
Technical Notes

Renesas V850E2/Fx4 Family In-Circuit Emulation

Contents

| | |
|---|----|
| Contents..... | 1 |
| 1 Introduction | 2 |
| 1.1 Debug Features | 2 |
| 1.2 Power Sequencing..... | 2 |
| 2 Getting Started..... | 3 |
| 3 Emulation Options..... | 4 |
| 3.1 Hardware Options | 4 |
| 3.2 CPU Configuration | 5 |
| 3.3 Power Source and Clock | 6 |
| 3.4 Initialization Sequence | 7 |
| 3.5 JTAG Scan Speed | 8 |
| 4 Setting CPU options | 9 |
| 4.1 CPU Options | 9 |
| 4.2 Debugging..... | 10 |
| 4.3 Reset..... | 11 |
| 4.4 Nexus Options..... | 12 |
| 5 Flash Mask Options and Security Flags | 13 |
| 6 Memory Access | 14 |
| 7 Hardware Breakpoints | 14 |
| 8 OSEK Debug Support | 15 |
| 9 Trace..... | 16 |
| 9.1 On-Chip Trace Concept | 16 |
| 9.2 On-Chip Trigger and Qualifier Configuration | 16 |
| 9.3 Trace Examples..... | 18 |
| 10 Profiler..... | 27 |
| 11 Execution Coverage..... | 31 |

1 Introduction

The Renesas V850E2/Fx4 ActiveGT POD is built from the iC3000/ActiveGT platform from iSYSTEM and a special V850E2/Fx4 emulation POD produced by Renesas. The two parts are connected with a wide flex cable that carries JTAG debug and Nexus trace signals.

1.1 Debug Features

- Execution breakpoints
- Access breakpoints
- Real-time memory access
- Trace
- Profiler
- Execution Coverage
- Flash Security and Mask Options programming

1.2 Power Sequencing

For a correct emulator operation a prescribed power sequence must be followed:

- At Power On the emulator must be switched on first, and then the target. In an automated test environments the delay in between should be about 300ms or more.
- At Power Off, first switch the target off, then the emulator.

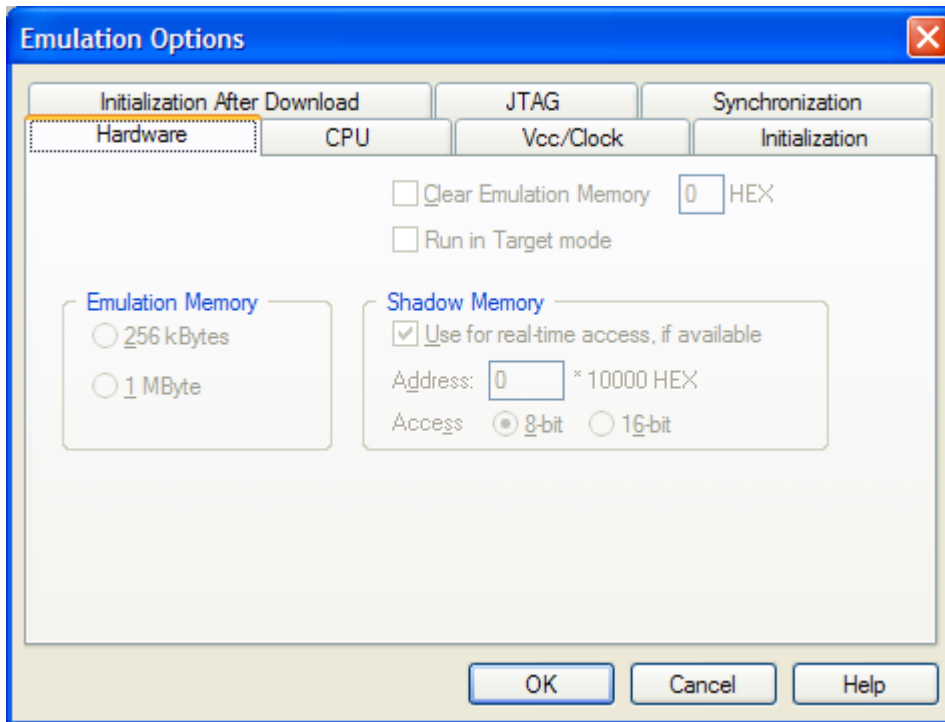
2

Getting Started

- 1) Connect the system.
- 2) Power up the emulator and then power up the target.
- 3) Execute debug reset.
- 4) The CPU should stop on reset location 0x0.
- 5) Open memory window at RAM location and check whether it is possible to modify its content.
- 6) If above steps passed successfully, the debugger is operational and code download should be possible.

3 Emulation Options

3.1 Hardware Options

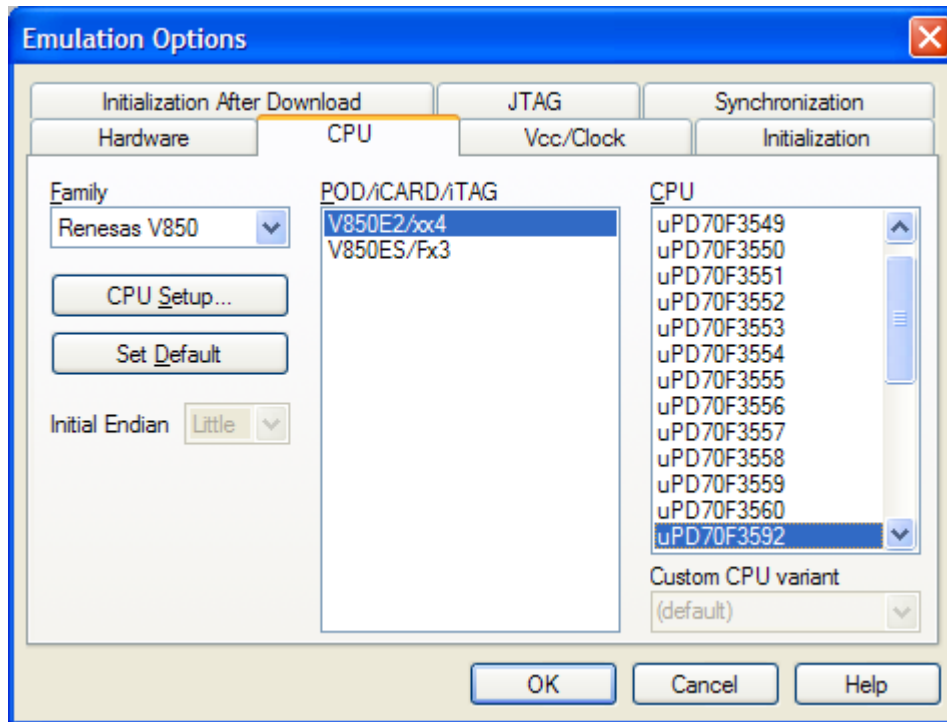


In-Circuit Emulator Options dialog, Hardware page

There are no user settable options here. The Emulation memory is actually a flash memory inside the Fx4 Umbrella device on the Renesas POD. There is also no need for the Shadow Memory as the Real-Time access is supported by the debug interface by default.

3.2 CPU Configuration

Select the emulated CPU family, the POD and the target device being emulated.



In-Circuit Emulator Options dialog, CPU Configuration page

CPU Setup

Opens the CPU Setup dialog. In this dialog, parameters like memory mapping, bank switching and advanced operation options are configured. The dialog will look different for each CPU reflecting the options available for it.

Set Default

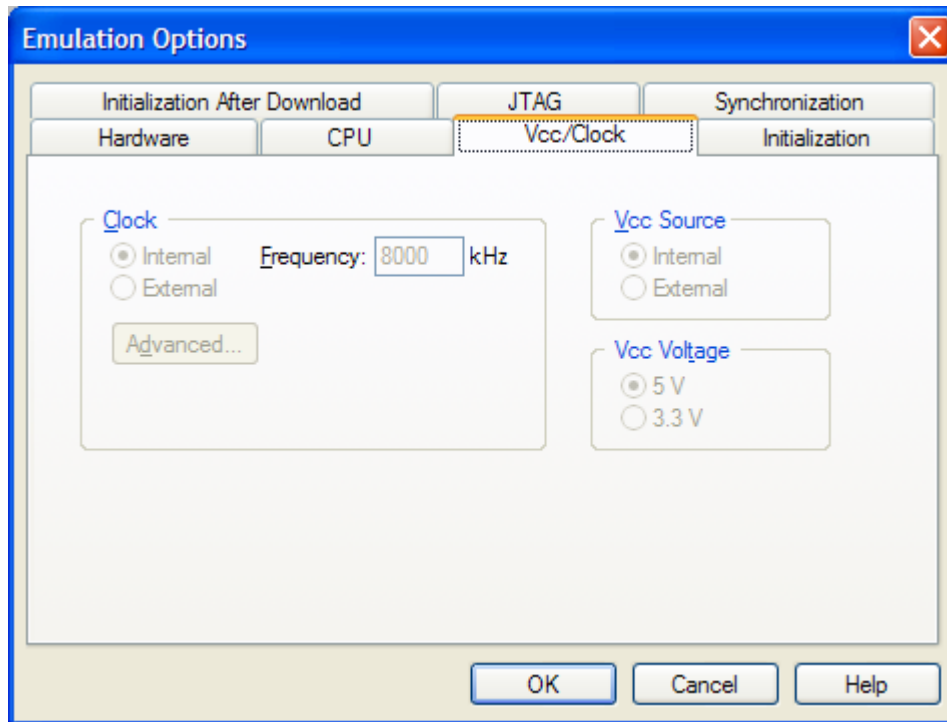
This button will set default options for currently selected CPU. These include:

- Vcc and clock source and frequency
- Advanced CPU specific options

Note: Default options are also set when the Family or a POD is changed.

3.3 Power Source and Clock

The Vcc/Clock Setup page determines the CPU's power and clock source.



In-Circuit Emulator Options dialog, Vcc/Clock Setup page

No option here, as well. The POD power is provided so that the emulator can be used standalone, that is, without being connected to a target board. As soon as the target is connected, the target power has to be applied. Note the correct sequencing of the emulator and target power.

There is no programmable clock control available from winIDEA. The Renesas POD Fx4 device with an internal clock oscillator that has a nominal frequency of about 8MHz. User program can then switch to one of the clock sources inside the Fx4 device. The Main oscillator requires an external quartz crystal. Please refer to the IC30740 POD Reference and Renesas Fx4 User Manual for details.

A 32.768kHz quartz crystal is provided on the Renesas POD for the Sub-Oscillator clock generator.

3.4 Initialization Sequence

Primarily, initialization sequence is used on On-Chip Debug systems to initialize the CPU after reset to be able to download the code to the target (CPU or CPU external) memory.

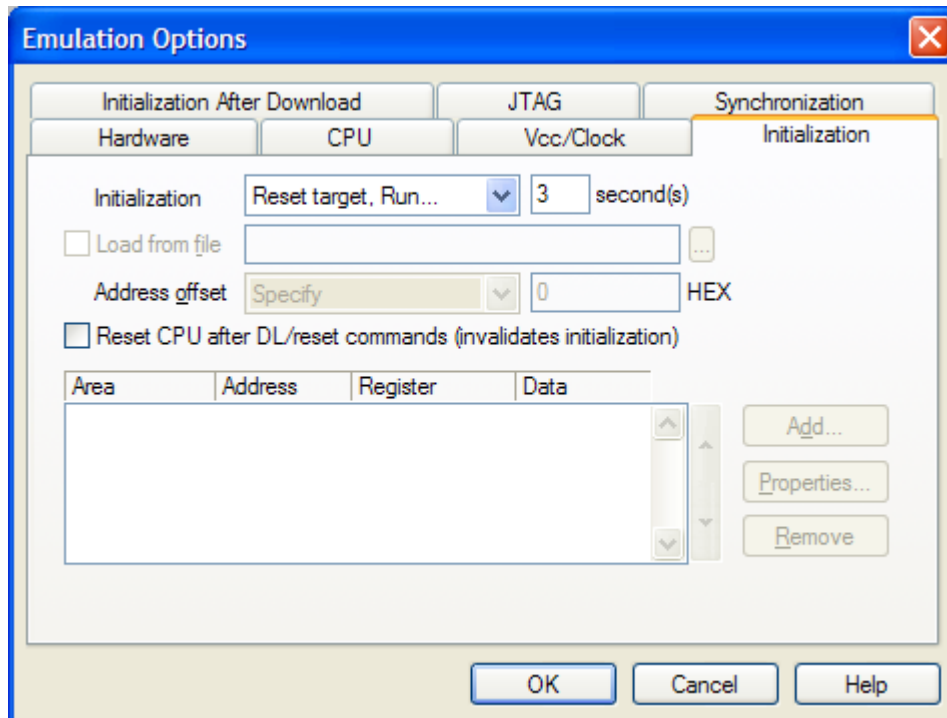
Typically, there is no need to use the initialization sequence in case of the In-Circuit Emulator emulating Single Chip mode. Initialization sequence may be required on some CPU families when it is required by the application being debugged. The debugger executes initialization immediately after reset and then downloads the code.

Basically, the initialization sequence can be set up in two ways:

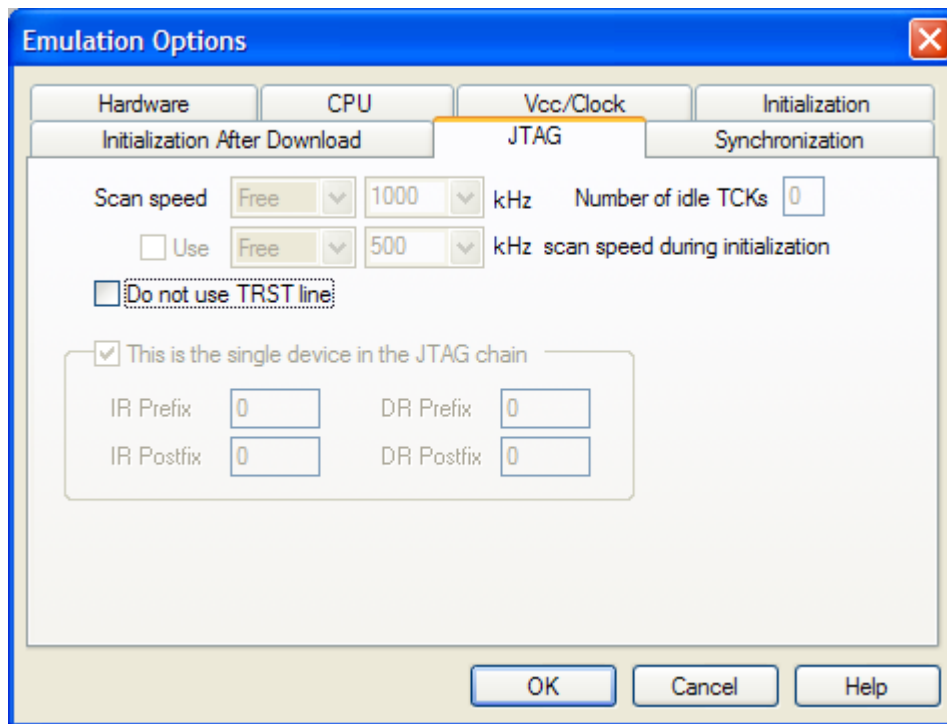
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.
2. winIDEA accepts initialization sequence as a text file with a .ini extension. The file must be written according to the syntax specified in the Appendix in the winIDEA Hardware User's Guide.

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 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.



3.5 JTAG Scan Speed



JTAG Scan Speed definition

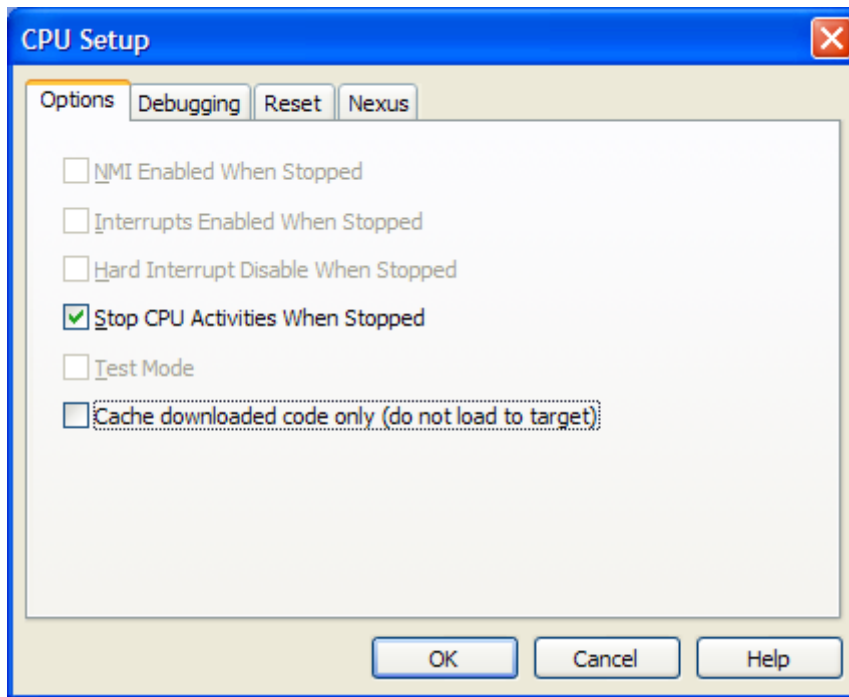
Scan speed

JTAG Scan Speed is automatically set to the Free mode. The free running JTAG clock is required by the V850E2 debug interface.

4 Setting CPU options

4.1 CPU 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.



CPU Setup, Options page

Stop CPU Activities When Stopped

When this option is checked, most of the peripheral functions are stopped when the application is stopped. For details, please refer to the Fx4 User Manual, Chapter 36, On-Chip Debug Unit. This option directly controls the SVSTOP bit in the EPC Emulation Break Control register. Note that some peripherals have an override bit to disable stopping when the CPU is stopped in debug mode.

In general, it is recommended that the option is checked in order to have more predictable behaviour of the debugged application using these peripheral functions.

Cache downloaded code only (do not load to target)

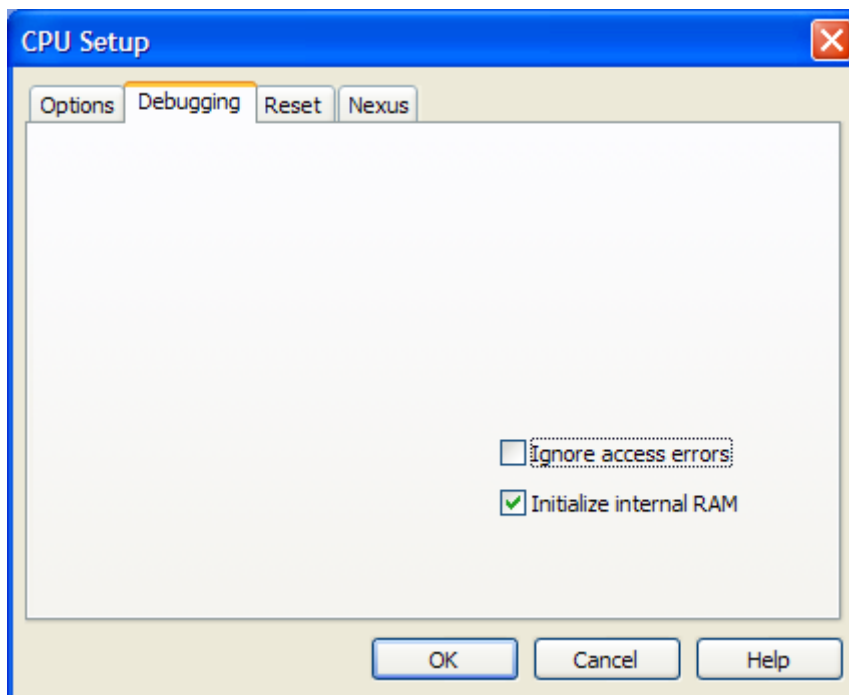
When this option is checked, the download files will not propagate to the target using standard debug download but the Target download files will. This is typically used where the flash memory has limited number program/erase cycles and when the downloaded code has already been programmed. Don't forget to uncheck the option when the program sources have changed!

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.

4.2 Debugging

This pane exposes two options related to a memory access. Displayed below are default settings. If the settings are reversed, it allows a post-mortem examination of the internal RAM that would otherwise be cleared by the debugger initialization.



Ignore access errors

Any access to an undefined address space or non-initialized internal RAM returns an error status. This is displayed with question marks '?' all over the memory window when such address base is given. However, check this option to be able to inspect the state of the internal RAM, and uncheck the 'Initialize internal RAM' option.

Initialize internal RAM

Per default this option must be checked. After reset, the microcontroller internal SRAM is not initialized and cannot be used. When the option is checked, the debugger initializes internal SRAM, which is required for internal flash programming.

Uncheck the option only when it's required to analyze the microcontroller state after the reset. Note that the Debug download will fail in this case!

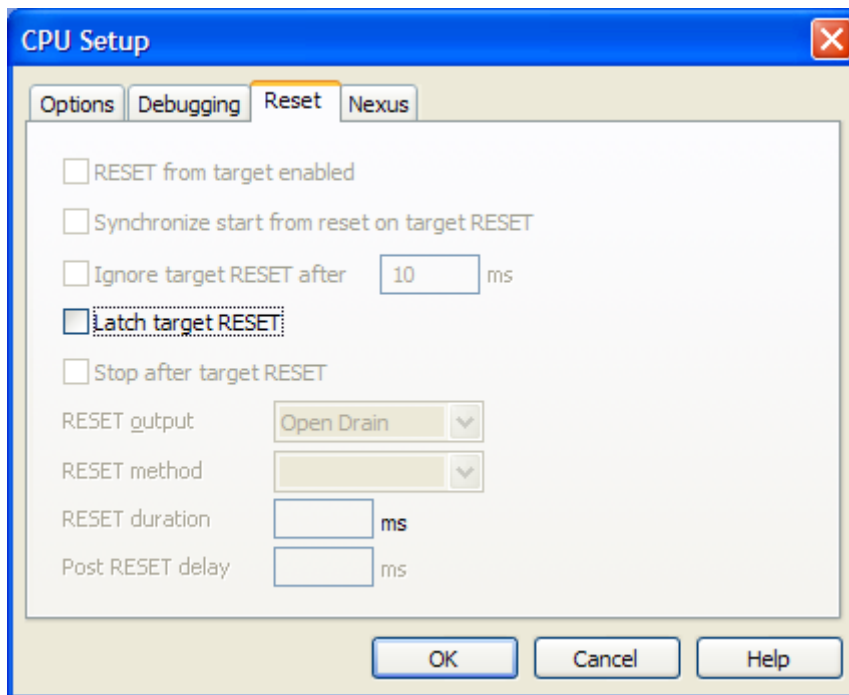
Use case would be to download the code to the flash with this option checked. Next, power off and power on complete system and then perform a debug reset only. At that point internal RAM is intact by the debugger and the microcontroller state can be inspected.

4.3 Reset

Latch target RESET

When the option is checked, the debugger latches active target reset until it gets processed. This yields a delay between the target reset and restart of the application from reset. If this delay is not acceptable for a specific application, the option should be unchecked. An example is an application where the CPU is periodically set into a power save mode and then waken up e.g. every 6ms by an external reset circuit. In such case, a delay introduced by the debugger would yield application not operating properly.

When the option is unchecked, the application is resumed after the internal (e.g. watchdog) or external target reset with a minimum delay.



4.4 Nexus Options

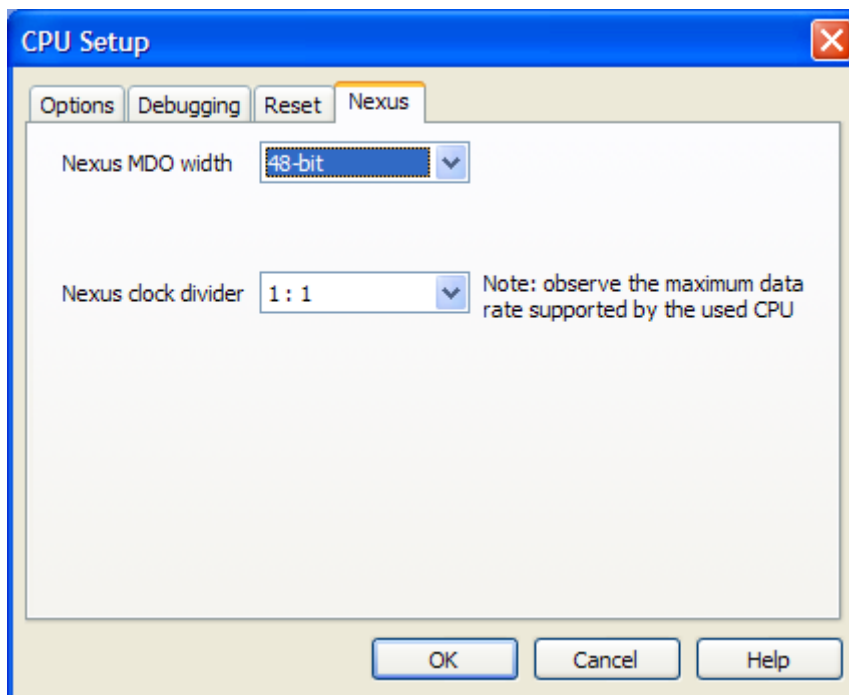
The Nexus Options dialog allows setting the Nexus MDO width and the Nexus clock divider. There is no uniform rule which setting is the best as it depends on the requirements.

16-bit port size is the default recommended value. It gives the best result in terms of session time and time accuracy. Profiler uses this configuration internally.

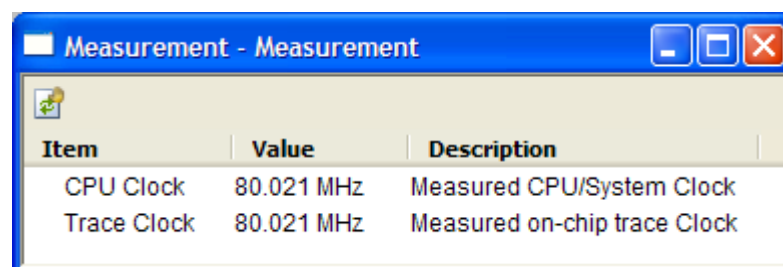
When tracing a lot of data, it is recommended to use 48-port size since 16-bit port size selection will yield overflows sooner.

Use 4-bit port size when maximum session time (tracing no or minimum data -> profiler) is required and a time stamp accuracy can be sacrificed.

Currently, the maximum Nexus trace clock frequency is 80MHz. Up to this frequency set the divider to 1:1 mode. If the core clock frequency is higher, then use lower divider settings. The lower divider settings can also be used if trace data integrity problems are suspected.



The trace clock frequency can be checked in the Plugins/Measurement window.



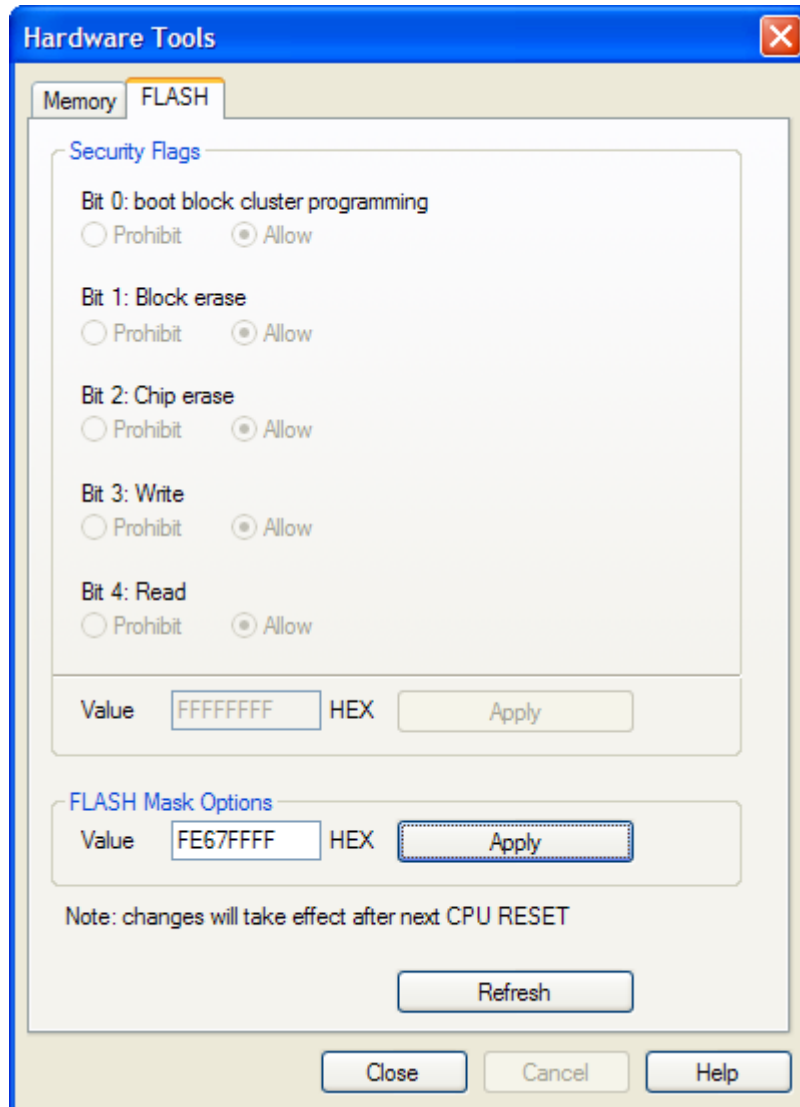
| Item | Value | Description |
|-------------|------------|------------------------------|
| CPU Clock | 80.021 MHz | Measured CPU/System Clock |
| Trace Clock | 80.021 MHz | Measured on-chip trace Clock |

The Trace Clock is measured on the Renesas POD trace connector, while the CPU Clock is calculated from the Nexus clock divider setting above. If the trace clock exceeds the hardware limits, then the trace window will contain errors or be blank even in Record everything mode. The displayed frequencies are shown for informational purposes only.

5 Flash Mask Options and Security Flags

The dialog under the *Hardware/Tools/FLASH* menu provide access to the V850E2/Fx4 Flash Mask option bits. Certain options need the CPU to go over reset to take effect. Please refer to the Fx4 User Manual.

Note: Programming of the Security Flags has been disabled to prevent locking the in-circuit emulation device. The flags are intended for end product protection, and are available for iC5000 and iC3000/iCard environments.



The picture above shows the recommended settings for the initial Flash Mask Options. The bits cleared disable both instances of the WDTA Window Watchdog Timer A.

Note: Download to flash will fail if the watchdog timer is not disabled.

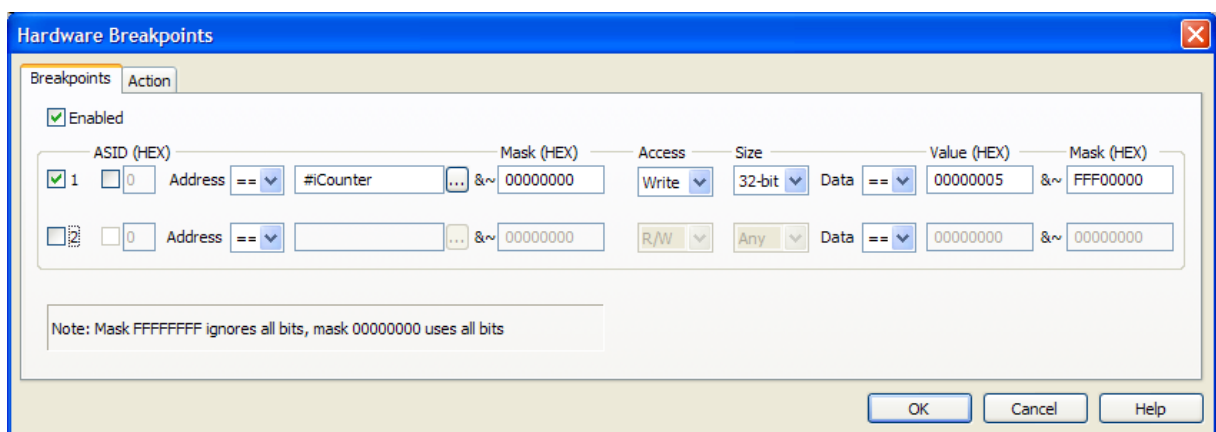
6 Memory Access

V850E2 debug interface features a real-time memory access, that does not require user program to be stopped and allows reading the memory while the application is running.

Note that large amounts of memory read affect the application performance.

7 Hardware Breakpoints

Access breakpoints are configured by opening the *Debug/Hardware Breakpoints* dialog. Two breakpoints are provided. Note that they are shared with the execution breakpoints from the *Debug/Breakpoints* dialog.

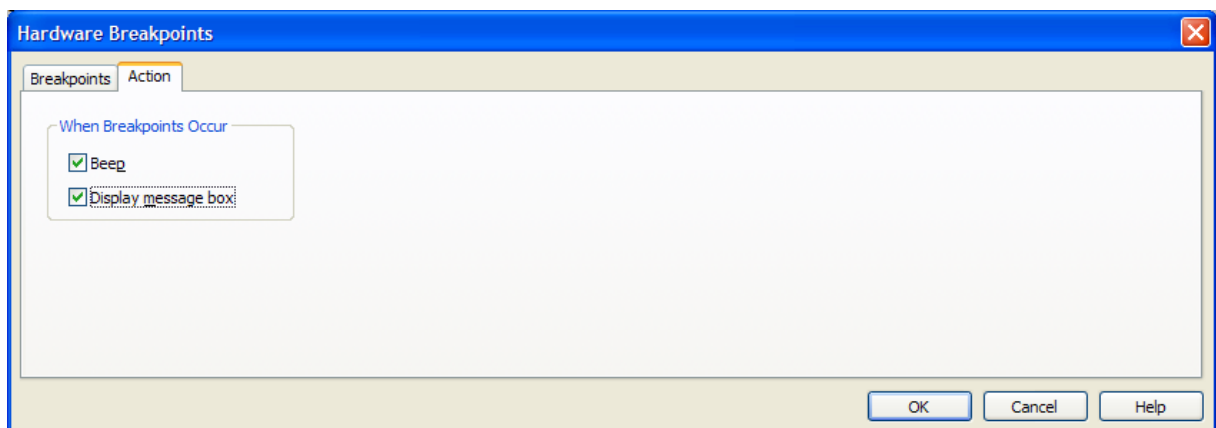


For every breakpoint an ASID - Address space identifier value can be set. When ASID is irrelevant, keep the check box unchecked.

Various logic conditions can be set for the address including address match and address mismatch. Address can be entered as a physical address or as a symbol, whose address is extracted from the debug information included in the download file.

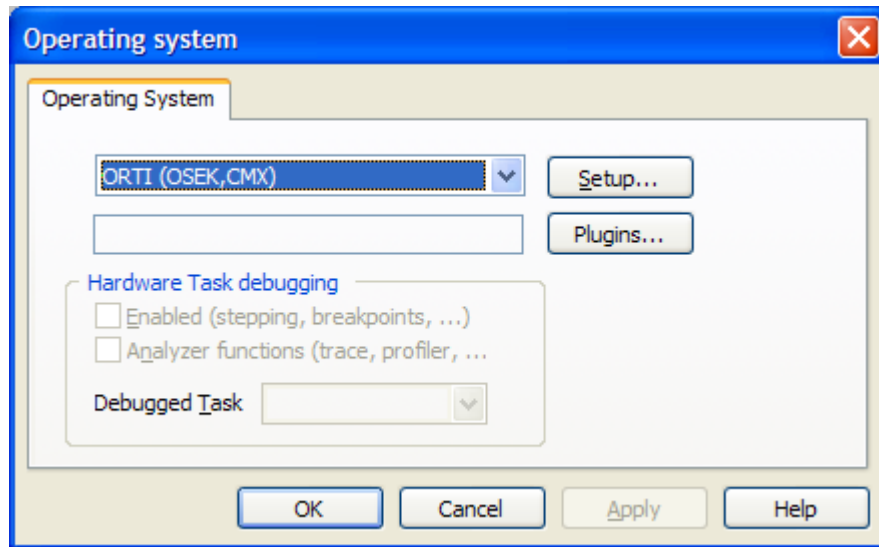
Data breakpoints also allow selecting memory access type, access size and optionally data value that can match or mismatch.

In the Action tab select the desired option to be performed when a breakpoint occurs.



8 OSEK Debug Support

Enable OSEK support by selecting 'ORTI (OSEK,CMX)' selection in the 'Debug/Operating System' dialog and then press the 'Setup' button.



Specify the path to the OSEK ORTI file, where the necessary debug information is kept.

9

Trace

The V850E2/Fx4 development system is based on a Renesas Fx4 Umbrella device that incorporates an On-Chip Trace Unit, the OCT. The on-chip trace is based on messages and has limitations compared to the in-circuit emulator where the complete CPU address, data and control bus is available to the emulator in order to implement exact and advanced trace features.

9.1 On-Chip Trace Concept

For program trace, trace port sends a message only for every executed non-sequential instruction, effectively on changes of a program flow. Each message contains the instruction type information and a destination program counter. Based on this information, the debugger reconstructs complete program flow by inserting sequential instructions between the recorded non-sequential instructions. This can work as long as the debugger has a complete code image of the application (download file) in order to know which sequential instructions are located between the non-sequential. For this reason, a self-modifying code cannot be traced.

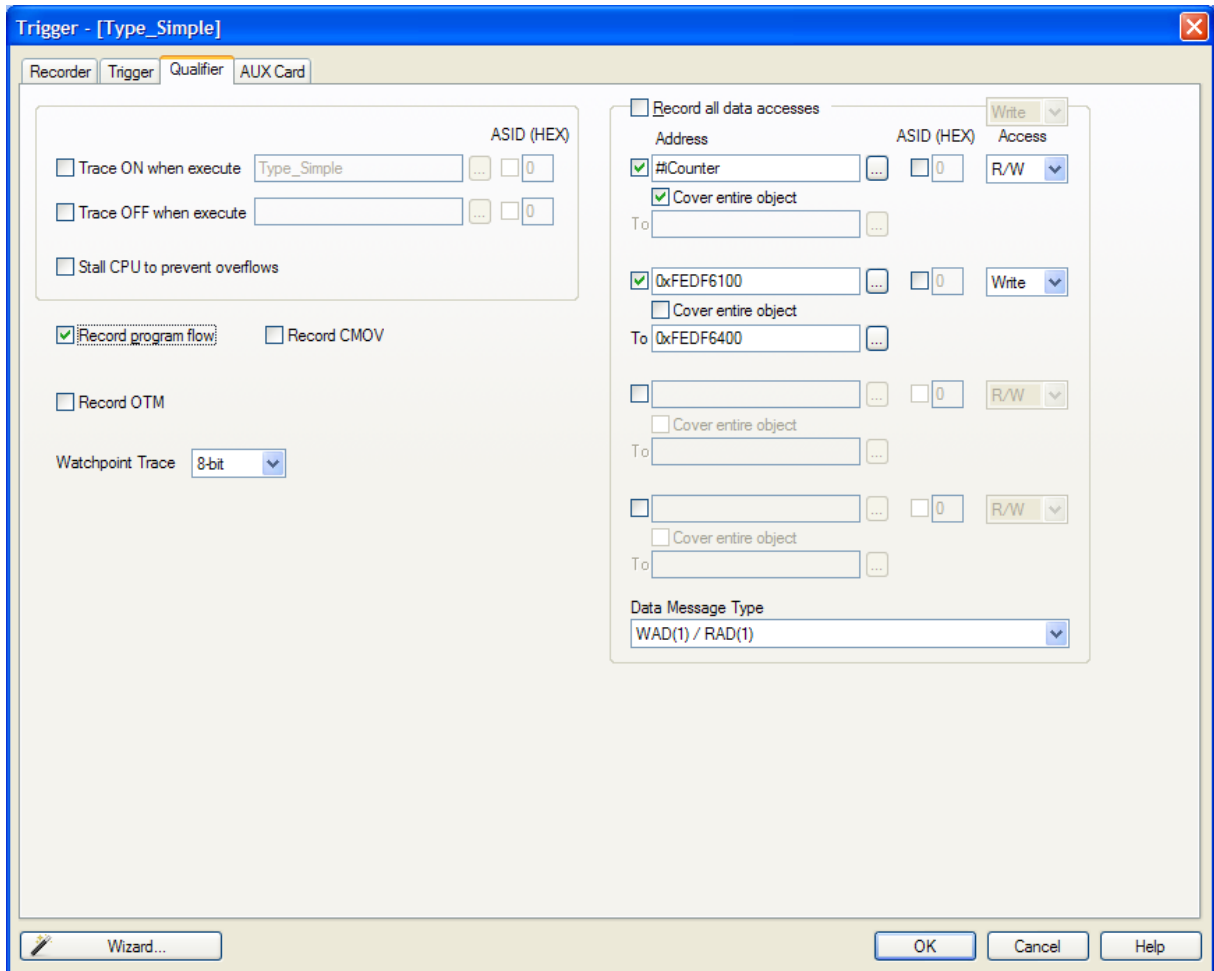
Transmitted OCT messages are appended with a time stamp information. That is a time of message, not of execution. All sequential instructions being reconstructed by the debugger, relying on the code image and inserted between the recorded addresses, do not contain exact time information. Any interpolation with the recorded addresses containing valid time stamp would be misleading for the user. Hence, all sequential instructions between the two non-sequential instructions, have the same time stamp value.

Data trace can record all data access cycles issued by the CPU. However, no access to the CPU core registers (R0-R31, FEPC, CTBP, etc) can be traced. Trace port bandwidth becomes quickly restrictive with the data trace enabled since data trace usually generates 2 messages for a single traced data access. When the number of trace messages exceeds the trace port bandwidth, an overflow message is sent out in order to inform the user that messages were lost. From that point on until the next synchronization message, there will be a gap in the trace display. It's up to the user then to either limit the traced data accesses, which yields less messages and thus no trace overflow, or to turn on the non-real-time trace mode, which stalls the CPU in order for the trace port to transmit all messages in the internal trace pipeline without loss. The non-real-time trace mode is turned on by checking the *Stall CPU to prevent overflows* option in the *Trigger/Qualifier* configuration dialog.

9.2 On-Chip Trigger and Qualifier Configuration

The *Qualifier* pane defines which CPU cycles are recorded by the trace. Complete program flow can be either recorded or not. Additionally, program flow recording can be limited to a specific part of an application code. Use *Trace ON* and *Trace OFF* option to do that. *Record CMOV* and *Record OTM* control the recording of Conditional Move instructions and Ownership Trace Messages.

Data accesses can be all recorded but this can easily lead to trace overflows. As an alternative, 4 data qualifier address ranges can be defined for the data accesses. See the next picture for an example.

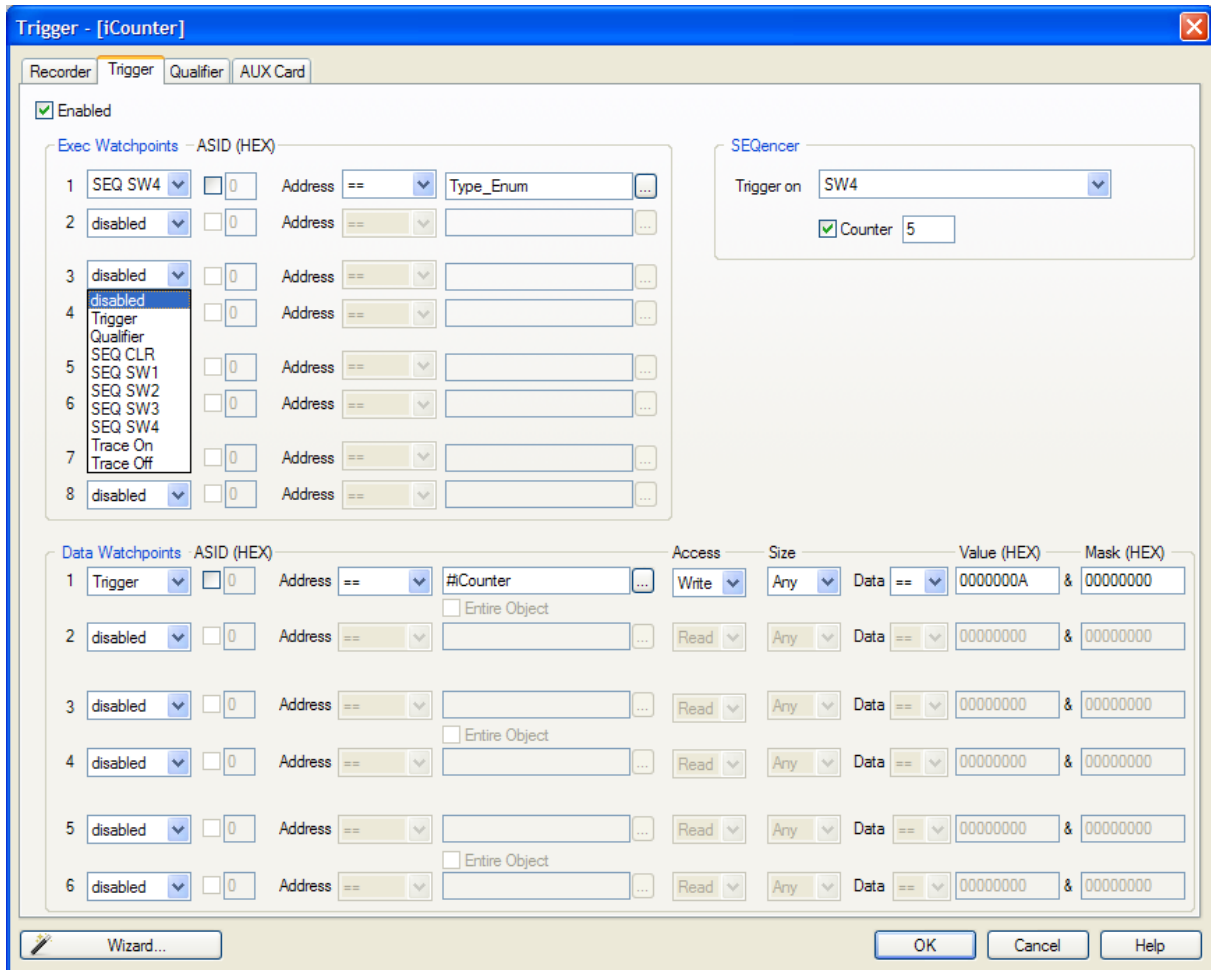


Trace Qualifier

The *Watchpoint Trace* width and *Data Message Type* are shown in their default, recommended settings. Other settings are available to aid solving any more demanding issues.

9.3 Trace Examples

A screenshot of the *Trigger* dialog shows that any trigger item can also function as a qualifier, as a Trace On or Off switch, or as an input to the Trigger Sequence Engine coupled with an event counter.

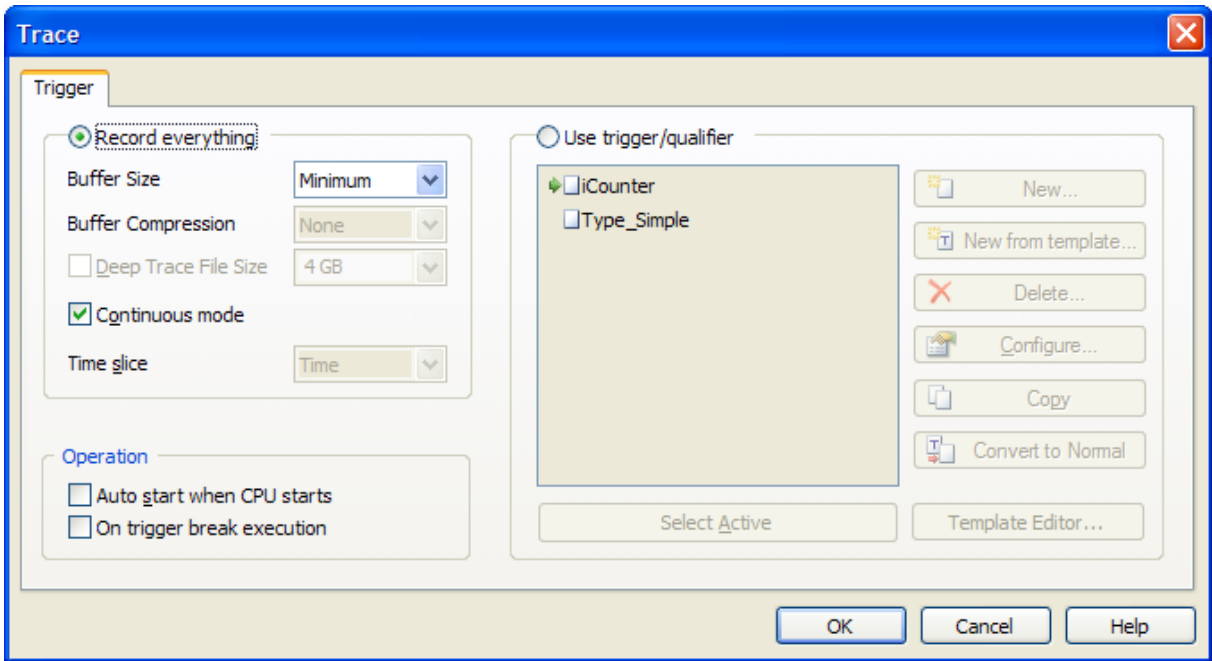


Trace Trigger

Example 1: Trace records the program execution (instructions, no data accesses) until the CPU is stopped.

First example explains how to record a complete program flow either from the application or trace start on or up to the moment, when the program stops. In first case, the trace would record and display program flow from the start until the trace buffer is full. Alternatively, the trace can stop recording on a program stop. The '*Continuous mode*' option 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 the *Debug/Stop* command issued by user.

Select the '*Record everything*' mode in the 'Trigger List' dialog. Set buffer size to minimum and check the '*Continuous mode*' option. Increase buffer size only when more program history is required since the trace upload time goes up with larger buffer size.



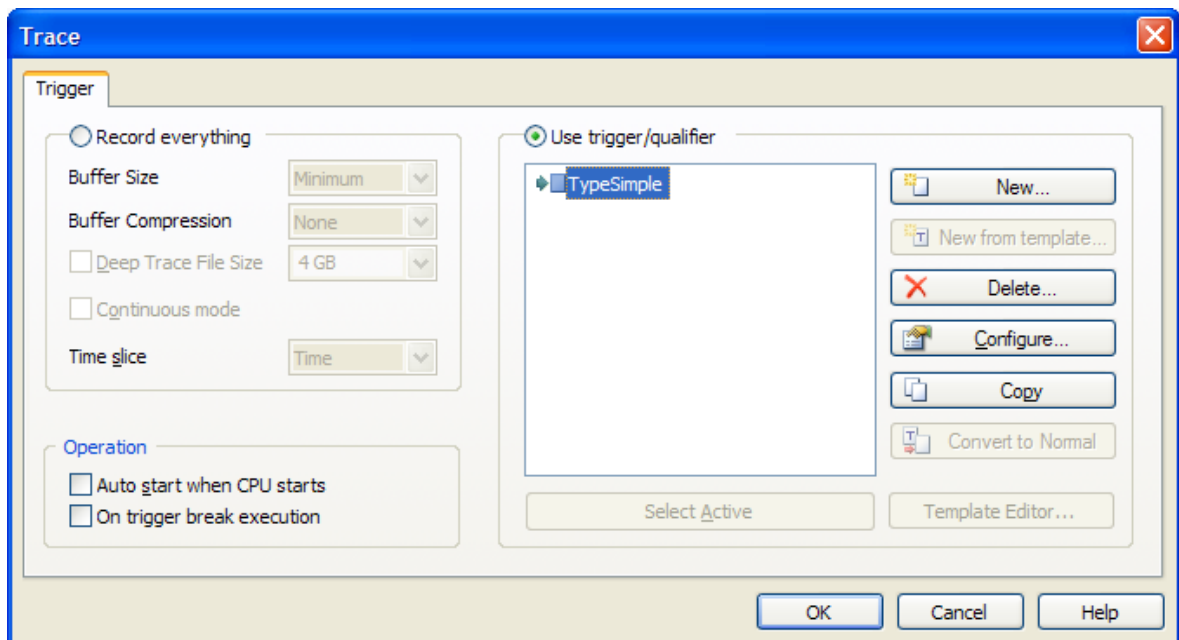
Before the program is set to run or while it is running already, activate the trace recording by the 'Trace begin' tool bar or the CTRL+B shortcut key. The trace stops recording when the program execution is stopped. After the trace stops recording, the collected information is analyzed and displayed.

Uncheck the '*Continuous mode*' option when the trace should record only until the trace buffer gets full.

'*Record everything*' mode always records program flow without data accesses. Use '*Trigger/qualifier*' mode when data accesses need to be recorded, as well.

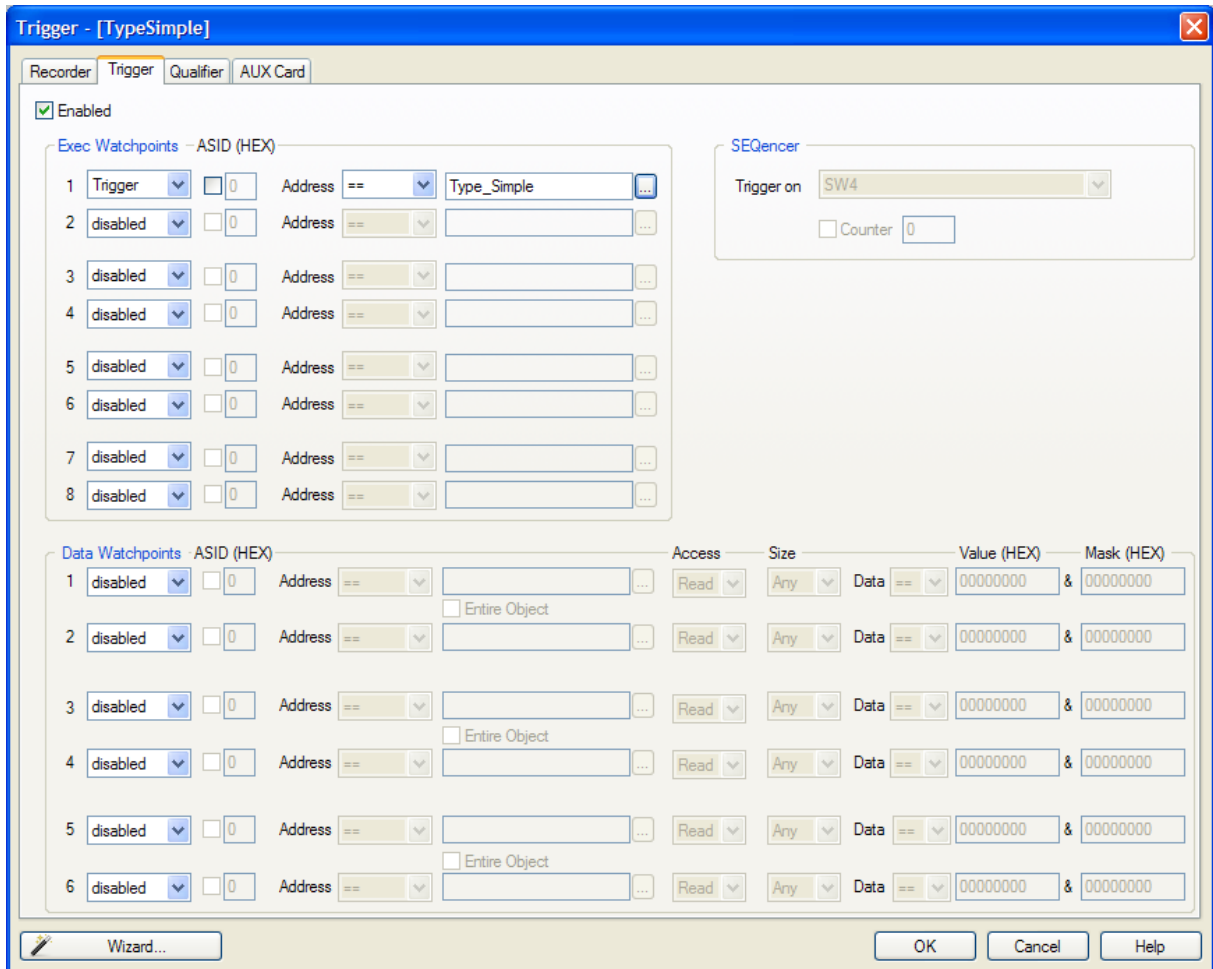
Example 2: Trace triggers when function *Type_Simple()* is called and program execution is recorded around the trigger event.

Select the '*Use trigger/qualifier*' mode in the 'Trigger List' dialog and configure a new trigger named simply: 'TypeSimple'.



Configure the trigger by invoking the 'Trigger' pane. First, select 'Enabled' in the Trigger combo box on top of the dialog, next set Execution Watchpoint 1 for the trigger condition and then enter Type_Simple address in the corresponding address field.

Leave the 'Qualifier' pane at default setting, which records program flow only.



Example 2 Trigger Settings

Now activate the trace and run the program. When the trigger event occurs, the trace will stop recording and display the recorded program. Let's take a look at the trace record as there are some differences in the display comparing to the standard in-circuit emulator trace, where the program flow is obtained by recording activities directly on the CPU bus.

| Number | Address | Data | Content | Time |
|--------------|----------|----------|--|----------|
| -6.3 | 000008D8 | 0200021E | } CPU_TestAsm_EXIT_ 1E02 CALLT #1E Instruction | -401 ns |
| -4.0 | 00000AC4 | 003F0640 | __Epoplp0 40063F00 DISPOSE 0,R31,[R31] Instruction | -201 ns |
| -2.0 | 000000C0 | 021EE595 | } 95E5 BR 00000082 Instruction | 0 ns |
| T 0.0 | 00000000 | 00000010 | Watchpoint | 0 ns |
| 1.0 | 00000082 | 57645201 | iCounter = 1; 0152 MOV 01,R10 Instruction | 94 ns |
| 1.1 | 00000084 | 80055764 | 64570580 ST.W R10,-7FFC[R4] Instruction | 144 ns |
| 1.2 | 00000088 | 0040FF80 | Type_Simple(); 80FF4000 JARL Type_Simple(000000C8),R31 Instruction | 244 ns |
| P 3.0 | 000000C8 | E80025E5 | void Type_Simple() Type_Simple E525 BR 00000114 Instruction | 944 ns |
| 5.0 | 00000114 | F1E10780 | 8007E1F1 PREPARE R23,R24,R25,R26,R27,R28,R29 Instruction | 1.177 us |
| 5.1 | 00000118 | 0000DD95 | 95DD BR 000000CA Instruction | 1.644 us |
| 7.0 | 000000CA | E000E800 | char c=0; 00E8 MOV R0,R29 Instruction | 1.672 us |

Example 2 Trace Result

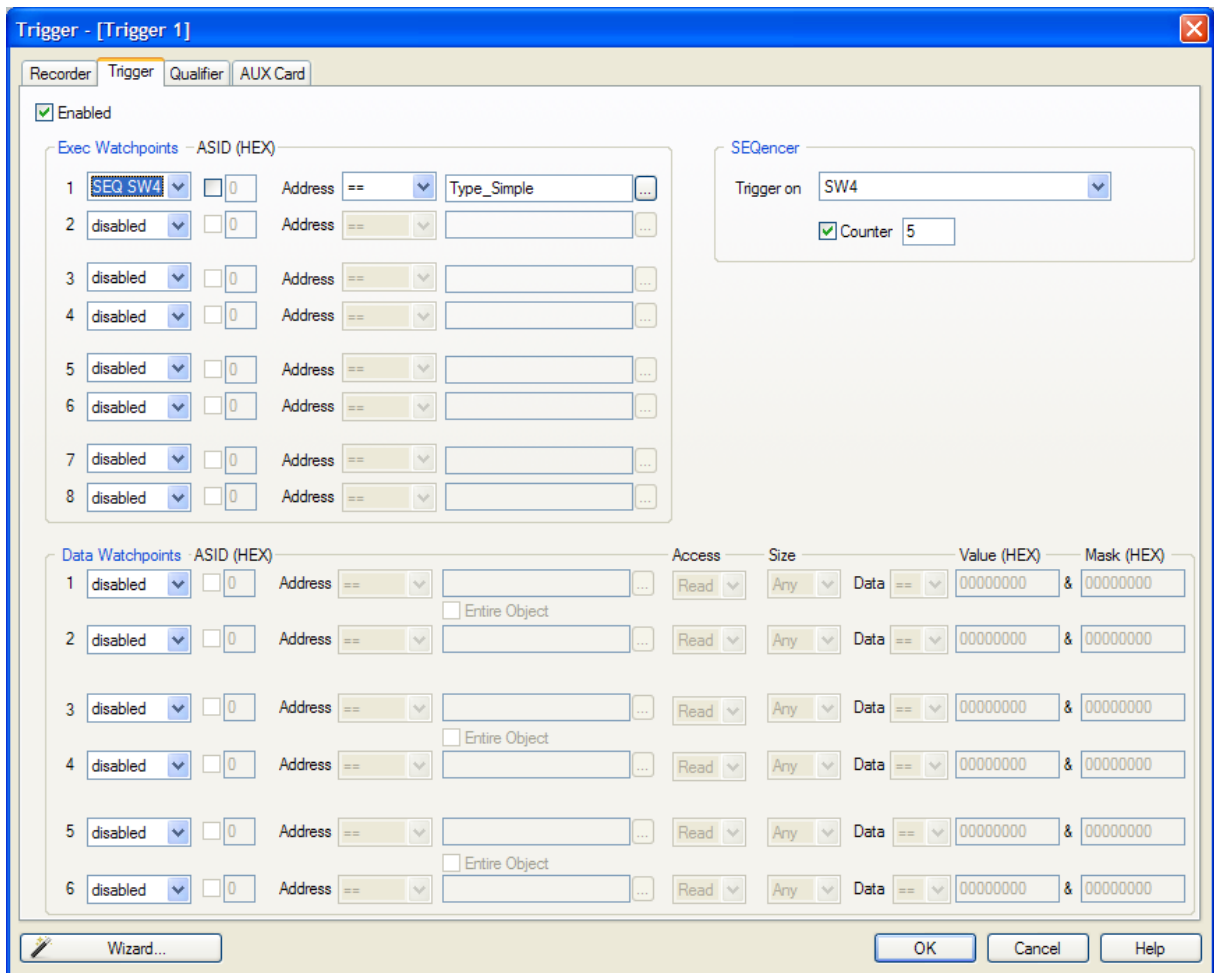
When the trigger occurs, trace port outputs a special trigger message called Watchpoint, frame 0 above. This is detected by the external trace hardware and recorded as a trigger event. Recorded trigger message is depicted with a red frame in the trace window.

In this example, we can see that actual Type_Simple() function started 3 frames after the Watchpoint message. This is due to a delay between the time when the instruction was executed and the time when belonging message is broadcast externally. It is user's concern to search for the code that actually generated the trigger message. The distance from the trigger frame to the actual trigger point in the reconstructed program also depends on the number of sequential instructions following the trigger event.

Example 3: The trace will start recording (instructions and data accesses) after the function Type_Simple() is executed for the fifth time.

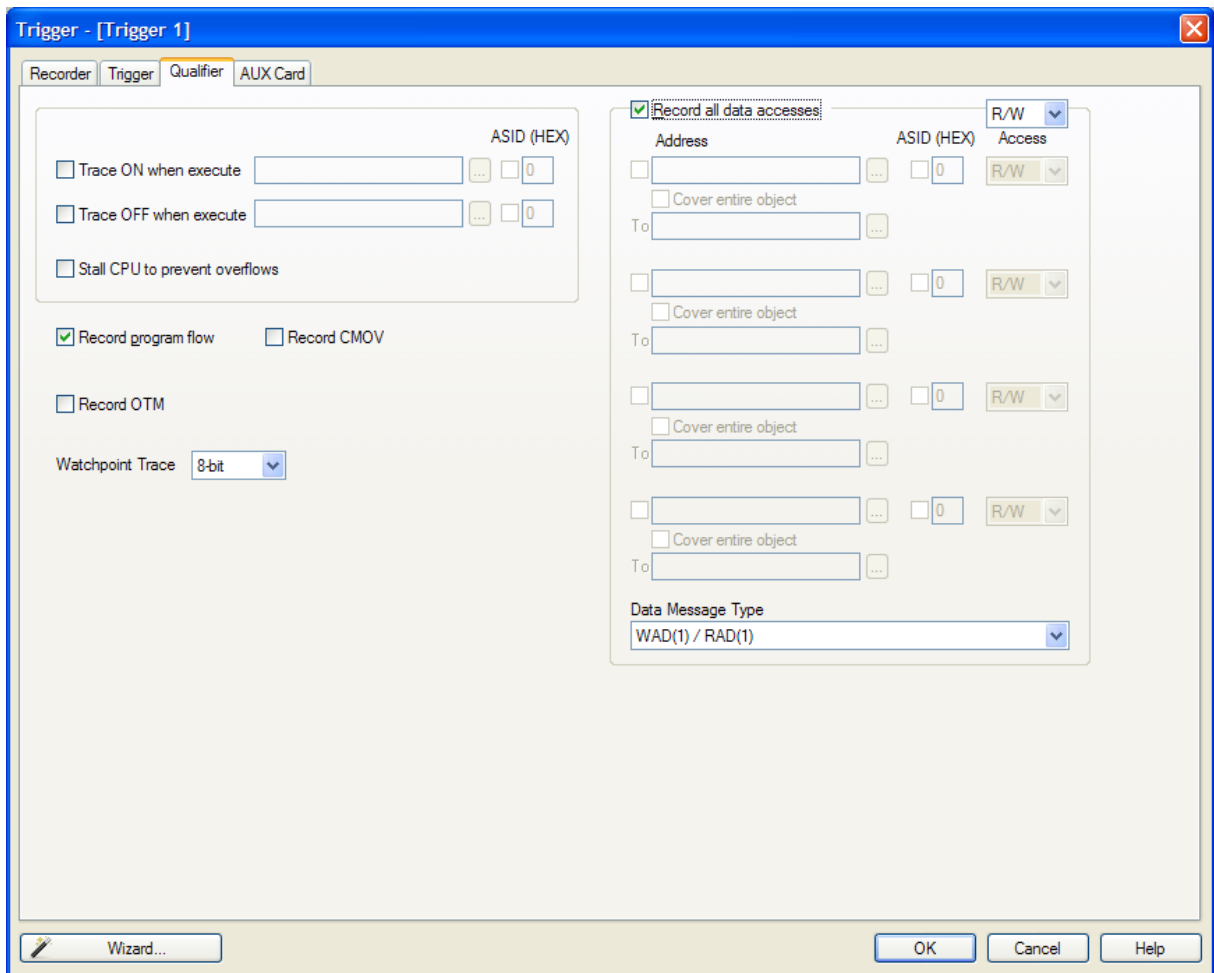
Select the 'Use *trigger/qualifier*' mode in the 'Trigger List' dialog and configure a new trigger called e.g. 'Trigger 1'.

Reuse the settings from the previous example. All remains the same except that Execution Watchpoint 1 is now configured for the sequencer respectively SW4 state of the sequencer instead for the trigger. The user can configure a sequence of up to 4 states (SW1-SW4) in the '*Sequencer*' field. Additionally, enable the counter, which connects it to the output of the sequencer and set its value to 5.



Example 3 Trigger Settings

Next, open the Qualifier pane in order to enable also a data access recording.



Example 3 Qualifier Settings

Start the trace and run the application from reset on. After *Type_Simple()* is executed five times, the trace will trigger and normally display the program around the trigger message. Let's analyze the trace record. It's possible that the trace record will contain the overflow messages. That happens due to the enabled data trace. Occurrence of overflows depends on the code, which can generate an arbitrary number of data accesses. If the CPU accesses the data very often in a short time, it can produce such amount of messages that the CPU internal Nexus trace FIFO buffer can no longer handle. If that happens, an overflow message is sent out signaling the external trace hardware that the trace information is lost and no longer valid. From that moment on the debugger can no longer reconstruct the program flow properly until the trace FIFO buffer is flushed and a synchronization message is broadcast. There are two solutions for this problem presented below.

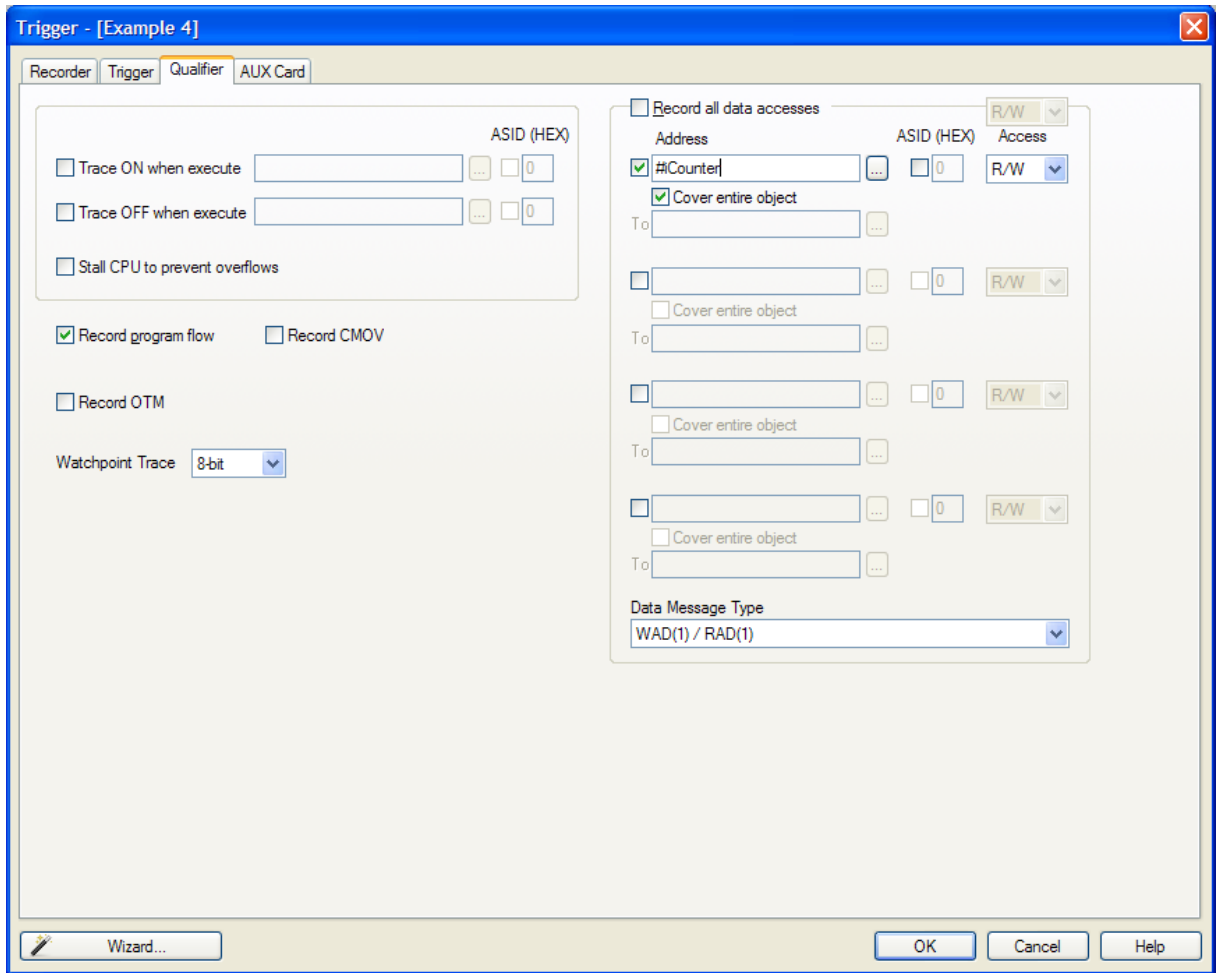
| Number | Address | Data | Content | Time |
|--------|----------|----------|--|------------|
| -13.0 | 000008F8 | 00000140 | __PROLOG_TABLE Read | -10.401 us |
| -11.0 | 000008C6 | FD950200 | 0002 CALLT #00 Instruction | -7.201 us |
| -9.0 | FEDF6384 | 000000B8 | Write | -7.201 us |
| -7.0 | 00000000 | 00000007 | Error | -5.601 us |
| -6.0 | 00000A38 | 00210780 | __Epushlp0 80072100 PREPARE R31,0 Instruction | -3.201 us |
| -4.0 | 000000C0 | 021EE595 | } 95E5 BR 00000082 Instruction | -1.601 us |
| -2.0 | FEDF6004 | 00000001 | iCounter Write | -1.601 us |
| 0.0 | 00000000 | 00000010 | Watchpoint | 0 ns |
| 1.0 | 00000082 | 57645201 | iCounter = 1; 0152 MOV 01,R10 Instruction | 1.200 us |
| 1.1 | 00000084 | 80055764 | 64570580 ST.W R10,-7FFC[R4] Instruction | 1.600 us |
| 1.2 | 00000088 | 0040FF80 | Type_Simple(); 80FF4000 JARL Type_Simple(000000C8),R31 Instruction | 2.400 us |
| 3.0 | 000000C8 | E80025E5 | void Type_Simple() Type_Simple E525 BR 00000114 Instruction | 4.000 us |
| 5.0 | FEDF6384 | E00AAF78 | Write | 4.000 us |
| 7.0 | FEDF6380 | D46790B4 | Write | 5.600 us |
| 9.0 | FEDF637C | 34743486 | Write | 7.200 us |
| 11.0 | FEDF6378 | 58416A84 | Write | 8.800 us |

Example 3 Trace Result

The first one is to use the non-real-time trace mode, which stalls the CPU until the FIFO buffer becomes ready for new messages and then resumes the execution. This trace mode is turned on by checking the ‘*Stall CPU to avoid overflows*’ option in the Trigger pane. Since this is a severe intrusion in the real-time program execution it is not a recommended solution unless the user is fully aware of the consequences.

The second and better solution is to limit the amount of the generated data access messages. The existing trace configuration records all data accesses although the user might be interested only in data accesses to a specific, limited region or even just to a few variables.

The qualifier pane setting in the Example 4 below configures the trace to record only data accesses to the 32-bit global variable named iCounter.



Example 4 Qualifier Settings

This setting yields a trace recording without overflow errors. See the next screenshot.

| Number | Address | Data | Content | Time |
|--------------|----------|----------|---|-----------|
| -8.3 | 000008D8 | 0200021E | } CPU_TestAsm_EXIT_ 1E02 CALLT #1E Instruction | -4.801 us |
| -6.0 | 00000AC4 | 003F0640 | __Epoplp0 40063F00 DISPOSE 0,R31,[R31] Instruction | -3.201 us |
| -4.0 | 000000C0 | 021EE595 | } 95E5 BR 00000082 Instruction | -1.601 us |
| -2.0 | FEDF6004 | 00000001 | iCounter Write | -1.601 us |
| T 0.0 | 00000000 | 00000010 | Watchpoint | 0 ns |
| 1.0 | 00000082 | 57645201 | iCounter = 1; 0152 MOV 01,R10 Instruction | 1.200 us |
| 1.1 | 00000084 | 80055764 | 64570580 ST.W R10,-7FFC[R4] Instruction | 1.600 us |
| 1.2 | 00000088 | 0040FF80 | Type_Simple(); 80FF4000 JARL Type_Simple (000000C8),R31 Instruction | 2.400 us |
| P 3.0 | 000000C8 | E80025E5 | void Type_Simple() Type_Simple E525 BR 00000114 Instruction | 4.000 us |
| 5.0 | 00000114 | F1E10780 | 8007E1F1 PREPARE R23,R24,R25,R26,R27,R28,R29 Instruction | 4.533 us |
| 5.1 | 00000118 | 0000DD95 | 95DD BR 000000CA Instruction | 5.600 us |

Example 4 Trace Result

10 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 uncover functions that are most time consuming or time critical and need to be optimized.

The profiler 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 instrumentation 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 times
- 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 profiler is entirely based on the trace recording. It first uses trace to record a complete program flow and then in off-line, functions' entry and exit points are extracted by means of software, the statistics is run over the collected information and finally the results are displayed. Note that total session time depends on the application and the amount of profiled objects.

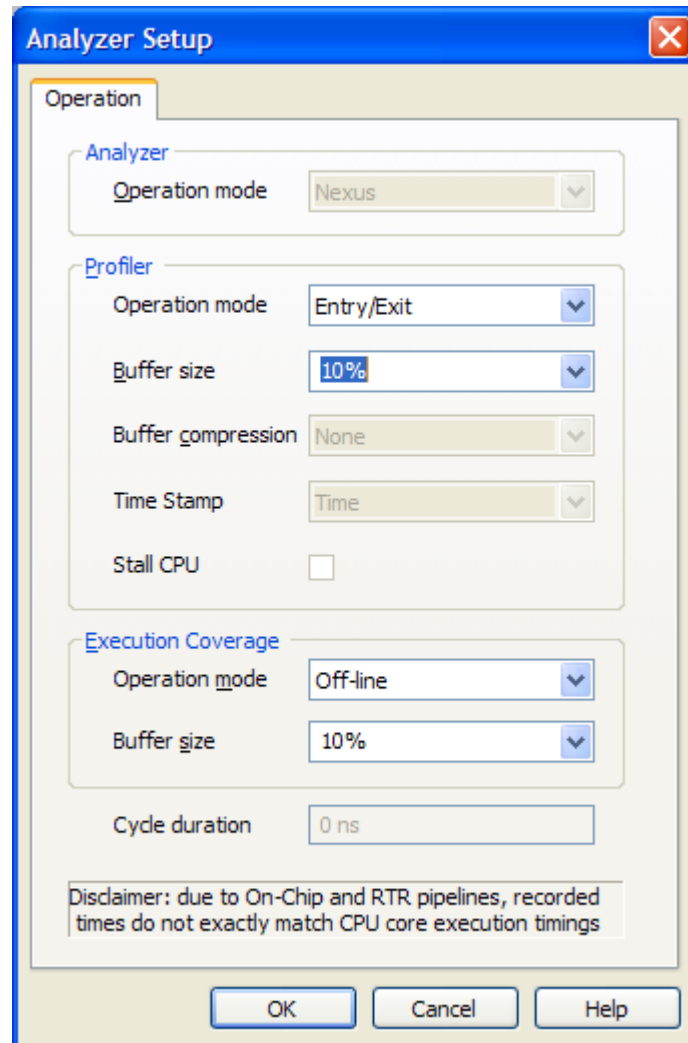
Refer to a separate document titled Profiler User's Guide for more details on profiler and its use.

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

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 the profiler, select the working profiler buffer size in the '*Hardware/Analyzer Setup*' dialog.



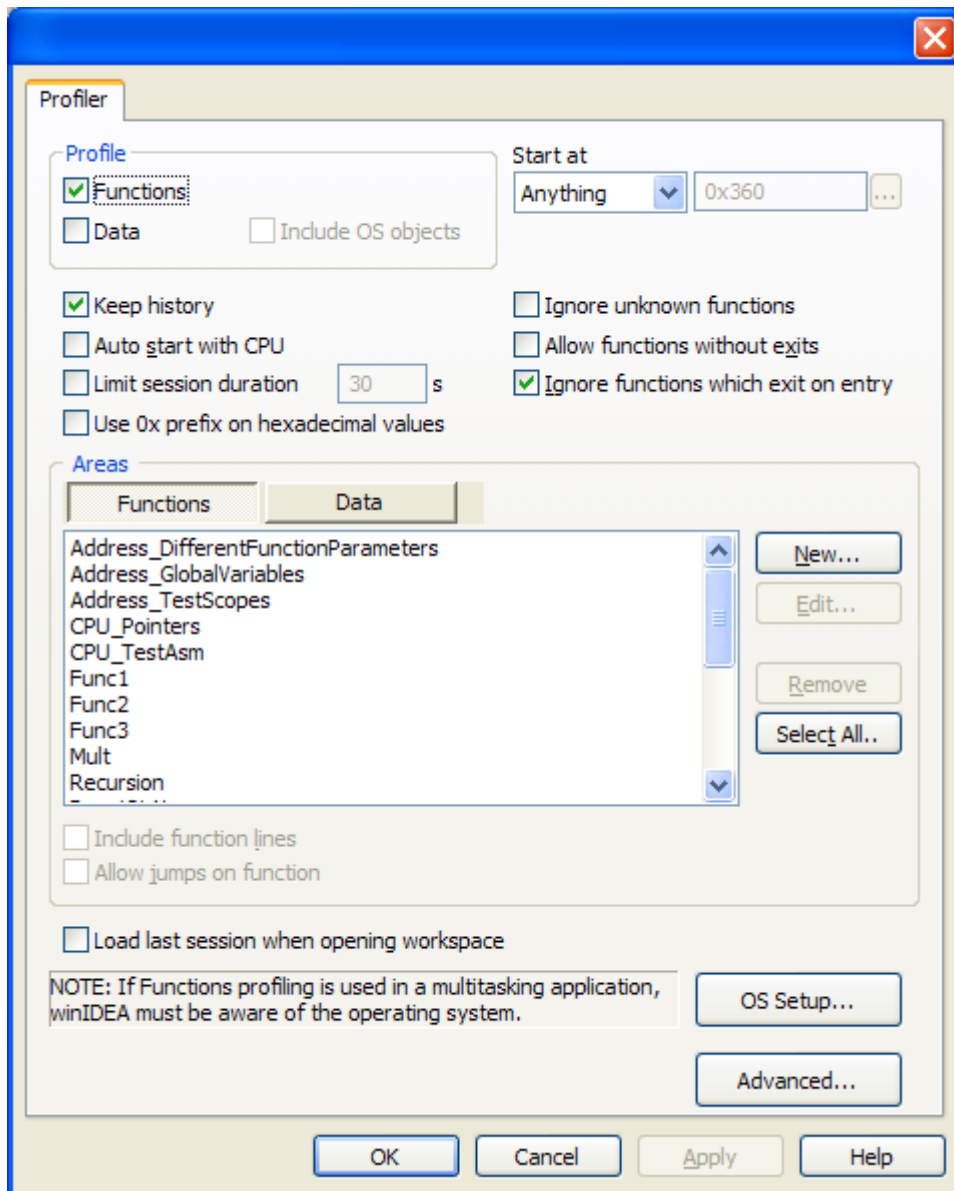
Next, select 'Profiler' window from the View menu and configure profiler settings (see next figure). Select 'Functions' option in the 'Profile' field when profiling functions. In order to profile a data variable, 'Data' should be checked instead. For instance, Data Profiler can be used as a Task Profiler, 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* MUST be profiled too! Data profiler is 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.

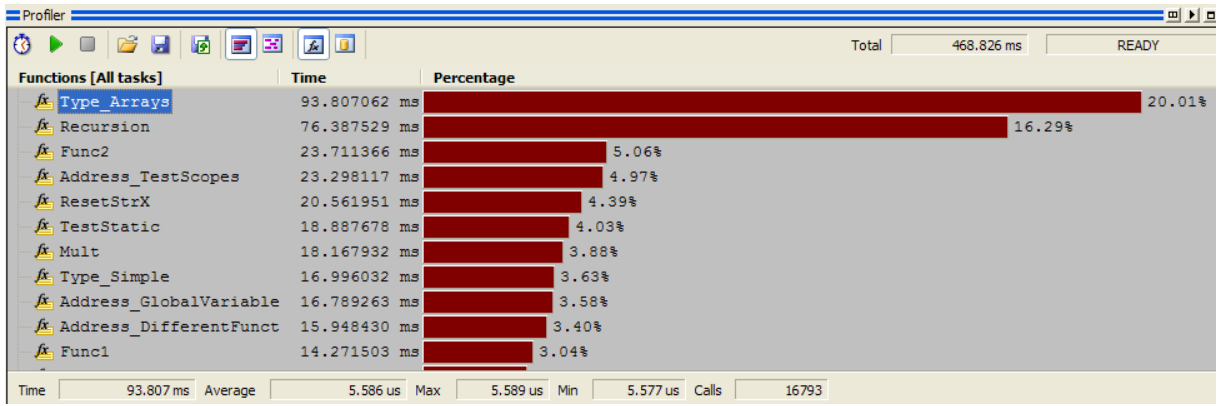
Finally, profiled functions are selected by pressing 'New...' button. It's recommended that 'All Functions' option is selected in the beginning.

The debugger extracts all the necessary information from the debug info, which is included in the download file and configures the emulator 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

11

Execution Coverage

Execution coverage tool records all addresses being executed, which allows the user to detect the code or memory areas not executed or not covered. 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 and can be fatal in mission critical software as it is never tested.

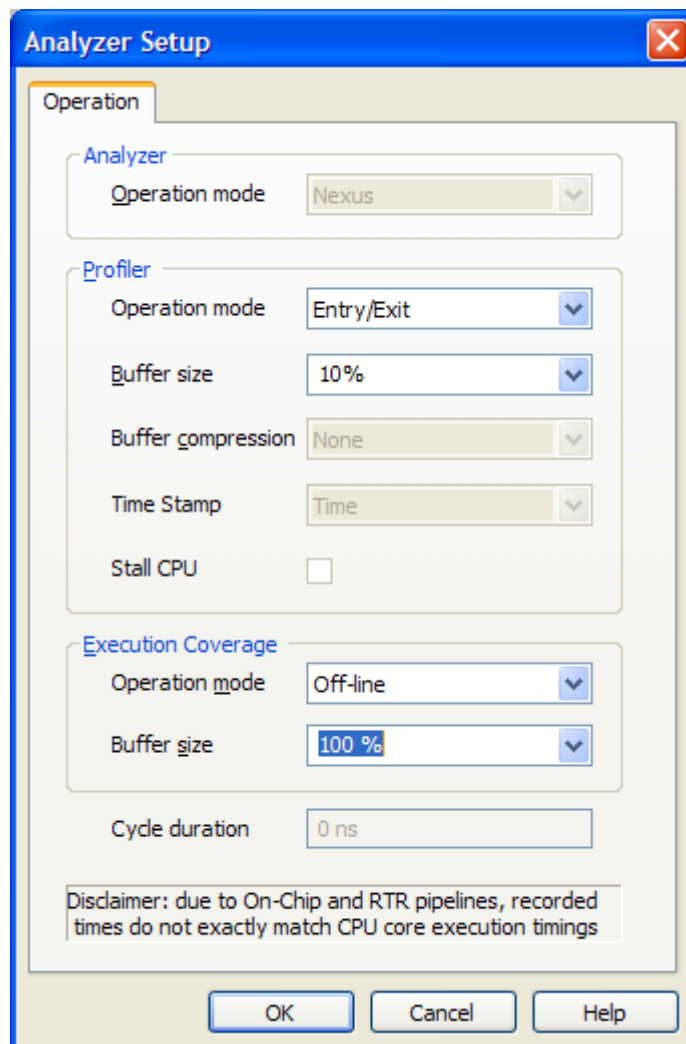
The execution coverage is entirely based on the trace recording. It first uses trace to record the executed code (capture time is limited by the trace depth) and then in off-line application executed instructions and source lines are extracted by means of software analysis tool and displayed in the Coverage window.

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

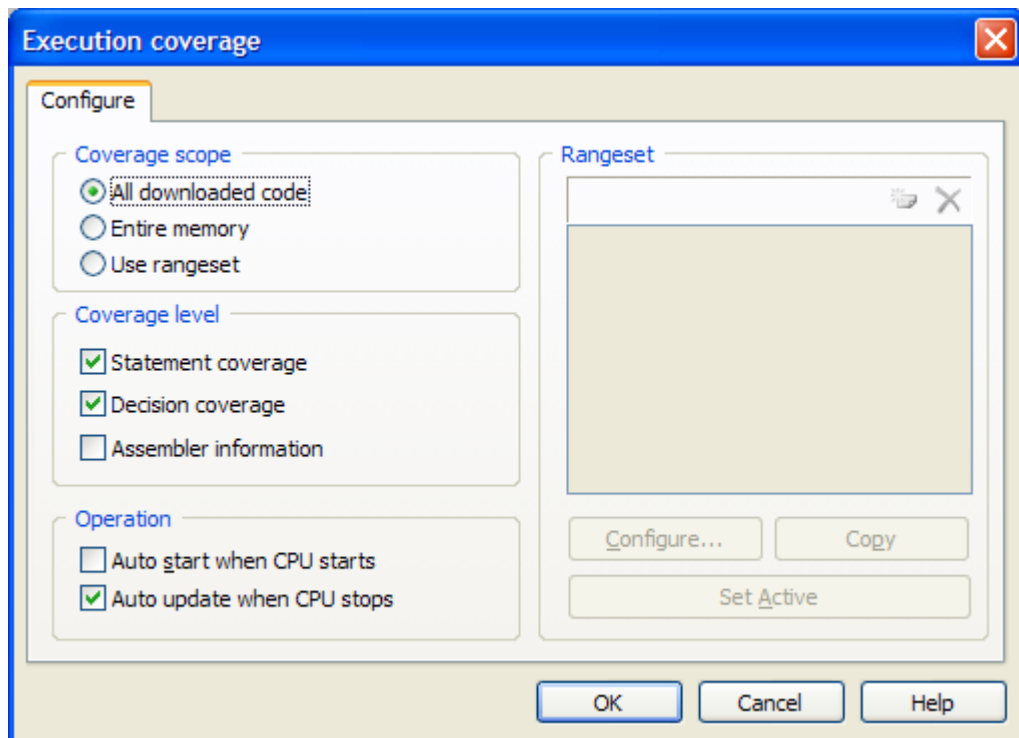
Note: Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace.

Typical Use

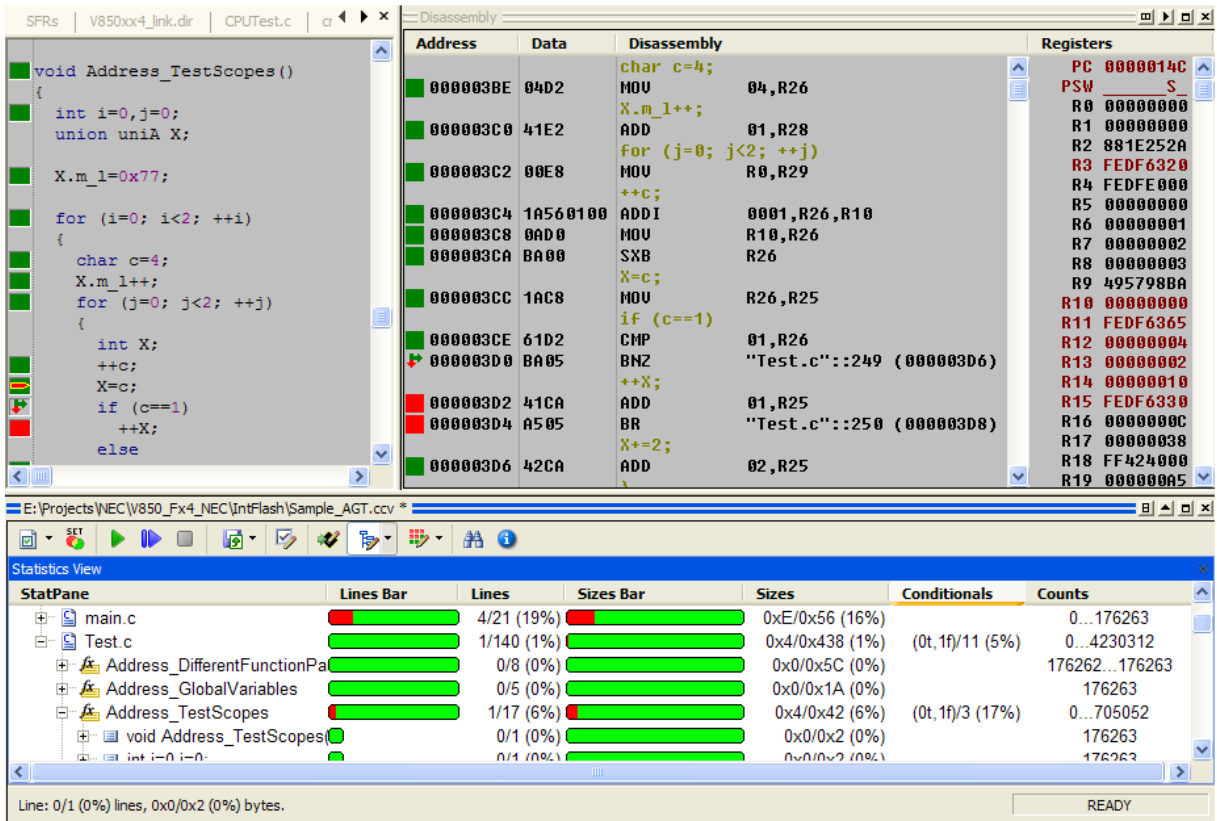
Select the execution coverage buffer size in the ‘*Hardware/Analyzer Setup*’ dialog.



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 configures the BlueBox 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

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.