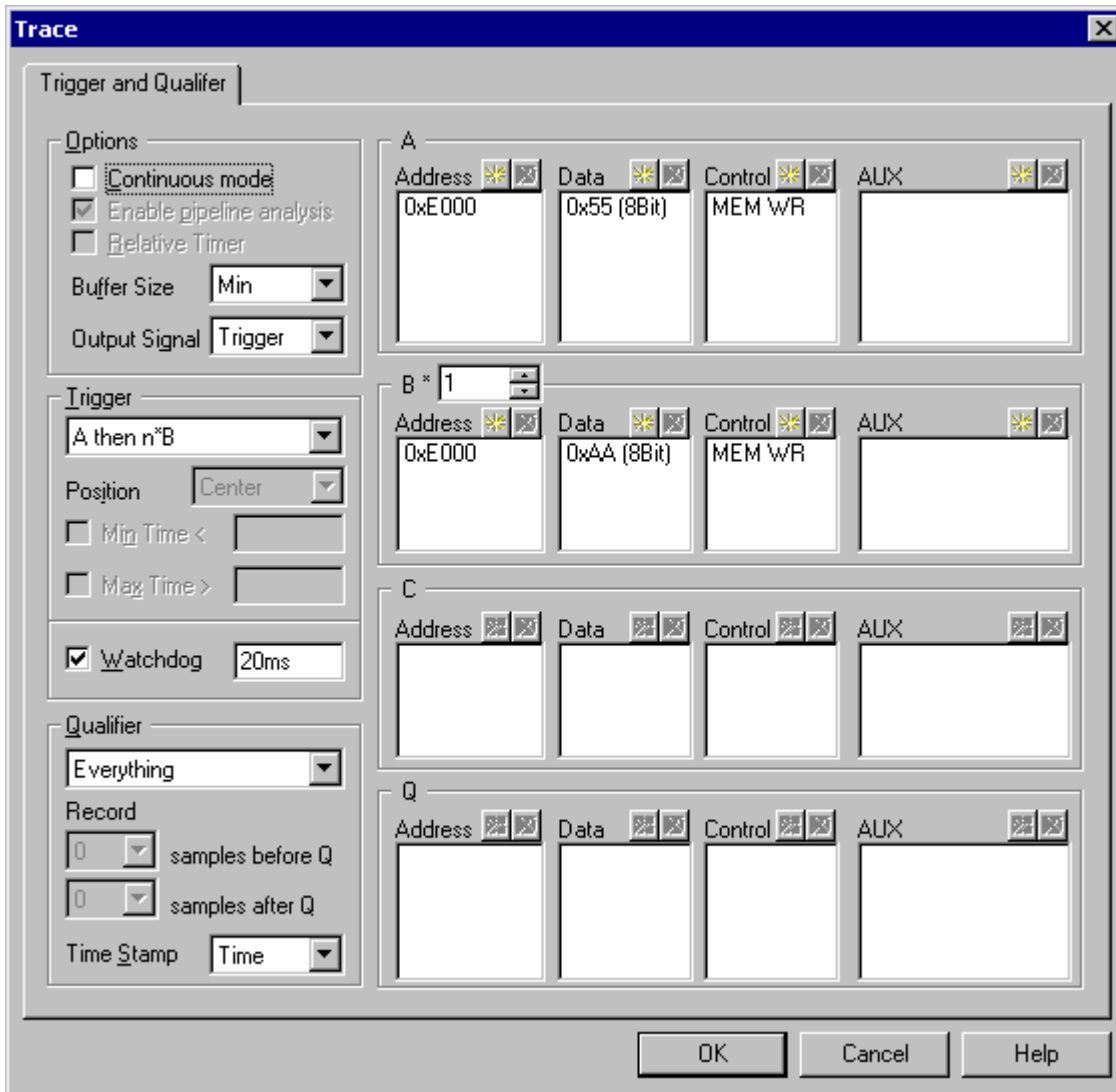


Figure 27. Trigger and Qualifier Configuration dialog

While the application operates correctly, the trace never triggers. When the application misbehaves, the trace triggers and stops the program before the system reset is initiated.

If a maximum trace history is required, select 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 54.7 ms by the WDT_RESET target signal.

The trace needs to be configured to trap the target watchdog timer time out and stop the program before the system reset is initiated. Then the user can find the code where the program misbehaves using the trace history.

One of the available external trace inputs (AUX0) is used, where the WDT_RESET signal from the target is connected. Refer to the hardware reference document delivered beside the emulation system to obtain more details on locating and connecting the AUX0 input.

- 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 540 ms for the trace watchdog timer time-out period.
- Next, 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 540 ms period.

Figure 28 depicts current trace settings.

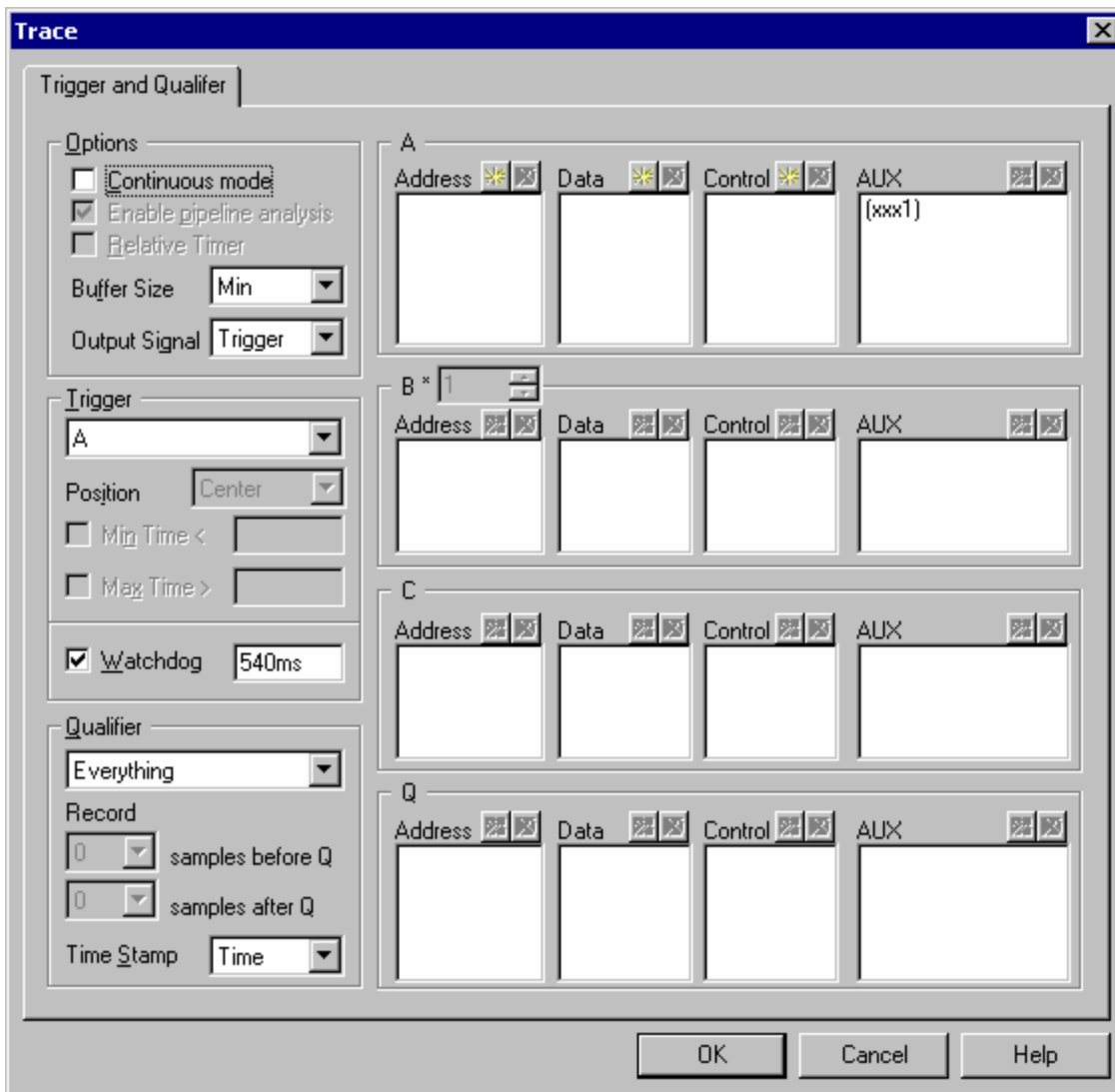


Figure 28. Trigger and Qualifier Configuration dialog

Again, while the application operates correctly the trace never triggers. However, when the application misbehaves, the trace triggers and stops the program before the system reset is initiated. If a maximum trace history is required, select maximum trace buffer size and position the trace trigger at the end of the trace buffer.

Example: The code being executed 2.3 ms after the CPU start has to be recorded.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- To record part of the program, which is distant from the CPU start point in a manner of time, select 'A->B duration tracker' for the trigger condition, check the 'Watchdog' option and enter the time when the trace triggers. Don't define anything for events A and B. This will ensure that the trace watchdog timer never resets and triggers on the first time-out that is after 2.3 ms.

Additionally, the trace can stop the program execution after 2.3 ms by checking 'On trigger break execution' option in the 'Trace Configuration' dialog. This trace configuration can be used to stop the program at any time.

Complete program flow around the trigger point can be recorded, or interesting parts only by using the qualifier. 'Q between B and C' qualifier type cannot be used in this configuration.

Figure 29 depicts current trace settings.

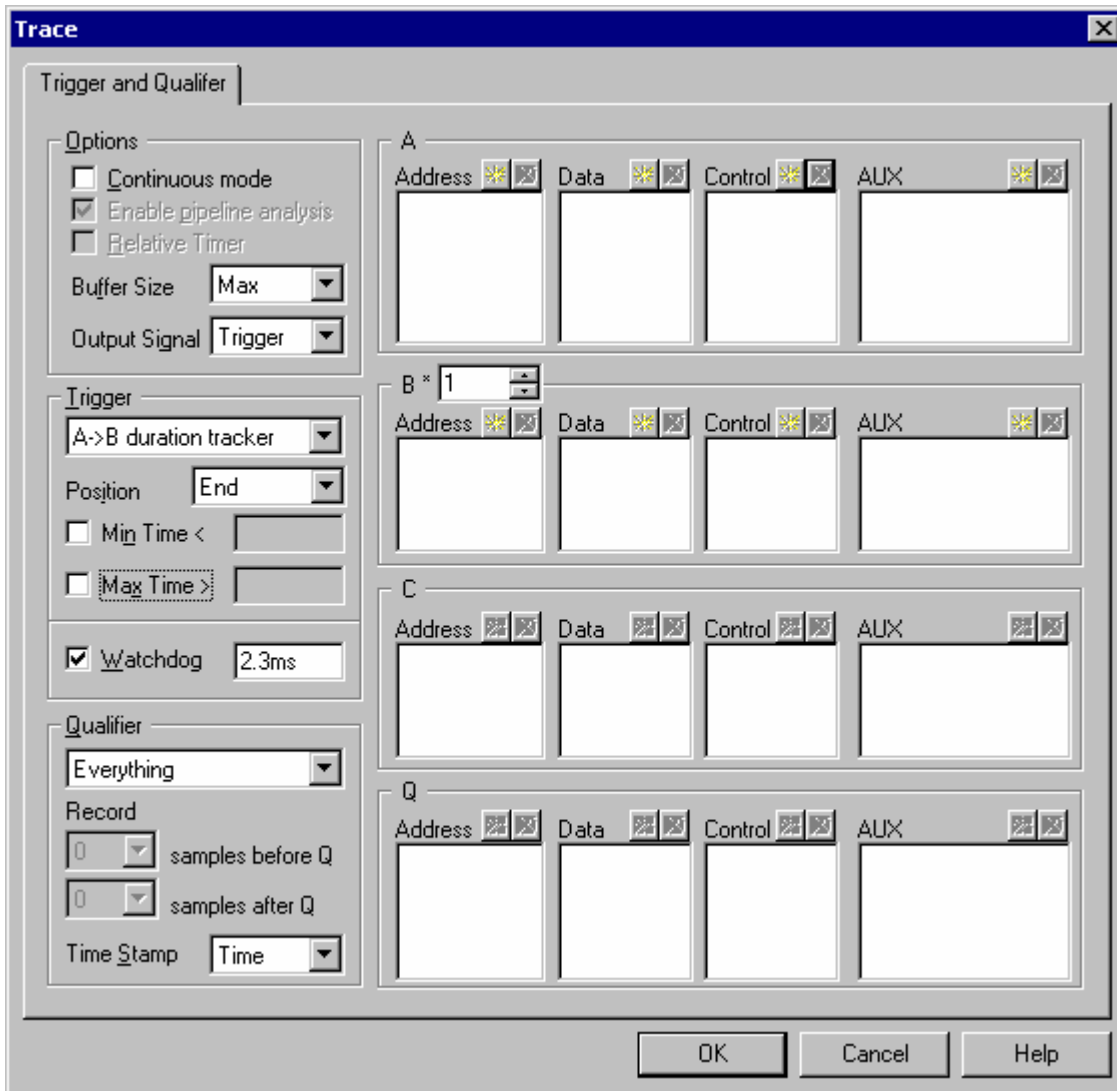


Figure 29. Trigger and Qualifier Configuration dialog

Start the session. Initialize the system, start the trace and run the application.

Figure 30 shows that the assembler instruction `INY` belonging to the `+aa;` source line of the `Type_Enum` function is executed 2.3 ms after the program start. It may be a bit confusing since the assembler instruction 'IN Y' holds 0 ns time stamp in the trace window. It's due to the fact that this is a trigger event, which always holds 0 ns time stamp. All recorded frames hold time stamp values relative to the trigger event respectively trigger frame. In this particular example, first instruction executed after reset (address `0xF000`) holds `-2.3 ms` time stamp, which is correct.

Number	Address	Data	Content	Time	AUX
-57723	F000	C787	CLRA CLRB OP-CODE Fetch (17)	-2.299905 ms	0000
-57722	F002	20CE	LDX #Var2 (2008) OP-CODE Fetch (13)	-2.299865 ms	0000
-57721	F004	2008	BRA F009 OP-CODE Fetch (15)	-2.299833 ms	0000
-57720	F004	2008	Code Read (1)	-2.299785 ms	0000
-57719	F006	6C02	Code Read (1)	-2.299753 ms	0000
-57718	F008	8E31	Code Read (1)	-2.299705 ms	0000

Number	Address	Data	Content	Time	AUX
-1	F302	ED13	++a; LDY 03,SP OP-CODE Fetch (15)	-33 ns	0000
0	F304	0283	INY OP-CODE Fetch (15)	0 ns	0000
1	F306	836D	STY 03,SP OP-CODE Fetch (13)	33 ns	0000
2	3FF7	0000	Data Read (f)	83 ns	0000
3	F308	85ED	++b; LDY 05,SP	116 ns	0000

Figure 30. Trace Window results

5.4. 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 data write/read to/from the specific memory location or an active auxiliary signal.

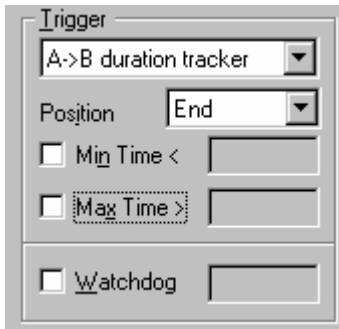


Figure 31. Trigger field

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 real time program execution. Additionally, the duration tracker can trigger and break the program, when the elapsed time between events A and B exceeds the limits defined by the user. 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.

Exceeding the limit(s) can stop the application and the code exceeding the limits can be found in the trace window.

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` 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.

Figure 32 depicts current trace settings.

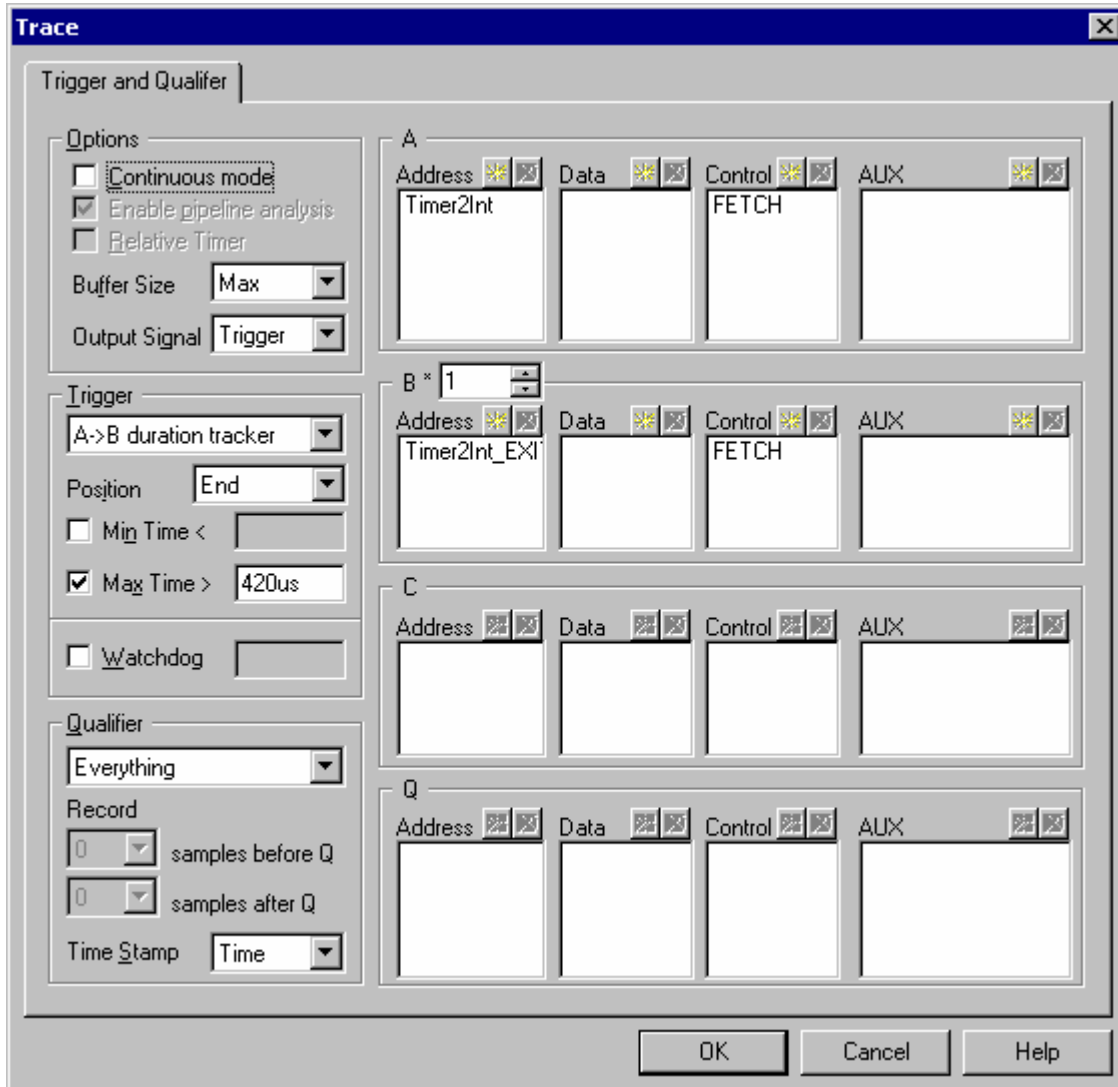


Figure 32. Trigger and Qualifier Configuration dialog

- If the program must stop when the trigger condition occurs, check 'On trigger break execution' option in the 'Trace Configuration' dialog (Figure 33). Note that the CPU stops few CPU cycles after the trigger event due to the latency of the hardware processing the trigger and stopping the CPU.

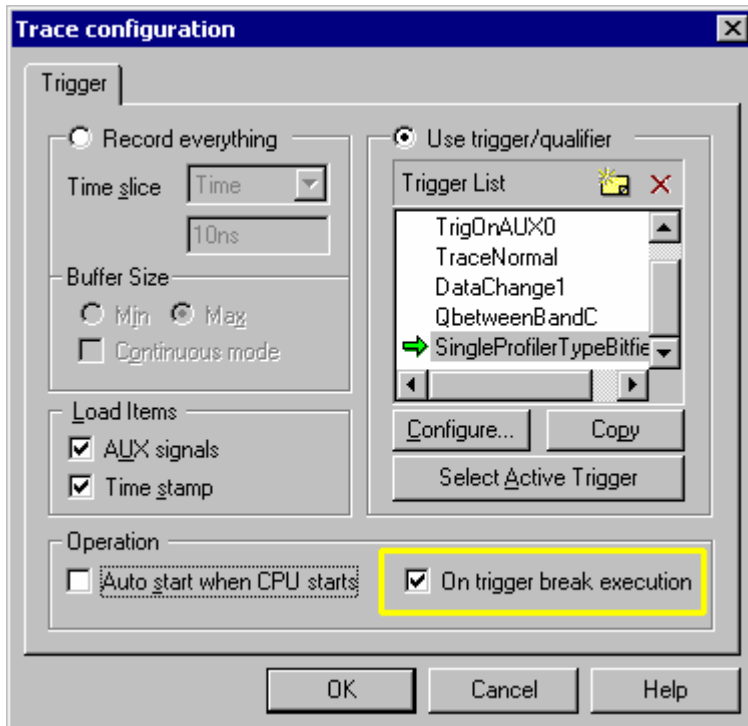


Figure 33. Trace Configuration dialog

- 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.



Figure 34. Duration Tracker Status Bar toolbar

The trace is configured. Initialize the system, start the trace and run the application. The application either runs or it is stopped by the trace trigger since we opted 'On trigger break execution' option. First, let's assume that the application behaves abnormally and the program stops by the trace trigger. 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	Data	Content	Time	AUX
-6	F2B2	1AB1	LEAX D,SP OP-CODE Fetch	-234 ns	0000
-5	F2B2	1AB1	Code Read	-200 ns	0000
-4	F2B4	ECF6	LDD 02,SP OP-CODE Fetch	-150 ns	0000
-3	3FD7	0004	Memory Write	-117 ns	0000
-2	3FD7	0000	Data Read	-84 ns	0000
-1	3FD7	0000	Data Read	-34 ns	0000
0	F2B6	5982	ASLD OP-CODE Fetch	0 ns	0000
1	F2B6	5982	Code Read	50 ns	0000
2	F2B8	1A59	ASLD LEAX D,X OP-CODE Fetch	83 ns	0000
3	F2B8	1A59	Code Read	116 ns	0000
4	3FDB	0000	Data Read	166 ns	0000

Figure 35. 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%)

Figure 36. 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 run for 27.88 ms.

If the `Timer2Int` routine doesn't exceed Min Time nor 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) profiling 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.

5.5. Q between B and C events

In this mode, the trace records particular CPU cycles, which are executed within the specific section of the code.

Example: The user wants to record memory writes to the `iCounter` variable, while function `Type_Pointers` is executed.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Select 'Q between B, C events' in the Qualifier combo box. Event B represents start and the event C stop trace recording condition. Recorded CPU activity can be further filtered out by the qualifier event Q.



Figure 37. Qualifier field

- Configure `Type_Pointers` entry point for the event B and `Type_Pointers` exit point for the event C event.
- Finally, configure `iCounter` memory write cycle for the event Q.

As soon as the event B occurs, the trace starts considering the qualifier Q and records the CPU cycles matching with the Q condition until the event C. Event B represent enable and event C disable condition for the qualifier Q.

Figure 38 depicts current trace settings.

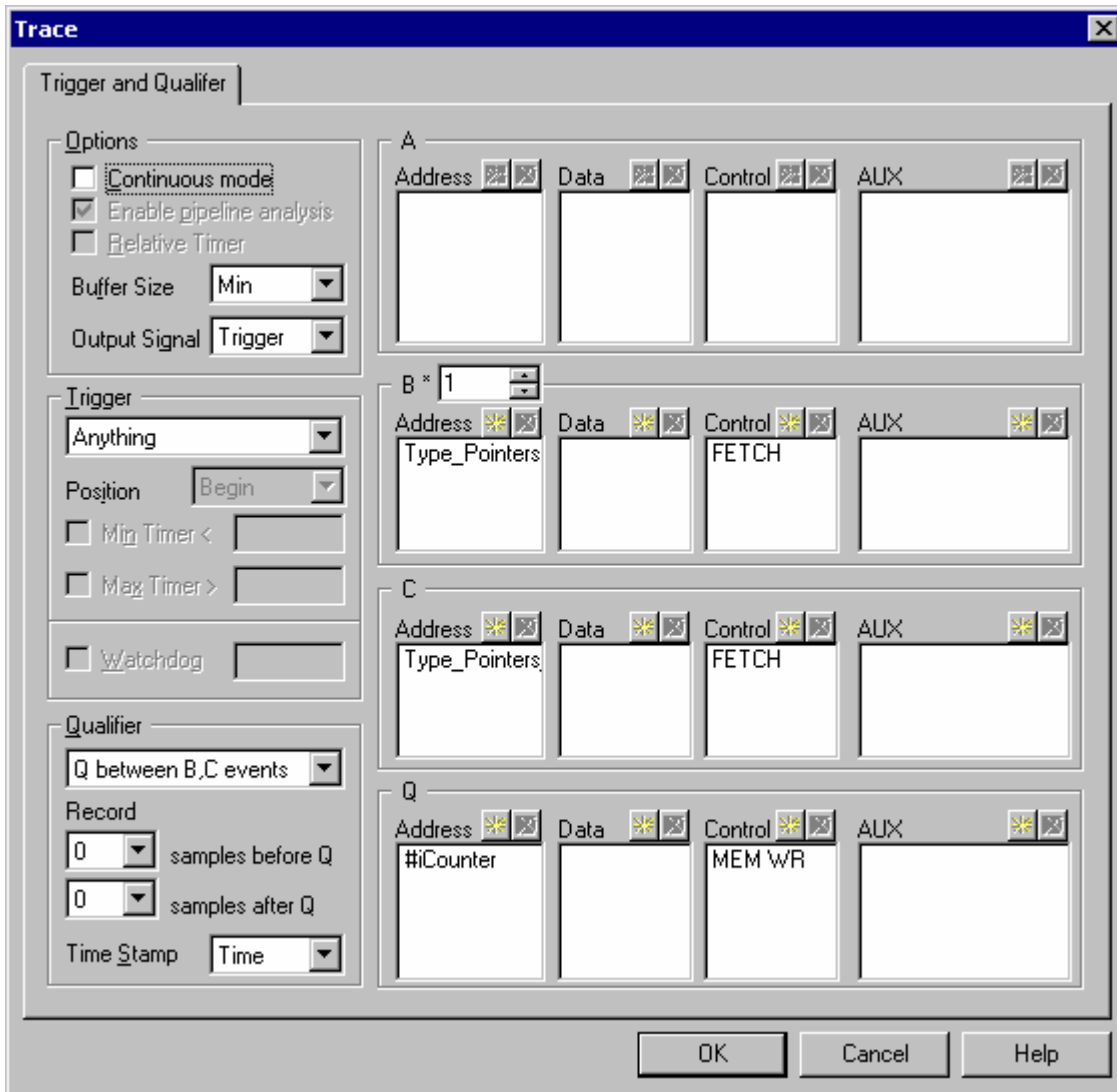


Figure 38. Trigger and Qualifier Configuration dialog

Figure 39 shows the trace results. `iCounter` variable is modified by the `Type_Pointers` function approximately every 805.7 μ s.

Number	Address	Data	Content	Time	AUX
0	2000	0004	<code>iCounter</code> <code>iCounter+1</code> Memory Write	0 ns	0000
1	2000	0010	<code>iCounter</code> <code>iCounter+1</code> Memory Write	805.683 μ s	0000
2	2000	001C	<code>iCounter</code> <code>iCounter+1</code> Memory Write	1.624150 ms	0000
3	2000	0028	<code>iCounter</code> <code>iCounter+1</code> Memory Write	2.455433 ms	0000

Figure 39. Trace Window results

Due to the CPU behaviour, it's possible that the `iCounter` memory write is not recorded by the trace (at current trace settings) although being modified by the application. Assume that we have the following function:

```

void Type_Pointers()
{
    char c;
    char *pC;
    char **ppC;

    c='A';
    pC=&c;
    ++c;
    ++*pC;
    ppC=&pC;
    ++(**ppC);
    ++iCounter;
}

```

The program flow recorded by the trace would be:

Number	Address	Data	Content	Time
15	3FF9	3FF7	Memory Write	533 ns
16	3FF9	0300	Data Read	566 ns
17	3FF9	0300	Data Read	616 ns
18	F206	20FD	++iCounter; LDY iCounter (2000) OP-CODE Fetch	650 ns
19	3FF7	0000	Data Read	700 ns
20	F208	0200	INY OP-CODE Fetch	733 ns
21	F208	0200	Code Read	766 ns
22	3FF6	0000	Data Read	816 ns
23	F20A	207D	STY iCounter (2000) OP-CODE Fetch	850 ns
24	3FF6	F644	Memory Write	900 ns
25	2000	0000	iCounter iCounter+1 Data Read	933 ns
26	F20C	1B00	} LEAS 05,SP OP-CODE Fetch	966 ns
27	F20C	1B00	Code Read	1.016 us
28	F20E	3D85	Type_Pointers_EXIT_ RTS OP-CODE Fetch	1.050 us
29	F210	F11B	Code Read	1.100 us
30	2000	01CA	iCounter iCounter+1 Memory Write	1.133 us
31	2000	0000	iCounter iCounter+1 Data Read	1.166 us
32	F212	87EF	Code Read	1.216 us
33	F212	87EF	Code Read	1.250 us
34	3FFB	0000	Data Read	1.300 us
35	3FFB	0000	Data Read	1.333 us
36	F09E	F316	Address_TestScopes(); JSR Address_TestScope OP-CODE Fetch	1.366 us

Figure 40. Trace Window results

Figure 40 shows that iCounter memory write is actually executed after the Type_Pointers function exits. Therefore, the trace configured like in the example, would not record any iCounter memory write cycles although expected by the user.

5.6. Pre/Post 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. 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.



Figure 41. Qualifier field

Example:

The application writes to the 16-bit `iCounter` variable from various functions constantly. Normally, only a function `Type_Pointers` writes `0x32` to the `iCounter` variable, which results in displaying a special graphic character on the LCD display connected to the CPU. However, it seems there is a bug in the application since the special character is displayed on the LCD also when the `Type_Pointers` is not called at all. The code writing `0x32` to the variable `iCounter` and not belonging to the `Type_Pointers` function will be found using the trace.

The trace needs to be configured to record `0x32` writes to the `iCounter`. Additionally, a section of the code executed before the write needs to be recorded to be able to find the source of the write. Using filter in the trace window, the user can filter out the code, which normally modifies the `iCounter`, and find the problematic code.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Leave the trigger set on 'Anything' by default.
- Select 'Q event' for the Qualifier and configure 'Record 8 samples before Q'.
- Finally, define the event Q. Set `iCounter` address and check 'Cover entire object range' option in the Address Item dialog (Figure 42).

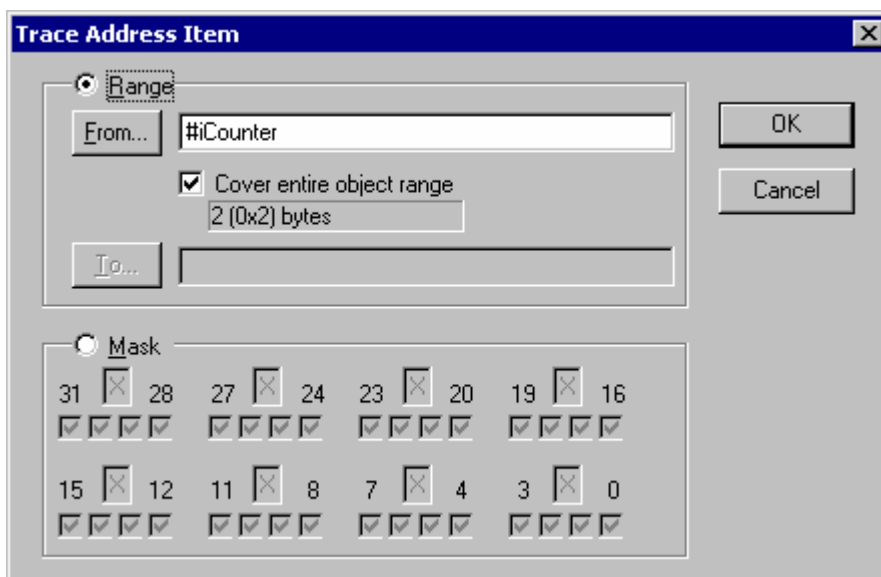


Figure 42. Event Q Address Item dialog

- Enter a value in the 'Data Item' dialog and select proper data size. Select '16 bits' data size since `iCounter` is a 16-bit variable.

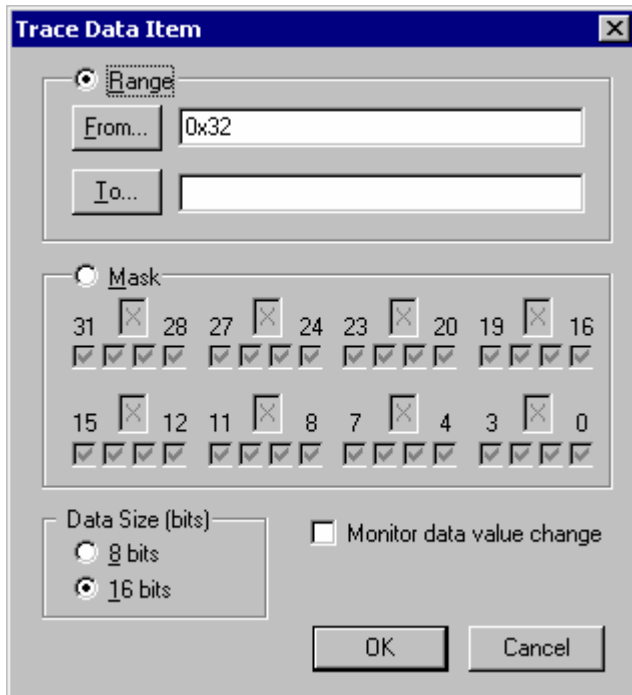


Figure 43. Event Q Data Item dialog

- Finally, set 'Memory Write' for the bus cycle type in the 'Control Item' dialog. The trace is configured.

Figure 44 depicts current trace settings.

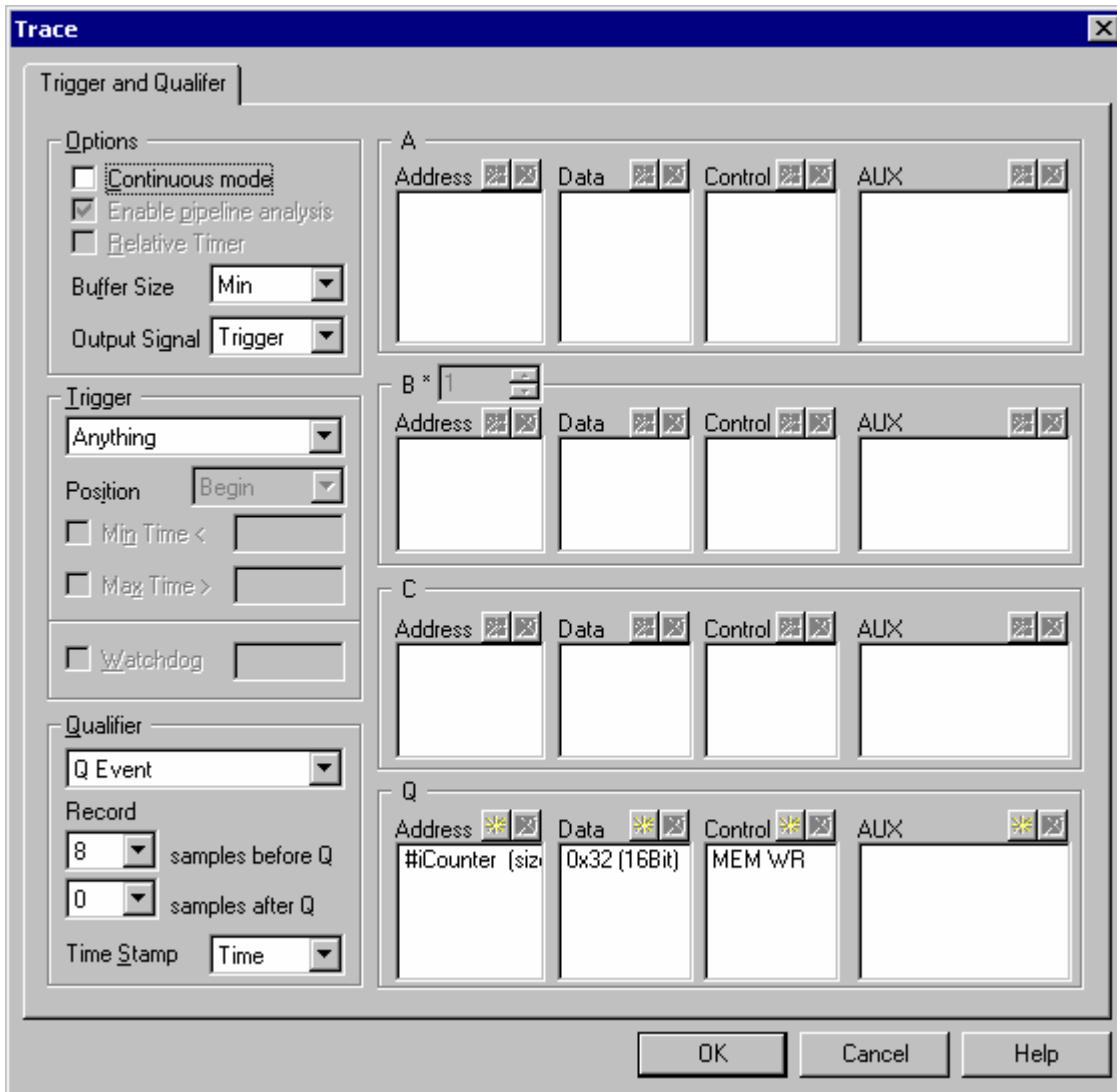


Figure 44. Trigger and Qualifier Configuration dialog

Initialize the system, start the trace and run the application. The trace records `iCounter` 0x32 memory write cycles and 8 previous CPU cycles. In this example, the trace buffer fills up slowly and the trace is stopped manually after the special character is displayed on the LCD when it shouldn't. Let's inspect the trace results (Figure 45).

Number	Address	Data	Content	Time
3785	F210	851B	} LEAS 05,SP OP-CODE Fetch	13.32870354 s
3786	F210	851B	Code Read	13.32870360 s
3787	F212	1B3D	Type_Pointers_EXIT_ RTS OP-CODE Fetch	13.32870363 s
3788	2000	0032	iCounter iCounter+1 Memory Write	13.32870368 s
3790	3FF6	0000	Data Read	13.36396448 s
3791	F20C	7D02	INY STY iCounter (2000) OP-CODE Fetch	13.36396451 s
3792	3FF6	F644	Memory Write	13.36396454 s
3793	2000	0000	iCounter iCounter+1 Data Read	13.36396460 s
3794	F20E	0020	Code Read	13.36396463 s
3795	F210	851B	} LEAS 05,SP OP-CODE Fetch	13.36396468 s
3796	F210	851B	Code Read	13.36396471 s
3797	F212	1B3D	Type_Pointers_EXIT_ RTS OP-CODE Fetch	13.36396474 s
3798	2000	0032	iCounter iCounter+1 Memory Write	13.36396480 s
3800	3FF6	0000	Data Read	13.39922560 s
3801	F20C	7D02	INY STY iCounter (2000) OP-CODE Fetch	13.39922563 s

Figure 45. Trace Window results

In the example, the trace recorded 58s of the CPU execution and encloses 8078 CPU cycles (frames). The trace buffer is quite empty comparing to the maximum possible trace buffer since there can be up to 128K frames recorded. Due to the huge amount of recorded cycles, we need to filter out all the expected CPU cycles to be able to allocate the problematic code. Some intuition is necessary to be able to find the problematic section of code. One of possible approaches will be described.

Figure 45 shows that normally Type_Pointers function exit precedes iCounter 0x32 memory writes. Save the trace record at this point (File/Save As...). Next, modify the qualifier settings in a way that only one CPU cycle is recorded before the qualifier.



Figure 46. Qualifier field

Make another trace record with new trace settings. Figure 47 shows the results.

1132	2000	0032	iCounter iCounter+1 Memory Write	13.32934453 s
1134	F212	1B3D	Type_Pointers_EXIT_ RTS OP-CODE Fetch	13.36460560 s
1135	2000	0032	iCounter iCounter+1 Memory Write	13.36460565 s
1137	F212	1B3D	Type_Pointers_EXIT_ RTS OP-CODE Fetch	13.39986673 s
1138	2000	0032	iCounter iCounter+1 Memory Write	13.39986676 s
1140	F212	1B3D	Type_Pointers_EXIT_ RTS OP-CODE Fetch	13.43512785 s

Figure 47. Trace Window Results

As before, `Type_Pointers` exit point precedes the memory write. We could start manually searching for a frame differing from the `Type_Pointers` exit point but it could take a while since the buffer can have up to 128K frames. Let's use the filter being built in the trace window to find the problematic code. Configure filter named `Filt1` set on `iCounter` memory write and filter named `Filt2` set on `Type_Pointers` exit point.

Filter expressions:

`Filt1 = (Address == {#iCounter}) & (Content = "Memory Write")`

`Filt2 = Address == {Type_Pointers_EXIT_}`

Hide states defined by these two filters by unchecking the 'State View Visibility' check boxes in the 'States and Filters' dialog.

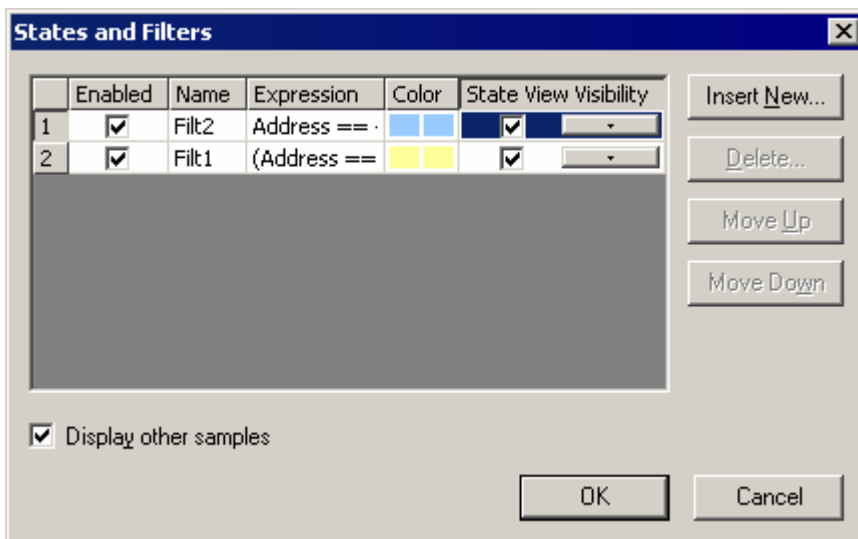


Figure 48. State and Filters dialog

Only a problematic code remains in the trace window (Figure 49).

Number	Address	Data	Content	Time
0	F1EA	3D1E	Type_Arrays_EXIT_ RTS OP-CODE Fetch	0 ns
2283	F26A	C68E	uA.m_a[2]=0x33; OP-CODE Fetch	26.86754676 s

Figure 49. Trace Window results - problematic code allocated

The problematic code is allocated at address 0xF1EA and 0xF26A. Double-click on the assembler instruction (first frame) or on the source line (second frame) points out the code being the source of the problem. If more CPU cycles than a single one before the problematic write are required, open the previously saved trace record and search for the problematic addresses 0xF1EA and 0xF26A. Figure 50 depicts the code at address 0xF26A writing 0x32 to the `iCounter` variable.

7606	F210	851B	Code Read	26.83435368 s
7608	2000	0032	iCounter iCounter+1 Memory Write	26.83435376 s
7610	F262	0200	INY OP-CODE Fetch	26.86754676 s
7611	200C	7CDB	iResultOK iResultOK+1 Memory Write	26.86754680 s
7612	F264	207D	STY iCounter (2000) OP-CODE Fetch	26.86754685 s
7613	2000	0000	iCounter iCounter+1 Data Read	26.86754688 s
7614	F266	C600	uA.m_a[1]=0x22; LDAB #22 OP-CODE Fetch	26.86754693 s
7615	F266	C600	Code Read	26.86754696 s
7616	F268	6B22	STAB OE,SP OP-CODE Fetch	26.86754700 s
7617	F26A	C68E	uA.m_a[2]=0x33; OP-CODE Fetch	26.86754705 s
7618	2000	0032	iCounter iCounter+1 Memory Write	26.86754708 s
7620	3FF6	0000	Data Read	26.90239120 s

Figure 50. Trace Window results

Note: An alternative to this example is to use 'Q between B and C events' trace configuration. Refer to the section 5.5. for more details on this trace configuration. Set trigger to 'Anything' and select 'Q between B, C events' in the Qualifier combo box. Event Q remains configured like in the example. Then configure `Type_Pointers` exit point for the event B and `Type_Pointers` entry point for the event C. The trace now records the code matching with the qualifier and not belonging to the `Type_Pointers` routine.

5.7. Data Change

A trigger or qualifier condition can be set on a change of a certain variable. Additionally, it can be set that the condition matches only when the data value is changing to a particular value. A single value, bit mask or a range can be entered for the value. Pre/Post Qualifier can be used supplementary.

Example: The trace records every write changing the 16-bit `iDataChangeVar` value and the belonging code modifying the value. The trace has to start recording on first `CPU_Pointers` function call.

- 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 and configure event A by entering `CPU_Pointers` entry point for the Address Item and 'OP-CODE Fetch' cycle for the Control Item.
- Select 'Q event' for the qualifier and define event Q.
- Enter the address of the `iDataChangeVar` variable in the 'Address Item' dialog either manually by writing its physical address or by picking up the `iDataChangeVar` address from the Symbol Browser. Make sure that 'Cover entire object range' option is checked too.
- In the 'Data Item' dialog (Figure 51), check the 'Monitor data value change' option and set '16 bits' for the Data Size since 16-bit variable is going to be monitored.

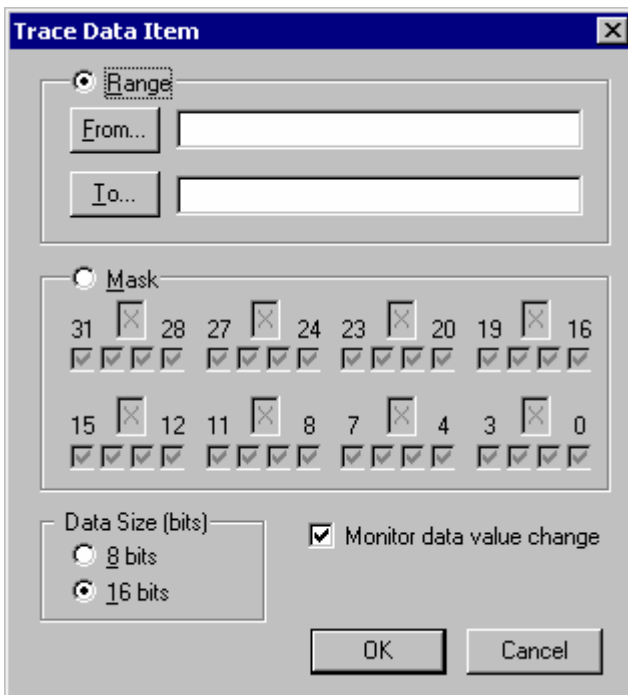


Figure 51. Event Q Data Item dialog

- Select 'Memory Write' cycle type in the 'Control Item' dialog.

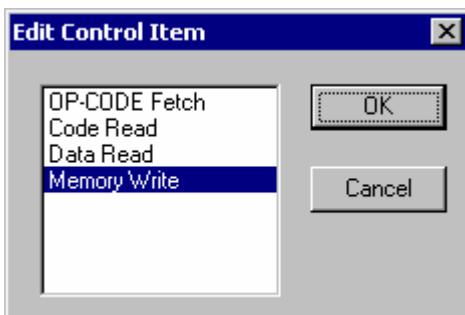


Figure 52. Q Control Item dialog

- Finally, configure Post Qualifier to record 4 samples before the event Q. This will ensure that the code (4 CPU cycles) modifying the `iDataChangeVar` variable will be recorded too. It's up to the user to record more or less cycles before the event Q.

Figure 53 depicts current trigger and qualifier settings.

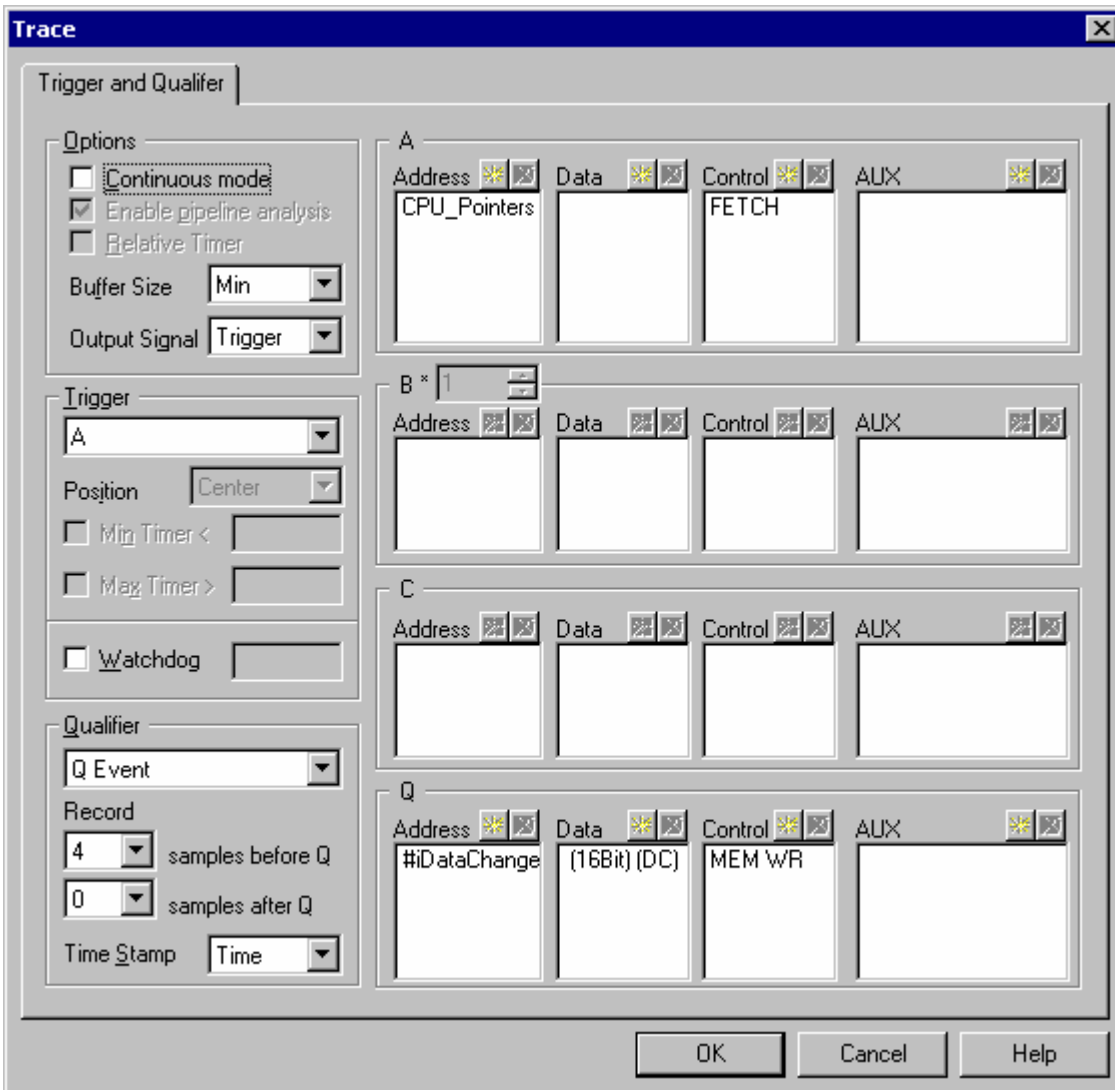


Figure 53. Trigger and Qualifier Configuration dialog

Initialize the system, start the trace and run the application. Figure 54 shows the results from the example.

Number	Address	Data	Content	Time
-8	F25A	0C20	Code Read	-442.817 us
-7	F25C	7D02	INP OP-CODE Fetch	-442.785 us
-6	200E	0002	iDataChangeVar iDataChangeVar+1 Memory Write	-442.737 us
-5	3FE7	3FEF	Memory Write	-432.905 us
-4	F3CC	C60E	uM.byte=0x6c; LDAB #6C OP-CODE Fetch	-432.849 us
-3	F3CE	6B6C	STAB 0014,SP OP-CODE Fetch	-432.817 us
-2	F3D0	14F0	Code Read	-432.785 us
-1	200E	0006	iDataChangeVar iDataChangeVar+1 Memory Write	-432.737 us
0	F13C	0E20	Code Read	0 ns
1	F13E	12CC	i=0x1234; LDD #1234 OP-CODE Fetch	50 ns
2	F13E	12CC	Code Read	83 ns
3	F140	7C34	OP-CODE Fetch	133 ns
4	200E	0001	iDataChangeVar iDataChangeVar+1 Memory Write	166 ns
5	F290	0E20	Code Read	57.166 us
6	F292	860D	bA.m_b4=0; BCLR 06,SP,#80 OP-CODE Fetch	57.200 us
7	F292	860D	Code Read	57.250 us
8	F294	C780	for(m=0;m<iCounter;m++) CLRB OP-CODE Fetch	57.283 us

Figure 54. Trace Window results

Let's analyze the results. iDataChangeVar variable changes the value at frames -1, -6, -11. Double-click on the source line or assembly instruction before the memory write points to the code writing a new value. Trigger point (CPU_Pointers function entry) itself is not recorded since the trigger point doesn't match with the configured qualifier condition.

Memory writes modifying the variable are recorded only. Memory writes writing the same value like previous memory write are not recorded. Plain qualifier condition ('Monitor data value change' option unchecked) must be used when all memory writes have to be recorded.

Example: The trace has to record every write modifying 16-bit iDataChangeVar variable to a value 0x6 and the belonging code.

- Select 'Use trigger/qualifier' operation type in the 'Trace configuration' dialog, add a new trigger and open 'Trigger and Qualifier Configuration' dialog.
- Set the trigger condition to 'Anything' and the qualifier to 'Q Event'.
- Next, configure event Q. Address and Control Item are set in the same way like in the previous example. Don't forget to check 'Cover entire object range' in the 'Address Item' dialog. Enter 0x6 value, check 'Monitor data value change' option and select '16 bits' data size for the 'Data Item' (Figure 55).
- Finally, configure the Pre Qualifier according to your needs. 'Record 4 samples before Q' is set in this example.

Figure 56 depicts current trigger and qualifier settings.

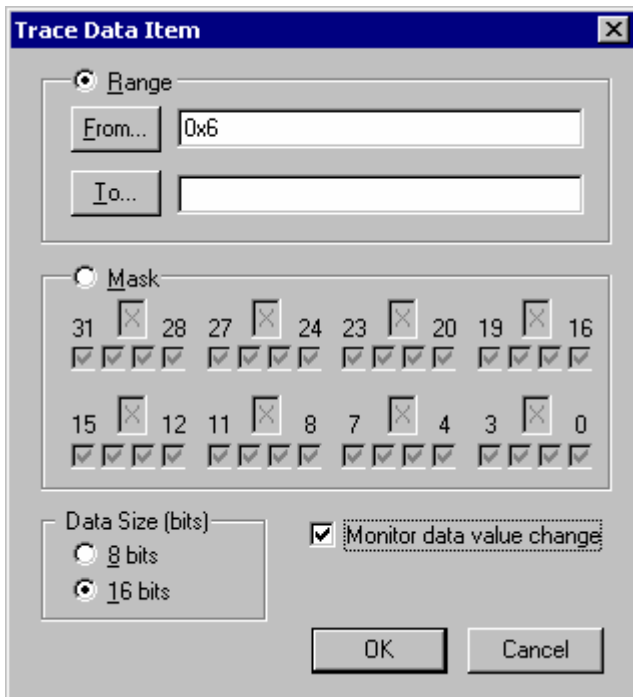


Figure 55. Event Q Data Item dialog

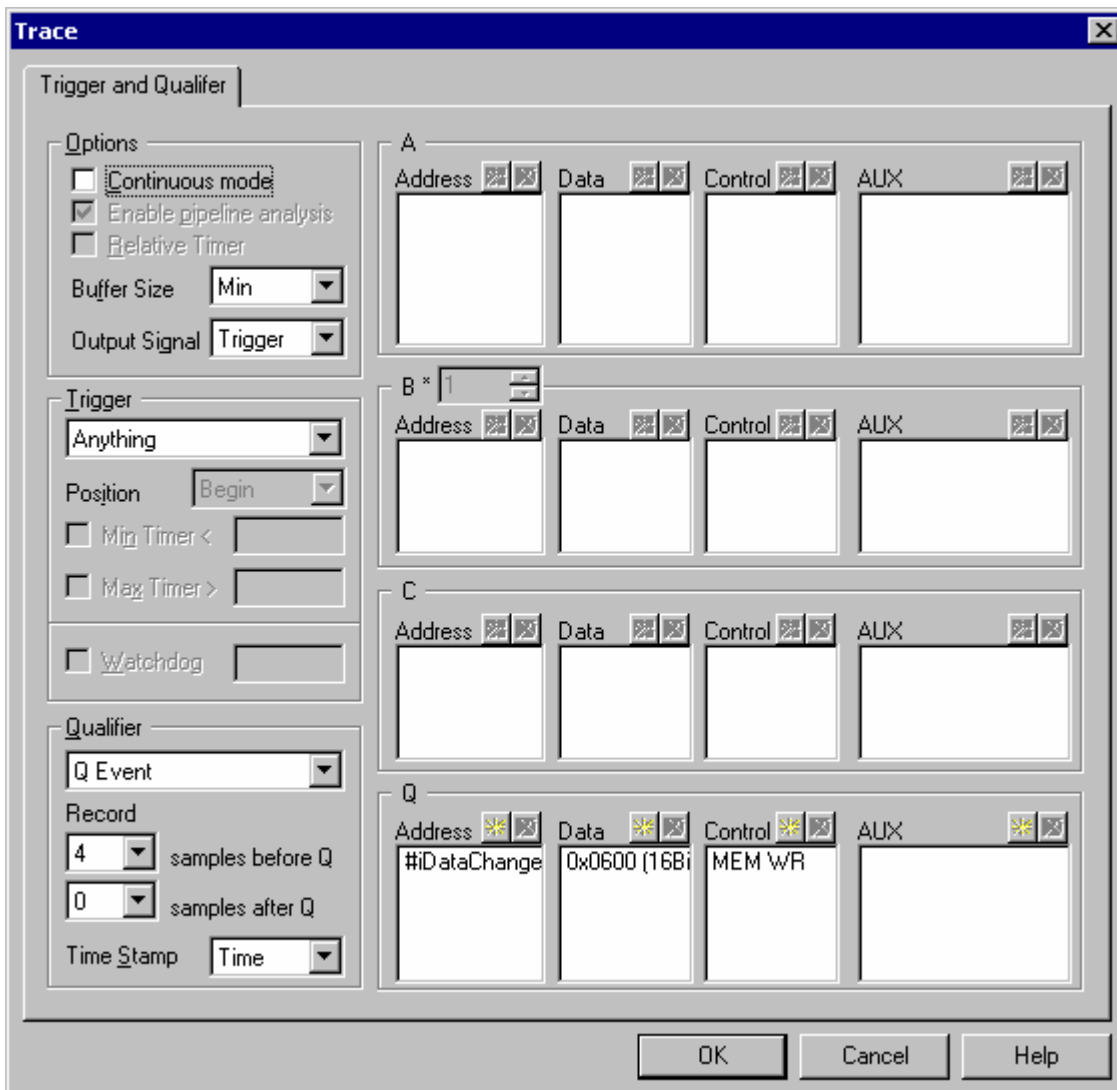


Figure 56. Trigger and Qualifier Configuration dialog

Initialize the system, start the trace and run the application. Figure 57 shows the results from the example.

8567	200E	0006	iDataChangeVar iDataChangeVar+1 Memory Write	750.632000 ms
8568	3FE7	3FEF	Memory Write	750.928116 ms
8569	F3CC	C60E	uM.byte=0x6c; LDAB #6C OP-CODE Fetch	750.928150 ms
8570	F3CE	6B6C	STAB 0014,SP OP-CODE Fetch	750.928183 ms
8571	F3D0	14F0	Code Read	750.928233 ms
8572	200E	0006	iDataChangeVar iDataChangeVar+1 Memory Write	750.928266 ms
8573	F402	C70E	for (i=0;i<2;++i) CLRB OP-CODE Fetch	751.380866 ms
8574	3FF7	0000	Memory Write	751.380916 ms
8575	F404	856C	STD 05,SP OP-CODE Fetch	751.380950 ms
8576	F406	04C6	char c=4; LDAB #PORTC (04) OP-CODE Fetch	751.381000 ms

Figure 57. Trace Window results

Figure 57 shows that iDataChangeVar variable is modified by different code. Double-click on the source lines will point out the program section writing 0x6 to the iDataChangeVar variable. In this particular example, Address_TestScopes and Type_Mixed write 0x6 to the variable.

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 September 2003. All rights reserved.