
Technical Notes

Renesas R8C In-Circuit Emulation

Contents

Contents.....	1
1 Introduction	2
2 Emulation Options.....	3
2.1 Hardware Options	3
2.2 CPU Configuration	4
2.3 Initialization Sequence	5
3 Setting CPU options	7
3.1 CPU Options	7
3.2 Advanced Options	9
4 Memory Access	10
5 Access Breakpoints	11
6 Trace.....	13
7 Execution Coverage.....	14
8 Profiler.....	15

1 Introduction

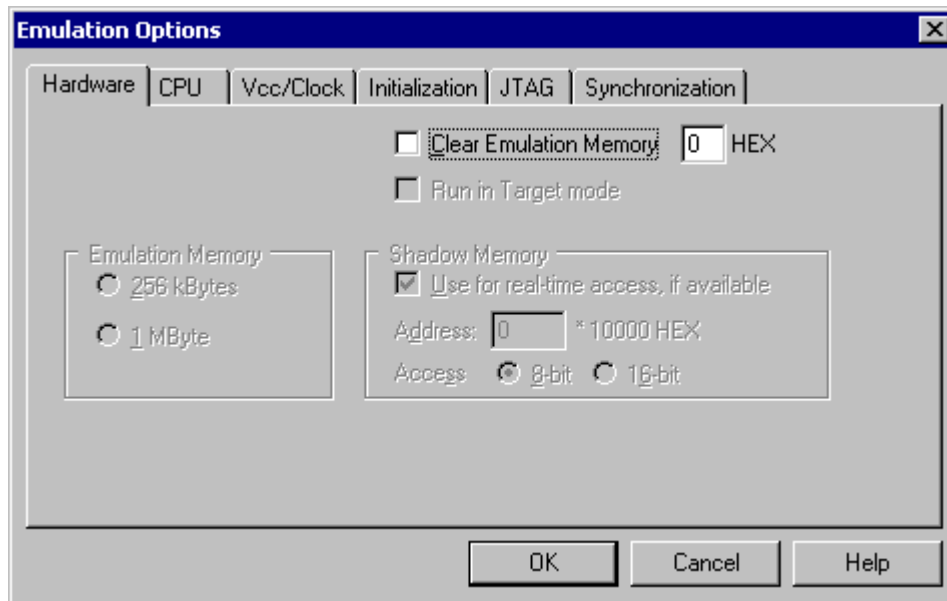
The R8C/3x Active PRO POD is an in-circuit emulator based on the Renesas special emulator (EVA) chip. All the emulation memory is internal to the EVA chip. EVA chip has 256KB internal RAM for flash emulation, 4KB internal RAM for data flash emulation and 10KB internal RAM for RAM emulation. EVA chip uses special serial interface (MSCI) for the real-time memory access to all the EVA memory.

Debug Features

- Unlimited execution breakpoints
- Access breakpoints
- Real-time access
- Trace
- Profiler
- Execution coverage
- Upload while sampling (Trace, Profiler)

2 Emulation Options

2.1 Hardware Options



In-Circuit Emulator Options dialog, Hardware page

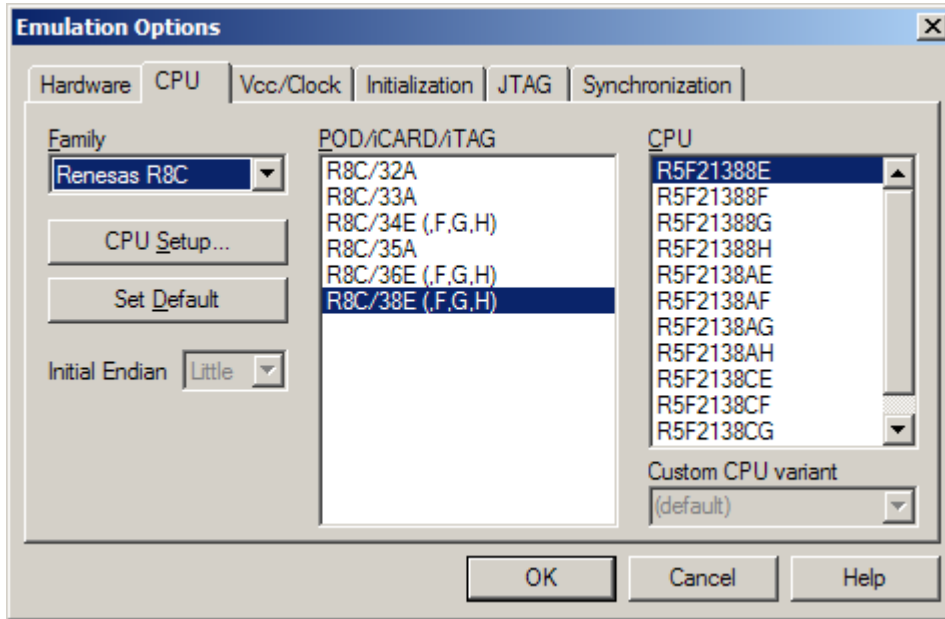
Clear Emulation Memory

This option allows you to force clearing (with the specified value) of emulation memory after the emulation unit is initialized.

Clearing whole emulation memory takes some time, so use it only when you want to make sure that previous emulation memory contents don't affect the current debug session.

2.2 CPU Configuration

With In-Circuit emulation besides the CPU family and CPU type the emulation POD must be specified



In-Circuit Emulator Options dialog, CPU Configuration page

CPU Setup

Opens the CPU Setup dialog. In this dialog, general and CPU specific parameters are configured. The dialog will look different for each CPU reflecting the options available for it.

Set Default

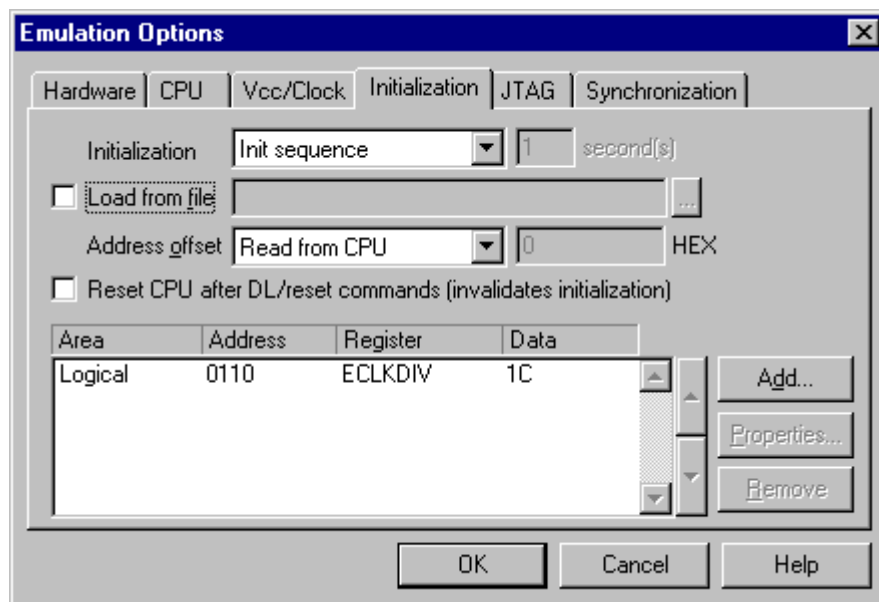
- This button will set default options for currently selected CPU. Default options are also set when the Family or a POD is changed.

2.3 Initialization Sequence

There is normally no need to use initialization sequence when debugging with an In-Circuit Emulator. 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. Normally there is no need at all to use the initialization sequence in case of the In-Circuit Emulator emulating Single Chip mode. Initialization sequence is required only for some CPU families when it is required by the application being debugged. That can be e.g. either to enable memory access to the CPU internal EEPROM memory or to some external target memory, which is not accessible after the CPU reset. In such case, the debugger executes initialization immediately after reset and then downloads the code. Additionally, the user can also disable CPU internal COP using initialization sequence if there is a need for that, etc.

The initialization sequence can be set up in two ways:

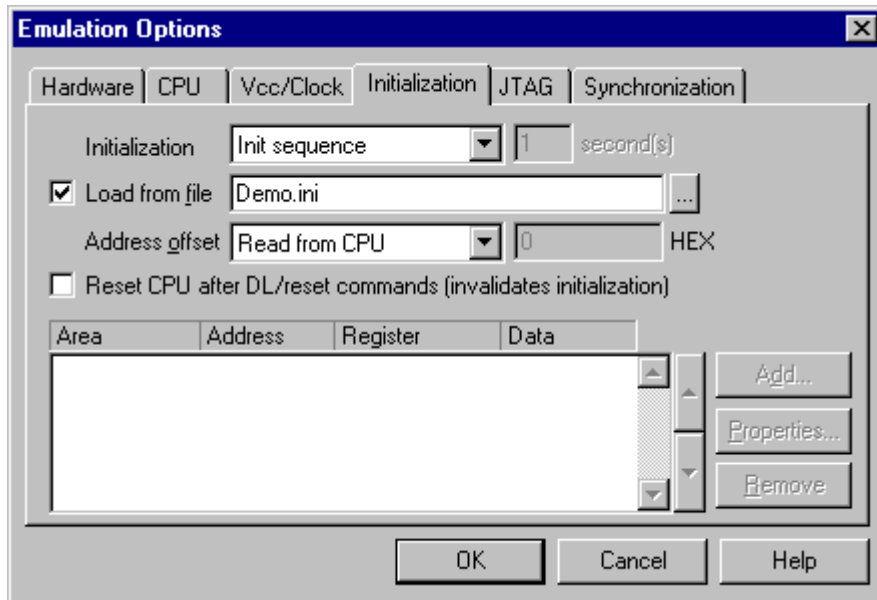
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



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

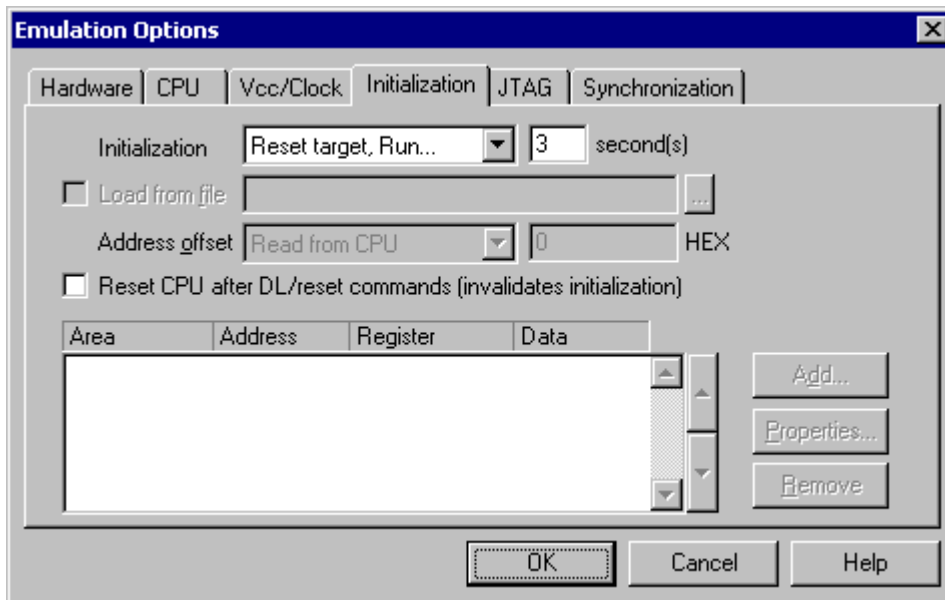
Excerpt from the sample SEQUENCE1.ini file:

```
S PTBD B 12          //comment
S PTBDD B FF
```



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

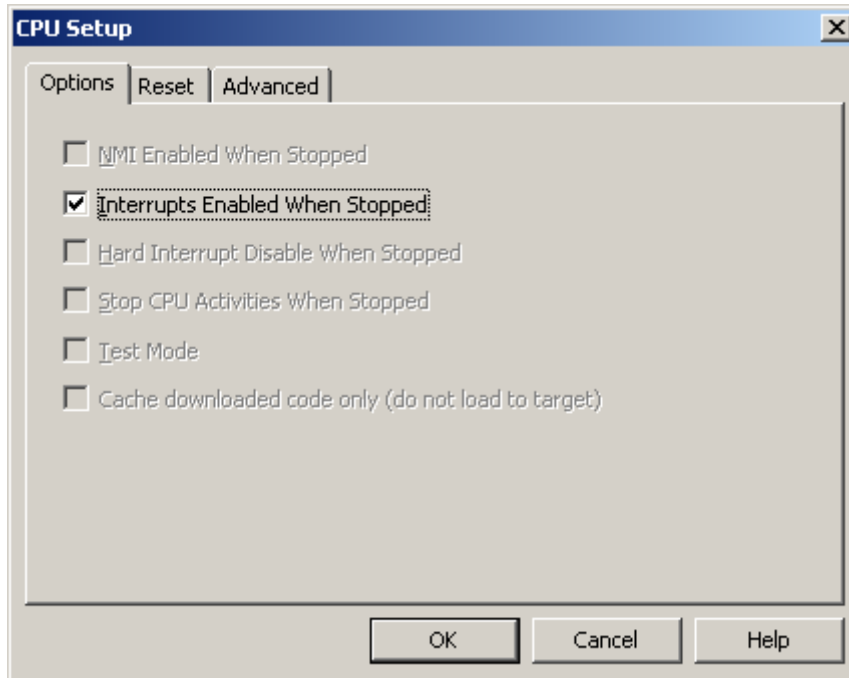
There is also a third method, which can be used too but it's not highly recommended for the start up. The user can initialize the CPU by executing part of the code in the target ROM for X seconds by using 'Reset and run for X sec' option.



3 Setting CPU options

3.1 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

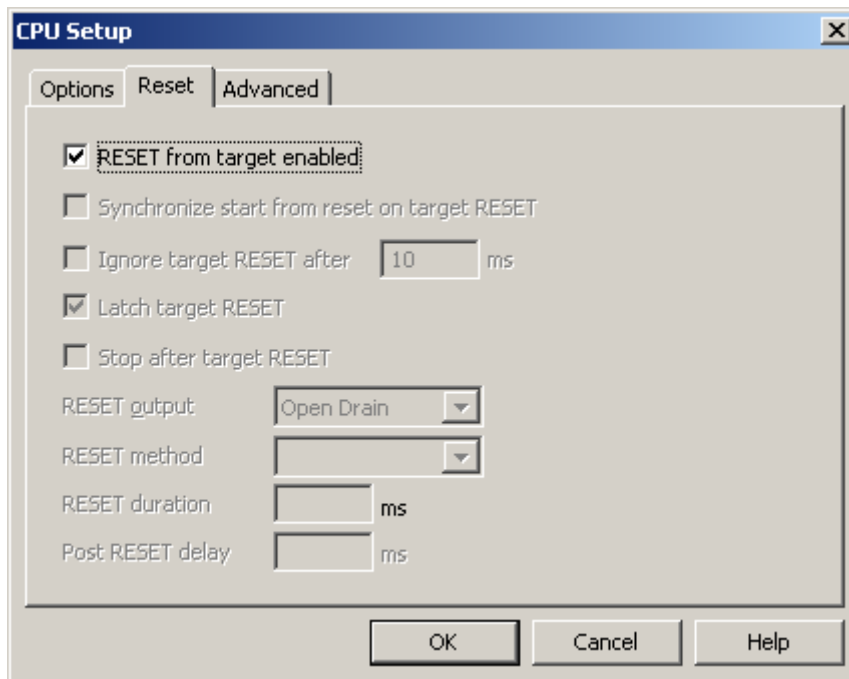
Interrupts Enabled When Stopped

The development system itself doesn't support servicing interrupts while the application is stopped (interrupts in background). This option impacts only on the CPU behaviour during the instruction and source step.

Disabling this option makes the Emulator mask the interrupts between a debug step command, which normally results in more predictive behaviour of applications using interrupts. This is a default setting.

If the option is checked, the Emulator doesn't mask interrupts and they can occur while stepping through the application. If there is a periodic interrupt, it may happen that the user will keep re-entering the interrupt while stepping. In such applications, it's recommended to disable this option.

3.2 Reset Options

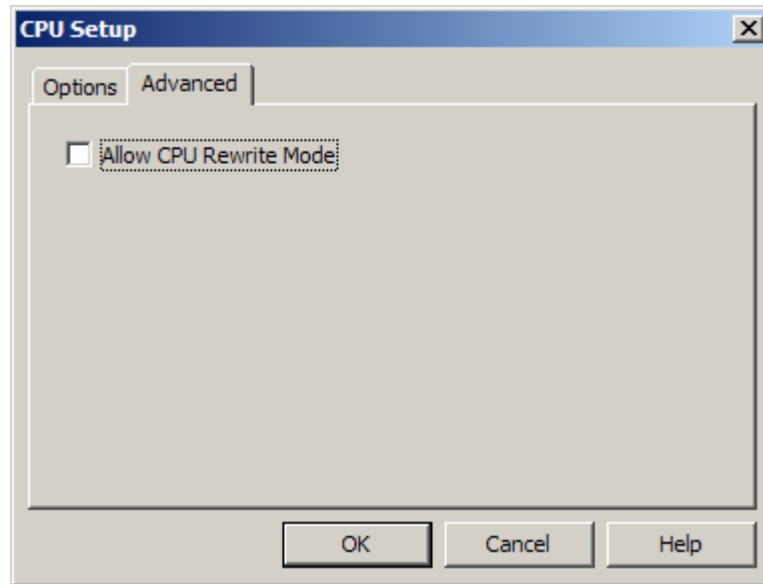


CPU Setup, Options page

RESET from Target Enabled

When checked, the target reset line is sensed, which can then reset the CPU while the CPU is running.

3.3 Advanced Options



R8C Advanced page

The R8C in-circuit emulator is based on Renesas EVA chip. When this option is checked, the application can modify EVA memory using flash programming functions.

Notes on debugging in CPU Rewrite Mode (extract from Renesas documentation):

- If the option is checked you cannot use setting software breakpoints in an internal ROM area.
- In CPU rewrite mode and erase suspend mode, do not stop the program. And do not single step an instruction shifting to CPU rewrite mode or erase suspend mode. The emulator will be uncontrollable in CPU rewrite mode and erase suspend mode.
- While executing the program that rewrites the CPU, do not perform the following operations; new setting or change of the settings of the RAM monitor area, or erase of any data. Otherwise, the CPU rewrite operation may not be properly executed.
- To reference data after executing CPU rewrite, stop the program at other than a rewrite control program area and use the Memory window etc.
- When erasing blocks or programming using DTC in CPU rewrite mode, do not refer to the memory of the blocks in the internal ROM area (program ROM and data flash) to be erased or written from the following windows.

If you refer to the memory contents of the internal ROM area to be erased or written during program execution, CPU rewrite may not be properly executed.

Although the program is stopped while DTC is activated, CPU rewrite by DTC is continued. Therefore, make sure to operate windows after CPU rewrite is complete.

Due to this last restriction, it is recommended to disable any real-time access in winIDEA, which would read internal ROM memory while the application is running.

Memory Access

The development tool features standard monitor memory access, which require user program to be stopped and real-time memory access based on shadow memory, which allows reading the memory while the application is running.

Real-Time Memory Access

The development system supports real-time memory read and write access by default. When emulator is using RT access, DMA cycles are generated which can slightly slow down user's code execution.

Watch window's Rt.Watch panes can be configured to inspect memory while the application is running. Optionally, memory and SFR windows can be configured to use real-time access as well.

Monitor Access

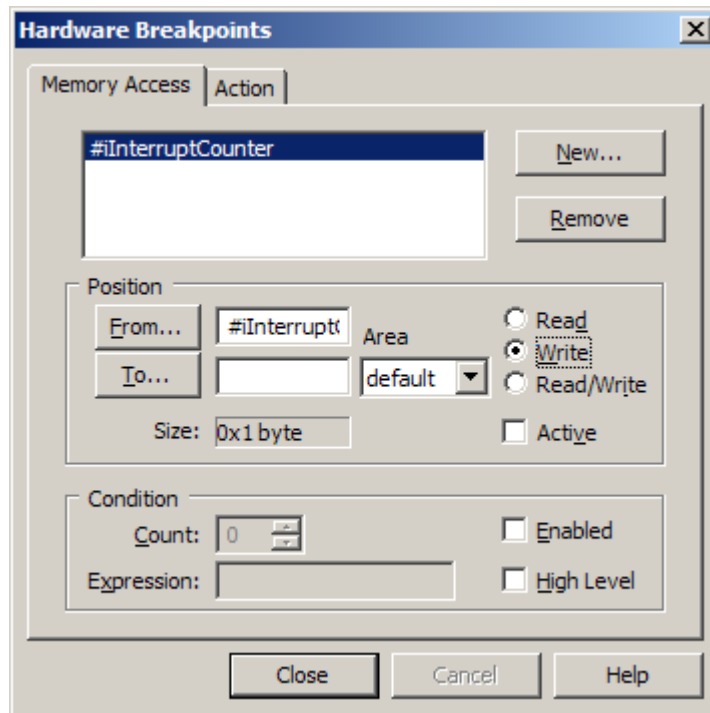
When monitor access to the CPU's memory is requested, the emulator stops the CPU and instructs it to read the requested number of bytes.

Since all accesses are performed using the CPU, all memory available to the CPU can be accessed. The drawback to this method is that memory cannot be accessed while the CPU is running. Stopping the CPU, accessing memory and running the CPU is an option, which, however, affects the real time execution considerably.

The time the CPU is stopped for is relative and cannot be exactly determined. The software has full control over it. It stops the CPU, updates all required windows and sets the CPU back to running. Therefore the time depends on the communication type used, PC's frequency, CPU's clock, number of updated memory locations (memory window, SFR window, watches, variables window), etc.

4 Access Breakpoints

Memory access breakpoints act when the CPU accesses a certain location. In-circuit Emulators can intercept memory accesses to any external memory space of the CPU. Memory access breakpoints are configured in the 'Memory Access' page of the breakpoint dialog.



Hardware Breakpoints dialog, Memory Access page

Position

Position defines the breakpoint's range, and access mode.

- 'From...' - defines the beginning of the breakpoint's region.
- 'To...' - defines the end of the breakpoint's region. If this field is left blank, the breakpoint's range is calculated from the 'From...' field as follows:
 - if the entry can be evaluated to a symbol, the range is configured to cover memory occupied by the symbol.
 - otherwise only the location specified in the 'From...' field is covered.

Note: You can always see the size of the range displayed in the Size field.

- Read, Write and Read/Write options define the type of access the breakpoint will act upon.

You can enable or disable a breakpoint by toggling the 'Active' option.

Condition

Conditional breakpoints are an advanced breakpoint feature. When a breakpoint is hit and the condition is enabled (the 'Enabled' option checked), following checks are performed:

- first the conditional expression is evaluated. If no expression is specified a non-zero result is assumed.
- if the expression evaluates to non-zero, the breakpoint occurrence counter is incremented.
- if the breakpoint occurrence counter is less or equal to the 'Count' specified, CPU execution is resumed, otherwise the CPU remains stopped.

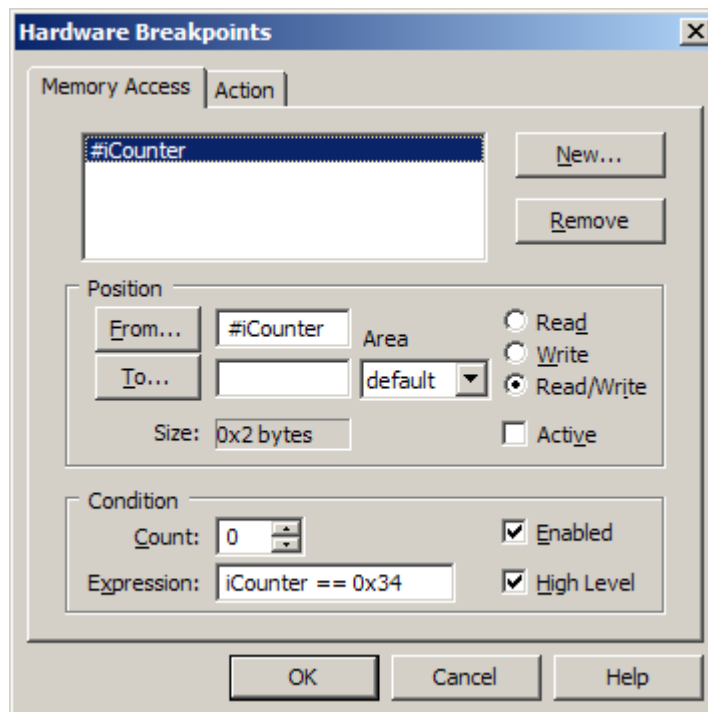
Note: All conditions are evaluated on the PC side thus incurring an overhead of up to 400ms.

If you need real-time complex breakpoints, use trace instead.

High Level

When this option is checked, the debugger performs one source debug step after the program stops on the breakpoint address and before the Condition expression is evaluated.

This ensures proper operation in cases like the following. For instance, there is a 16-bit variable `iCounter`. Access breakpoint is set on `iCounter` address and Condition '`iCounter == 0x34`' is specified.



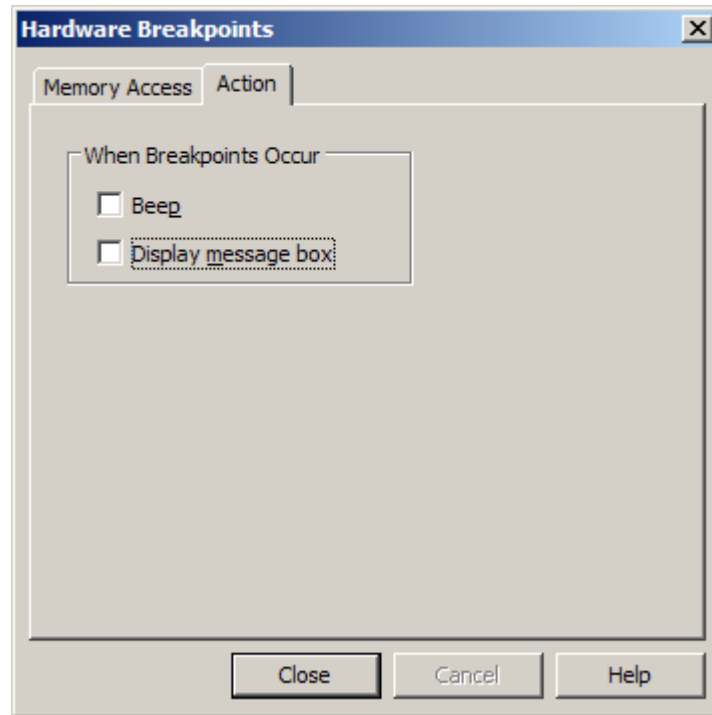
Now, there is this source code `iCounter=0x1234`; in the application, for which the compiler may generate two one-byte accesses (e.g. first store `0x34` and then store `0x12`). If high-level would not be checked, the debugger would break in such case although the condition doesn't really match. Breakpoint would first hit on address match and then the debugger would evaluate `iCounter`, which would match too although there is more code to be executed belonging to this source line. With 'High Level' option checked, the debugger executes additional source step after the breakpoint hits on address match. This ensures that all the code belonging to the source line is executed before the Condition is evaluated and also our conditional breakpoint would not stop the application.

Adding a Breakpoint

To add a breakpoint, click the 'New' button.

In the breakpoint location dialog specify the address for the breakpoint.

When Breakpoints Occur



Breakpoints dialog, Action page

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

This setting is global for all access breakpoints covered on the respective page.

5 Trace

The development system features full CPU bus trace, which allows detail insight into the program execution without stopping the CPU. It exposes application behaviour on the CPU access type, disassembly and C language level including the time information and all that without intrusion on the CPU execution time. Many application problems can be only resolved using the trace and its trigger/qualifier. Refer to a separate document for more details on trace configuration and its use.

6 Execution Coverage

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

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

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

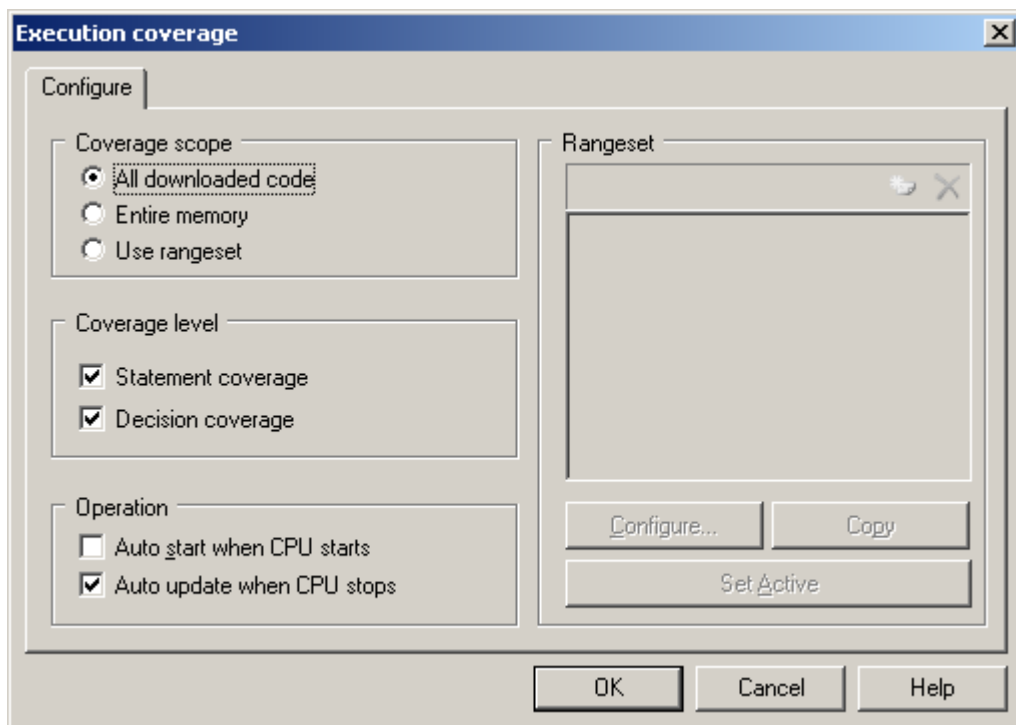
The major advantage of the real-time execution coverage is that it can run indefinitely, which does not apply for the off-line coverage, which is available on some architectures.

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

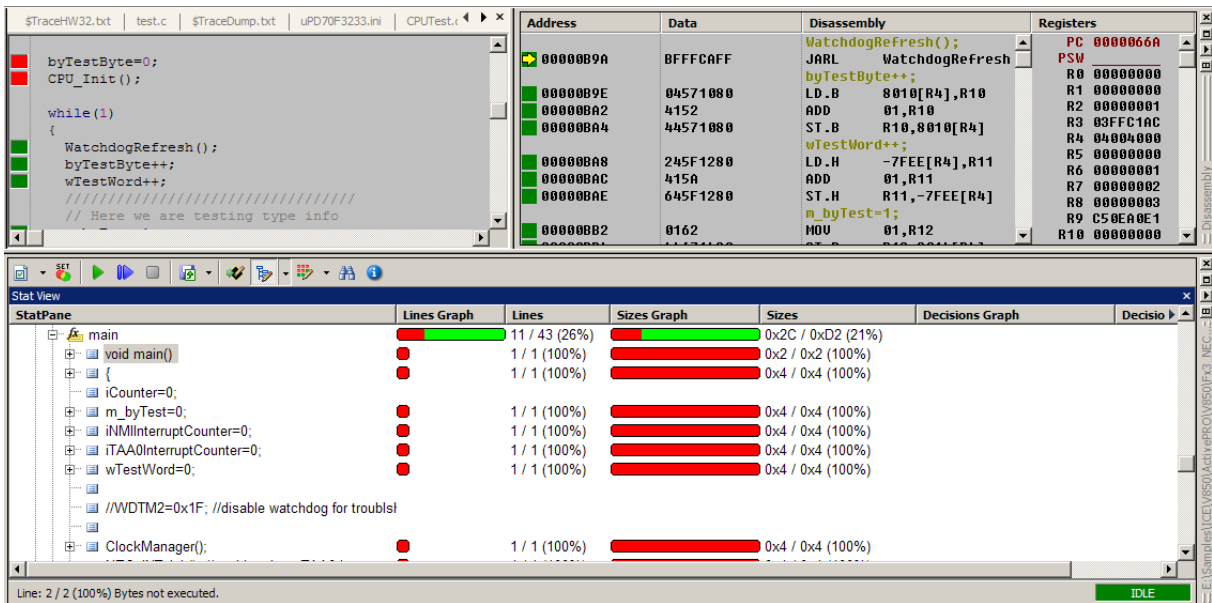
Trace, Profiler and Execution Coverage functionalities cannot be used at the same time since they are all based on the trace. Single functionality can be used at the time only

Typical Use

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



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



Execution Coverage results

7 Profiler

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

• Functions Profiler

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

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

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

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

• Data Profiler

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

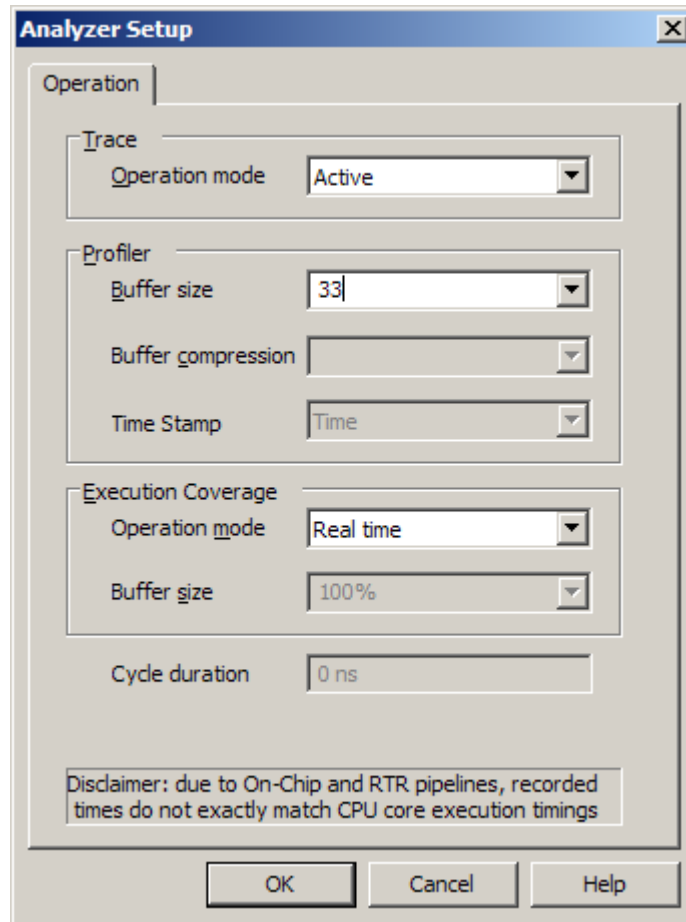
When an operating system is used in the application, task profiler can be used to analyze task switching. When the task profiler is used in conjunction with the functions profiler, functions' execution can be analyzed for each task individually.

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. Single functionality can be used at the time only.

Typical Use

To use profiler, first set working profiler buffer size in the 'Hardware/Analyzer Setup' dialog. Any value between 1 and 100 can be entered manually, representing used percentage of the available buffer.



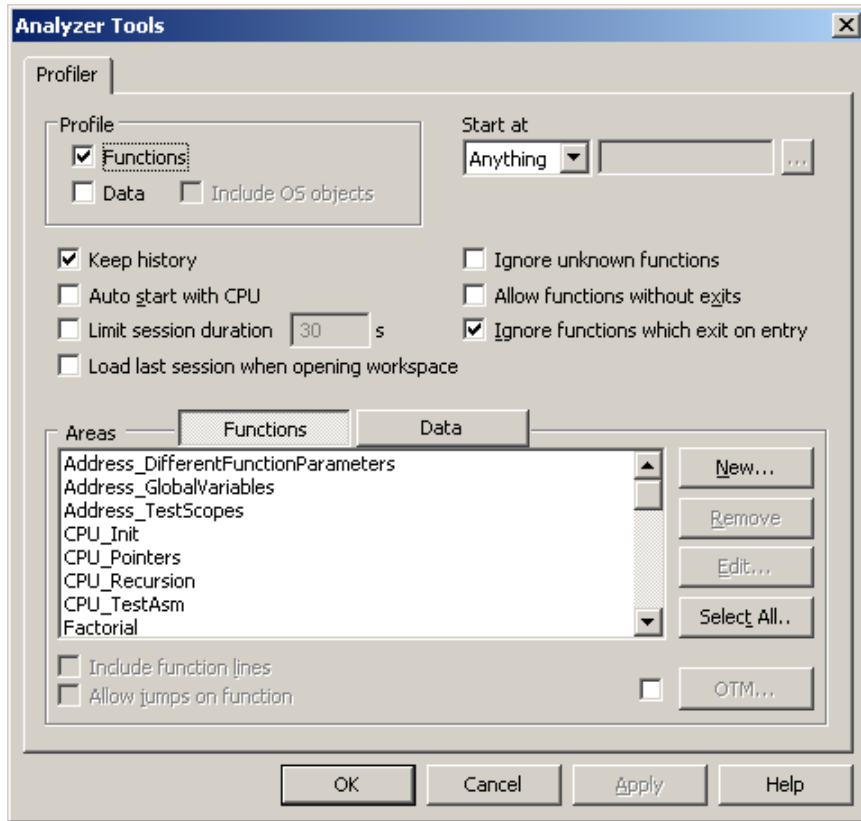
Next, select 'Profiler' window from the View menu and configure profiler settings (see next figure). Select 'Functions' option in the 'Profile' field when profiling functions. In order to profile a data variable, 'Data' should be checked instead. For instance, Data Profiler can be used as a Task Profiler,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

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

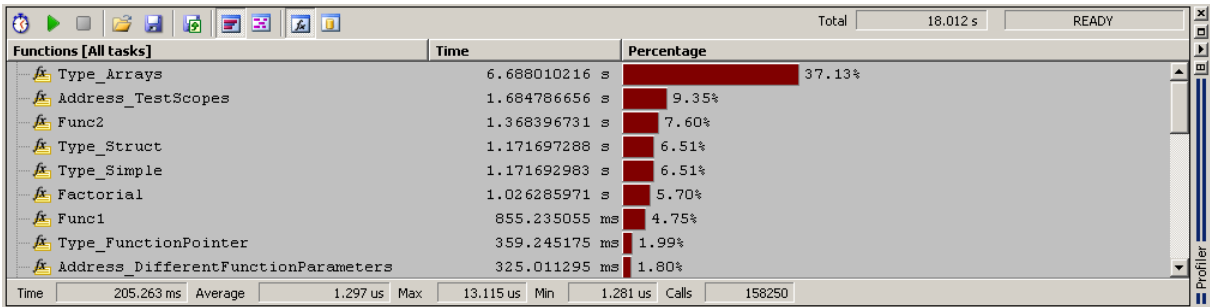
Finally, profiled functions are selected by pressing 'New...' button. It's recommended that 'All Functions' option is selected for the beginning.

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

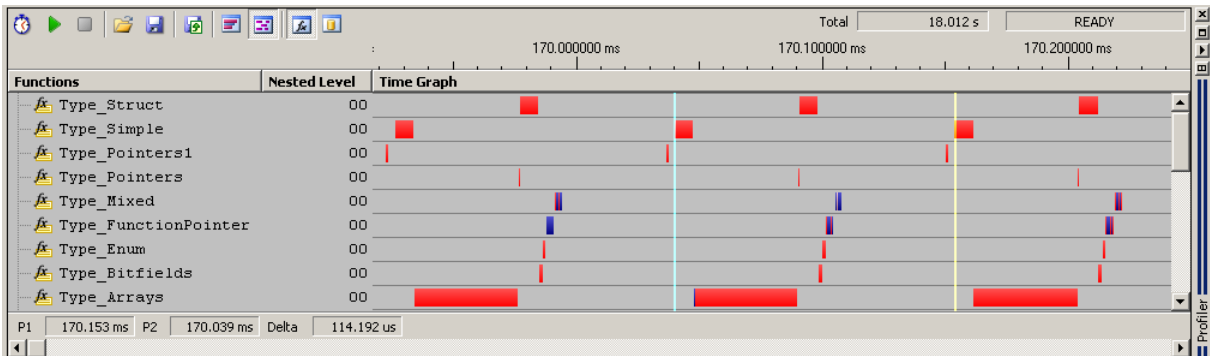


Profiler configuration settings

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



History view



Statistics view

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.