Vector MICROSAR Profiling

Application Note

September 2023

Contents

Overview

Task State and ISR Profiling via Instrumentation

Step 1: Enable ORTI and OS Timing-Hooks 5
Step 2: Instrument and Generate Profiler XML using iTCHi 6
Step 3: Configure winIDEA 8
Step 4: Start profiling 9

Non-Instrumented Task State and ISR Profiling

Step 1: Configure OS/RTE Profiling in DaVinci Configurator 12
Step 2: Use iTCHi to Generate Profiler XML 13
Step 3: Configure winIDEA and start profiling 14

Runnable Profiling via Instrumentation

Step 1: Enable the VFB Trace Hooks 17
Step 2: Generate the Instrumentation 18
Step 3: Configure winIDEA 20

Spinlock Profiling

Running Task and ISR Profiling

Step 1: Configure ORTI Export in DaVinci Configurator 23
Step 2: Configure winIDEA and start profiling 24

Trace Configuration

Infineon AURIX 26
- Basic Configuration 26
- Data Trace Single Variable 27
- Data Trace Address Range 29

ARM STM Trace 30
RH850 Software Trace 31
NXP/ST Power Architecture 33

Overview

This document explains how to profile and analyze the timing-behavior of Vector MICROSAR based AUTOSAR applications. You should be familiar with AUTOSAR classic profiling, the different types of Profiler objects (e.g., tasks, ISRs and Runnables) and the trace capabilities available on the microcontroller to properly utilize this resource.

Task State Profiling

The first step to analyze Vector MICROSAR is task state and running ISR analysis. There are two approaches for this type of analysis:

- Instrumented
- Non-Instrumented (also called *native*) task state profiling.

Instrumented task state and ISR profiling works by utilizing the Vector MICROSAR OS Timing Hooks. It is the recommended approach in cases where the amount of data comparators is limited (four or less) or when data tracing is not available. This approach works by writing into a single global variable or utilizing instrumentation trace capabilities by the microcontroller (see choosing the right instrumentation technique) and is straightforward to set up. The downside is that instrumentation is necessary.

Native task state and ISR profiling works by tracing the OS data structures of the Vector MICROSAR OS. Each task has its own state variable which combined with the running task variable represents the current state of a task. The microcontroller must provide enough data comparators to trace all of these variables to make this approach viable.

Note that Vector MICROSAR also supports Running Task and ISR profiling via the ORTI file. This approach does not provide task state information and is therefore not desirable for most use cases. Nevertheless, if your goal is basic CPU load analysis, you can follow the Running Task and ISR Profiling section.

Runnable Profiling

When task state and ISR profiling is working, the next step is to add Runnables.

Historically, Runnables can be profiled by utilizing program flow trace. Based on experience, this approach does not yield satisfactory results in most cases, either because of bandwidth limitations or because of intricacies in the compressed program flow logic by the semiconductor vendors. Therefore, instrumented Runnable profiling via instrumentation is the recommended approach. It works by instrumenting the RTE VFB trace hooks.

Spinlock Profiling

Lastly, the Vector MICROSAR OS Timing hooks provide support for Spinlock instrumentation. The winIDEA Analyzer supports Spinlock Profiling via these hooks.

Task State and ISR Profiling via Instrumentation

Task state and ISR profiling via instrumentation utilizes the Vector OS Timing Hooks to instrument task states and ISR executions. This approach provides detailed insights into the system's behavior by capturing task state transitions and ISR events. This method involves the following steps:

1 Enabling in DaVinci Configurator:

a. ORTI file generation

b. OS Timing Hooks generation

2 Configuring iTCHi to generate instrumentation code and Profiler XML file.

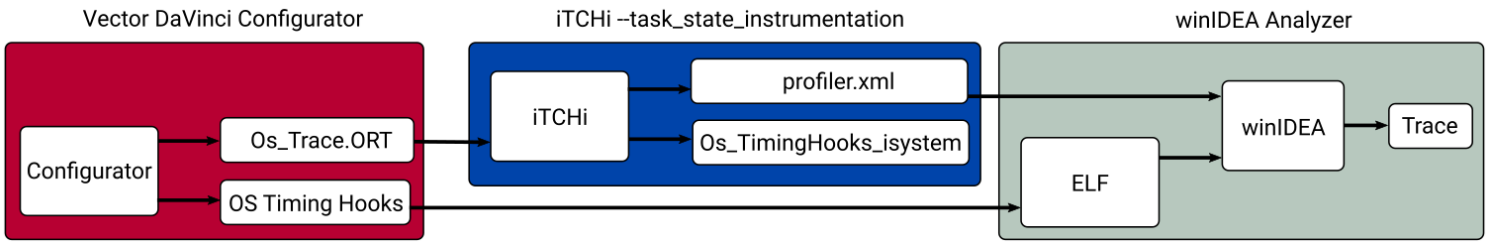
iTCHi is a program that helps users configure the winIDEA Analyzer to record OS and RTE aware hardware traces. You can find the Windows 64-bit executable `itchi-bin.exe` in the `scripts/itchi` directory of your winIDEA installation. A graphical-user interface is also available. To launch it, navigate to the OS tab under the Application settings in winIDEA. There, use the iTCHi Wizard button to launch the GUI. Note that the GUI is not necessarily self-explanatory and you probably want to continue reading this document.

▶ You can find more information via iTCHi Readme.

3 Recompiling the application with the generated instrumentation code.

4 Configuring winIDEA to analyze the trace data using the generated Profiler XML file.

5 Configuring hardware tracing to record the instrumentation variables.



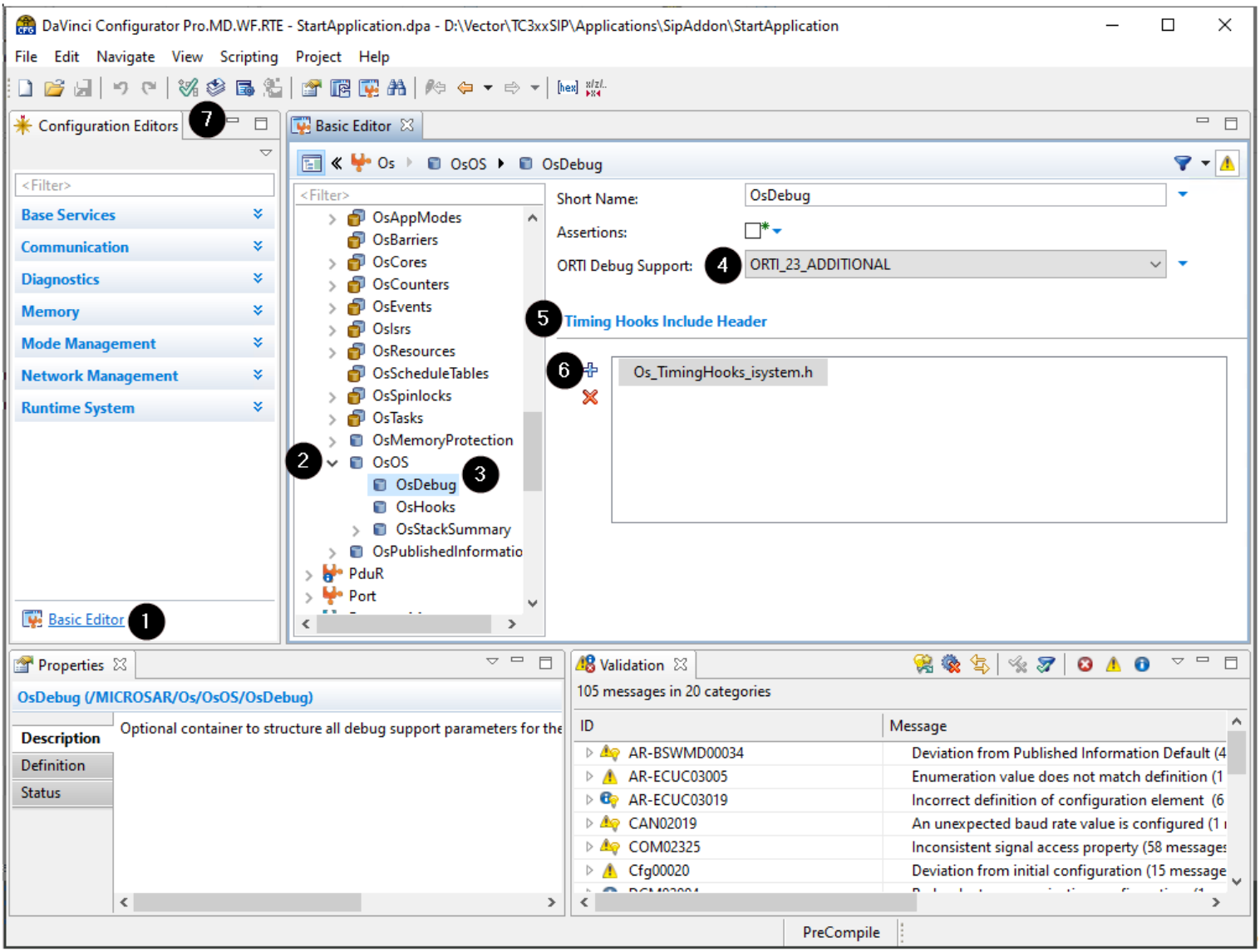
Vector Task State Instrumentation Workflow

Step 1: Enable ORTI and OS Timing-Hooks

For OS Task and ISR profiling via instrumentation, you need to enable ORTI file generation and the OS Timing Hooks in your Vector MICROSAR project. Follow these steps:

1. In DaVinci Configurator, open the Basic Editor.
2. In the Basic Editor, expand the OS node.
3. Navigate to the OsOS node and select the OsDebug node.
4. Activate ORTI Debug Support by selecting `ORTI_23_STANDARD` or `ORTI_23_ADDITIONAL`.
5. In the *OsDebug* view, locate the *Timing Hooks Include Header setting*.
6. Add a new header by clicking the plus symbol and name it `Os_TimingHooks_isystem.h`.
7. Regenerate the OS.

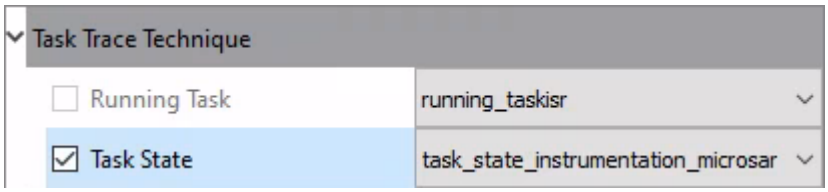
This process generates the OS ORTI file. After generating the OS, you should find a file `Os_Trace.ORT` in your `Appl/GenData` directory. It also enables the OS Timing Hooks, which can now be implemented in the specified header file.



Step 2: Instrument and Generate Profiler XML using iTCHi

To implement the OS hooks and generate the respective Profiler XML, follow these steps:

1. In the iTCHi wizard, make sure your *itchi.json* file is selected or create a new one if necessary.
2. Ensure your ORTI file and Profiler XML file are specified correctly.
3. Select the Task State analysis technique.
4. Choose `task_state_instrumentation_microsar` as the command and press *Next*.



5. Under `task_state_inst_microsar`, configure the OS instrumentation header and source file.

a. Point `vector_os_timing_hooks_h` to the `Appl/Include` directory of your project.
b. Point `vector_os_timing_hooks_c` to the `Appl/Source` directory of your project.
c. Leave the filenames as they are, so the string for the header file would be:

```
<your_davinci_dir>/Appl/Include/Os_TimingHooks_isystem.h
```

6. Depending on your microcontroller, pick the right `instrumentation_type`.

▸ Refer to section Trace Configuration for more information.

- In most cases, `data_trace` is the best approach.
- For RH850 controllers without data trace capabilities, select `software_trace`.
  - In this case, also change the `sft_dbpush_register` to `10`.
- For devices that have STM, select `stm_trace`.
  - In this case, also configure `stm_base_address` and `stm_channel`.
  - Note that the STM base address is device-specific.

7. Unselect `software_based_coreid_gen`.

This allows the winIDEA analyzer to get the core ID from the trace and usually works best.

8 Click *Generate* to create the instrumentation and Profiler XML file.

task_state_inst_microsar	
vector_os_timing_hooks_h	Appl/Include/Os_TimingHooks_isystem.h
vector_os_timing_hooks_c	Appl/Include/Os_TimingHooks_isystem.c
instrumentation_type	data_trace
software_based_coreid_gen	☐
trace_variable	isystem_trace
trace_variable_definition	
stm_base_address	0xE9000000
stm_channel	0x10
sft_dbpush_register	10

The process has generated the `Os_TimingHooks_isystem.c` and `Os_TimingHooks_isystem.h` source files, as well as the Profiler XML file. These files are now ready to be integrated into your project.

Adding C and H files to the build process

After generating the Profiler XML and instrumentation code, it's necessary to add the C and H files to the build process of your Vector MICROSAR application.

1 Copy the generated files to Appl\Source and Appl\Include directory in your project.

(If you did not generate them to those locations already.)

- `Os_TimingHooks_isystem.c`
- `Os_TimingHooks_isystem.h`

> *Adding* `Os_TimingHooks_isystem.c` *is only required for* `data_trace` *as it contains the definition of the trace variable.*

When using an Infineon AURIX with multiple cores, edit `Os_TimingHooks_isystem.h` to map the trace variable into global LMU RAM. There is a comment in the source file that explains how to do that.

2 Add `Os_TimingHooks_isystem.c` to one of the makefiles.

Include the following line in the appropriate section of the makefile:

a. APP_SOURCE_LST += Source\Os_TimingHooks_isystem.c

b. Again, this is only required for `data_trace`.

3 Build your application.

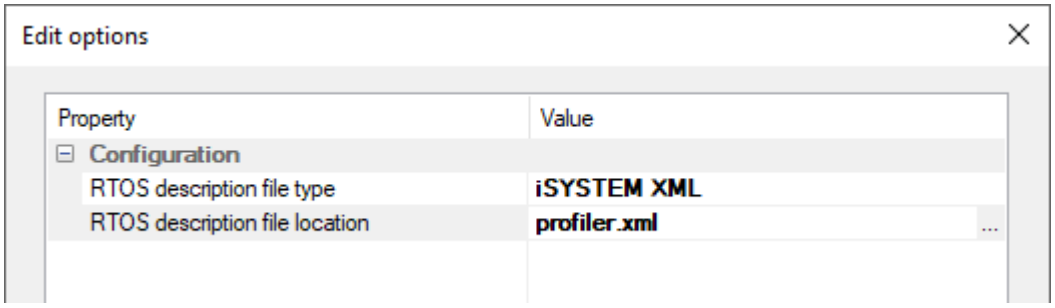
Open a command prompt in the build directory, and execute the following command: `.\m.bat`

Once the build has finished, download the instrumented application via winIDEA. For data tracing, add the `isystem_trace` variable to a Watch Window and confirm that it changes when you enable real-time updates. For instrumentation tracing, this step does not apply.

Step 3: Configure winIDEA

To configure winIDEA to use the generated Profiler XML file, follow these steps:

1. In winIDEA, navigate to *Debug / Configure Session / Applications / OS*.
2. Select and configure AUTOSAR.



▸ How to configure AUTOSAR OS?

3. Perform a *Download* or *Load Symbols* action to apply the changes.

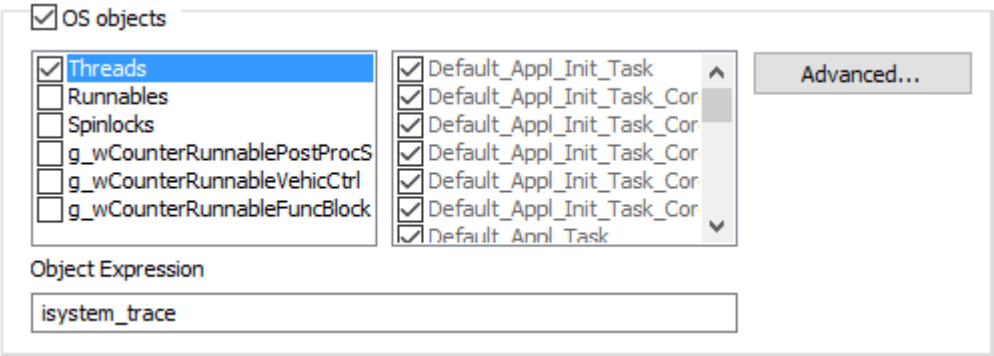
Step 4: Start profiling

After configuring the Profiler XML for Task State and ISR profiling, you can now use the winIDEA Analyzer to record and profile the instrumentation data.

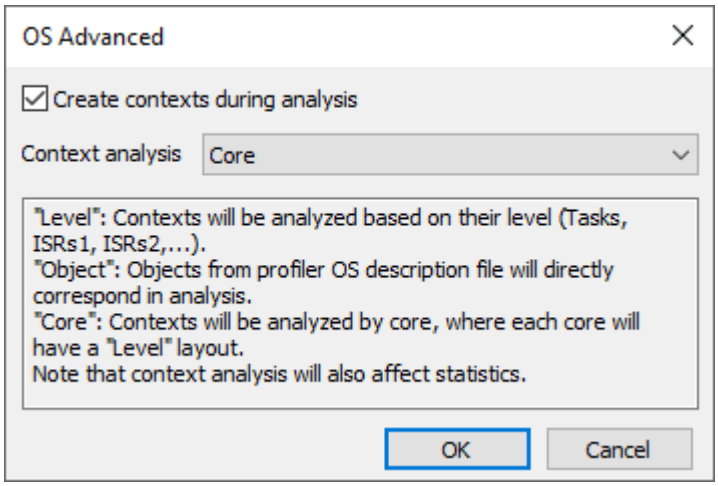
1 Create new Automatic Trace Configuration.

▸ Need help with configuring Automatic Trace Configuration?

2 Enable *OS Objects* and *Threads* under *OS Objects*.



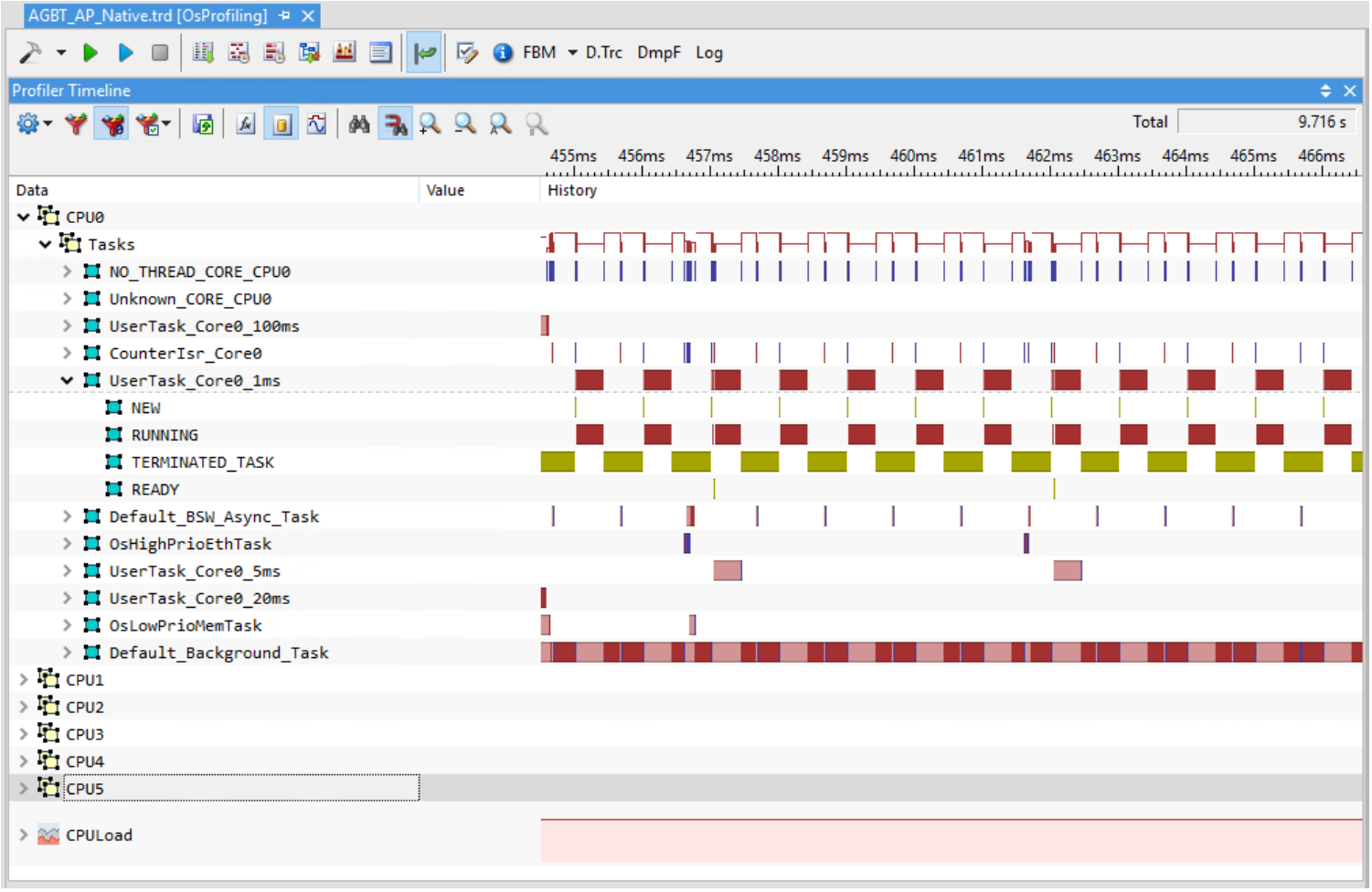
3 (Recommended) For multi-core systems, change the *Context analysis* to *Core* via *Advanced* button.



> *Some architectures might require manual hardware trigger configuration. If you don't see any data, manually configure the hardware trace to record the* `isystem_trace` *variable. Also, check that you have configured the correct Analyzer cycle duration.*

4 Start the Analyzer session.

If everything is set up correctly, you should see a trace like the one shown below.



If you don't see any data or the data does not look plausible, please check the [Knowledge Base](#).

Non-Instrumented Task State and ISR Profiling

This section explains how to profile Task states and running ISR information utilizing data tracing without instrumentation (meaning no changes to the source code are necessary).

To facilitate this use case, the ORTI file provides a `STATE` attribute for each task. By evaluating the state expression, the state of the task at a given moment can be deduced. The challenge with Vector MICROSAR state tracing is that the state attribute consists of multiple variables as shown in the following listing.

```
TASK Default_Appl_Init_Task {
PRIORITY = "OsCfg_Task_Default_Appl_Init_Task_Dyn.Priority";
STATE = "OsCfg_Core_OsCore_Core0_Status_Dyn.OsState == 2 ? (
OsCfg_Trace_OsCore_Core0_Dyn.CurrentTask == &OsCfg_Trace_Default_Appl_Init_Task ?
0 : OsCfg_Task_Default_Appl_Init_Task_Dyn.State ) : 0xFF";
/* other attributes here */
}; /* Default_Appl_Init_Task */
```

To configure this use case properly, two points have to be kept in mind:

- *Enough data trace comparators to record all variables that are part of the state expression must be available.*
- *Variables that don't change during the trace recording, have to be preset to their expected value in the Profiler configuration.*
  - *For example, the OsState variable in the listing above is only written once at the startup of the application. To be able to start a recording at a different point in time, this variable has to be set to 2.*
  - *Similarly, for a background task, the **State** variable always has the value 1 for **READY**.*

The following configuration steps are required for state analysis without instrumentation:

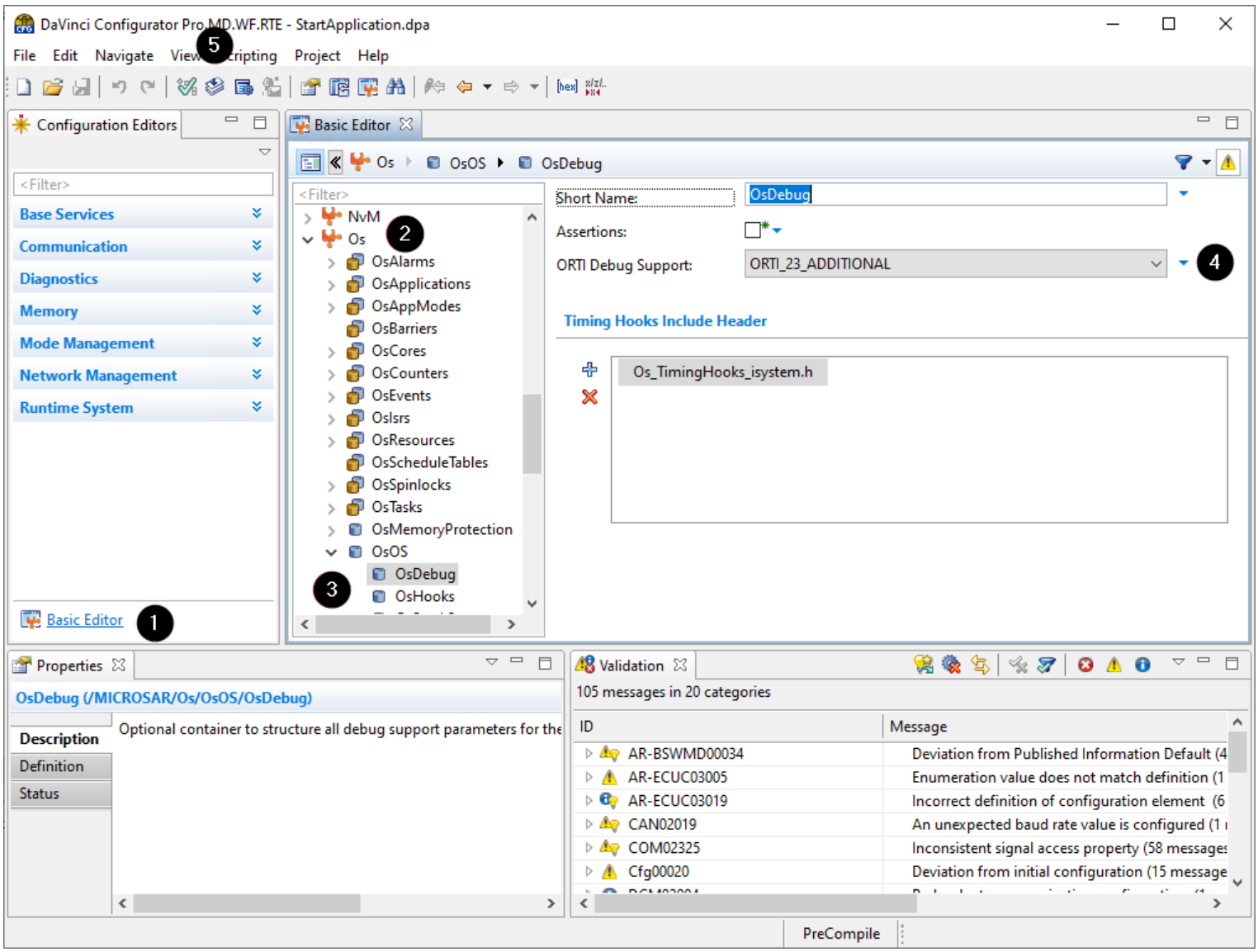
1. Enabling ORTI file generation in *DaVinci Configurator*.
2. Configuring iTCHi to generate the Profiler XML file.
3. Configuring winIDEA to record all variables that are part of the STATE expression.
4. Configuring winIDEA to analyze the trace data using the generated Profiler XML file.
5. Configuring hardware tracing to record the instrumentation variables.

Step 1: Configure OS/RTE Profiling in DaVinci Configurator

For OS Task and ISR profiling, you need to enable ORTI file generation in your Vector MICROSAR project. Follow these steps:

1. In DaVinci Configurator, open the *Basic Editor*.
2. In the Basic Editor, expand the *OS node*.
3. Navigate to the *OsOS* node and select the *OsDebug* node.
4. Activate ORTI Debug Support by selecting `ORTI_23_STANDARD` or `ORTI_23_ADDITIONAL`.
5. Regenerate the OS.

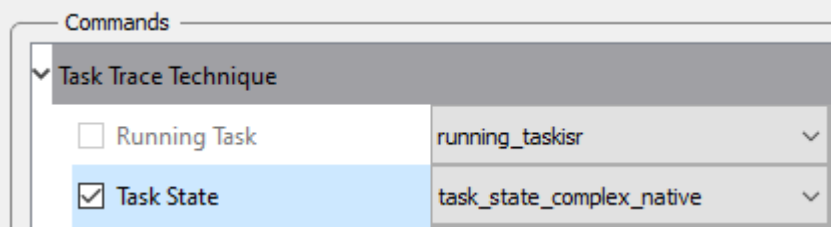
This process enables the OS ORTI file generation. After generating the OS, you should find a file `Os_Trace.ORT` in your *Appl/GenData* directory.



Step 2: Use iTCHi to Generate Profiler XML

To generate a *Profiler XML* for task state and ISR analysis without instrumentation, follow these steps:

1. In the *iTCHi wizard*, make sure your *itchi.json* file is selected.
2. Ensure your *ORTI* file and *Profiler XML* file are specified correctly.
3. Select the *Task State analysis* technique.
4. Choose `task_state_complex_native` as the command and press *Next*.

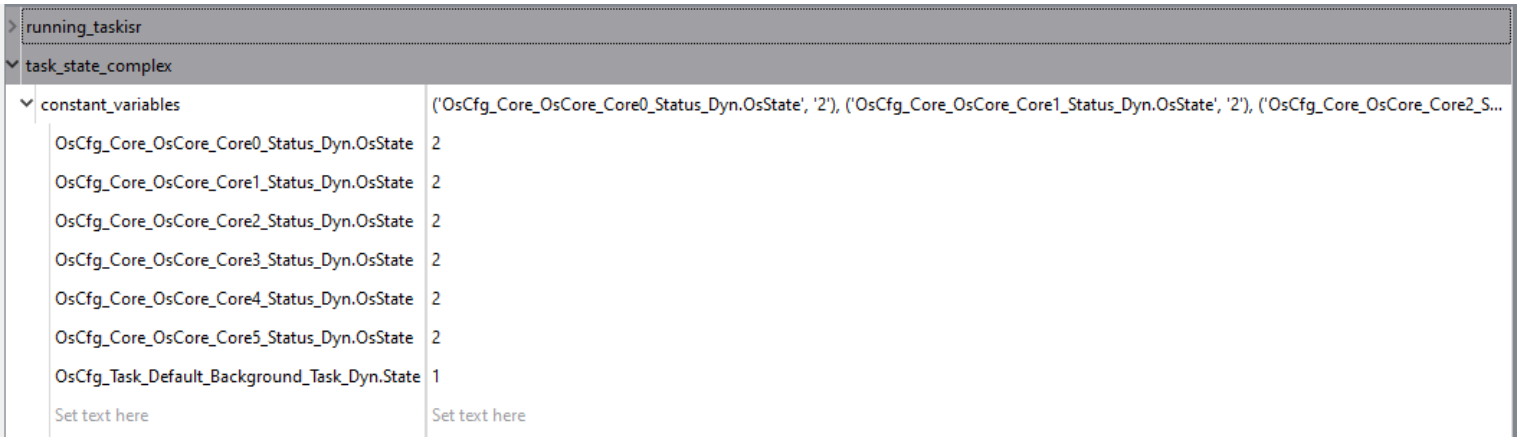


You can leave the `running_taskisr` settings as they are.

5. Under `task_state_complex`, use `constant_variables` to set all non-changing variables to their expected constant value.

- For Vector MICROSAR, set *OsState* variables to 2. The screenshot below shows how this would look like for an application that uses six cores. Also set task State variables to 1 (meaning `READY`) for tasks that might not otherwise change their state. This might be required for background tasks.
- If you start recording before the startup of the application, you don't have to do this.
- For other operating systems, you can leave `constant_variables` empty.

6. Click *Generate* to create the Profiler XML.



Step 3: Configure winIDEA and start profiling

After adding the generated Profiler XML to winIDEA, follow these steps to finish the configuration:

1 Create a new Manual Trace Configuration via *View / Analyzer / Create New Configuration*.

▸ Need help with configuring Manual Trace Configuration?

2 Enable data trace for all variables that are part of the state expressions.

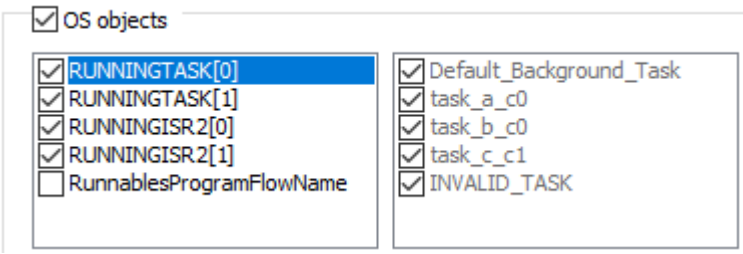
▸ Follow the Trace Configuration.

You can use the iTCHi `--log_trace_symbols` flag if you are not sure which variables are required.

3 Press OK and reopen the configuration, and add the relevant Application.

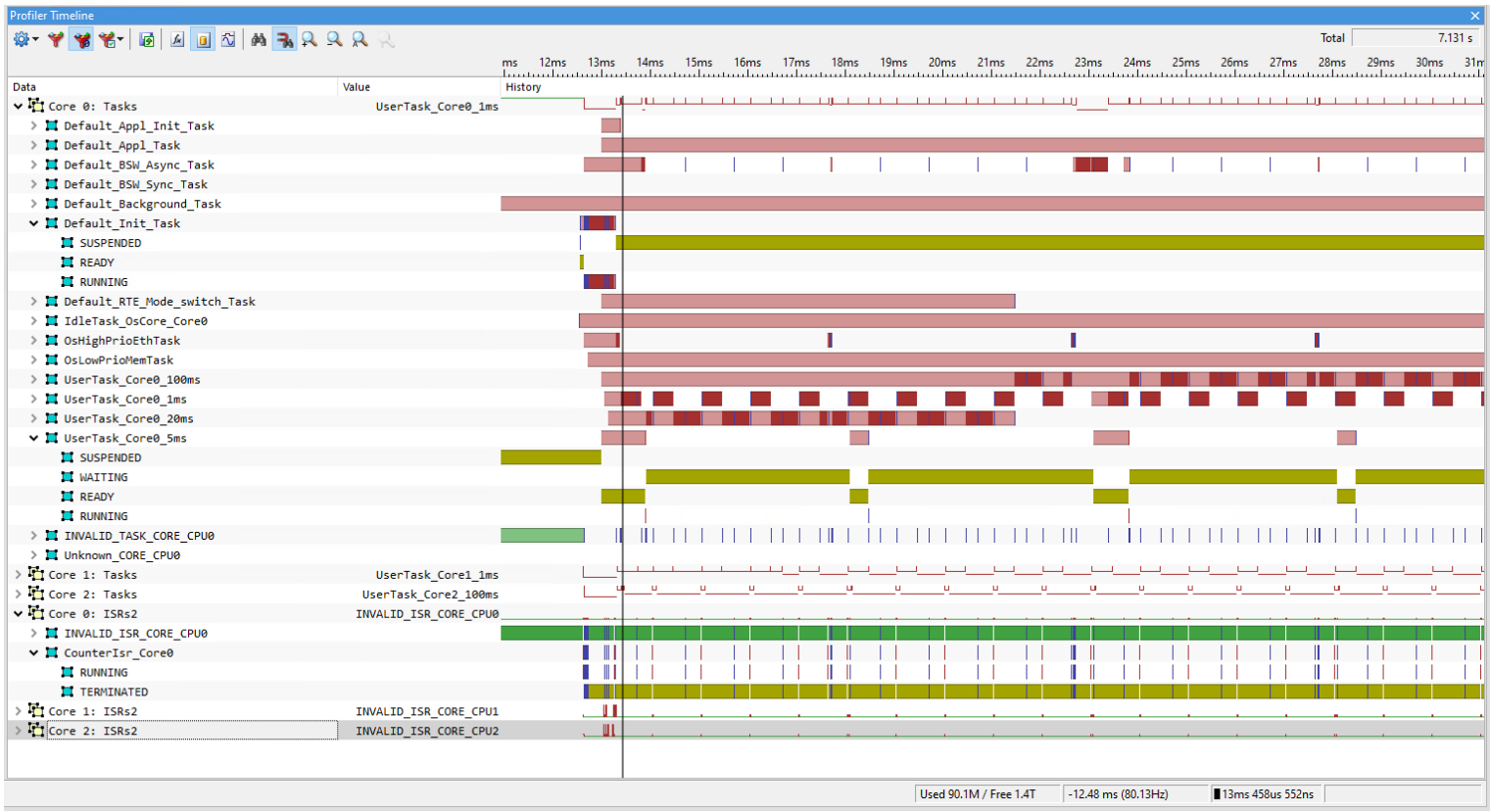
The application has to include all cores that you want to profile.

4 Enable OS Objects and all task and ISR objects.



5 Start the Analyzer session.

If everything is set up correctly, you should see a trace like the one shown below.

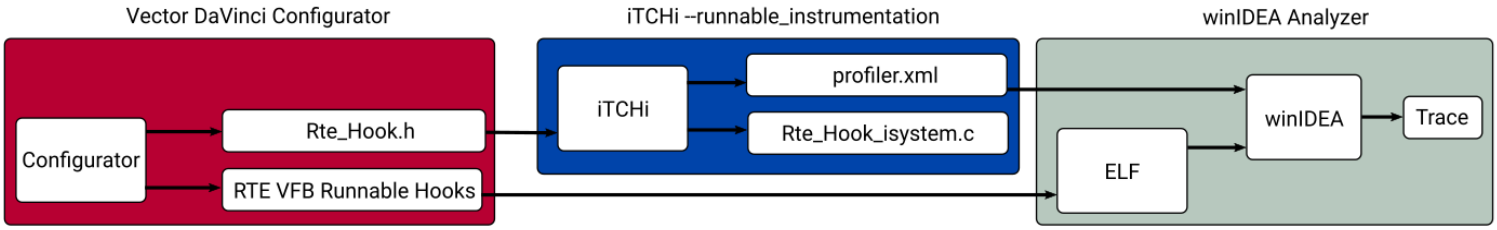


If you don't see any data or the data does not look plausible, please check the [Knowledge Base](#).

Runnable Profiling via Instrumentation

Runnable profiling via instrumentation utilizes the AUTOSAR RTE Virtual Function Bus (VFB) trace hooks. This approach provides insights into the Runnable runtime behavior in addition to Tasks and ISRs. It involves the following steps:

1. Enabling the VFB trace hooks in DaVinci Configurator.
2. Configuring iTCHi to generate instrumentation code and *Profiler XML file*.
3. Recompiling the application with the generated instrumentation code.
4. Configuring winIDEA to analyze the trace data using the generated *Profiler XML file*.
5. Configuring hardware tracing to record the instrumentation variables.



Runnable Instrumentation Workflow

Step 1: Enable the VFB Trace Hooks

For Runnable profiling via instrumentation, enable the RTE VFB trace hooks.

1. In *DaVinci Configurator*, navigate to *Runtime System General*, and then to Rte VFB Tracing.
2. Enable the checkbox next to Enable VFB Tracing.
3. Add the start and return hooks for RTE tracing using the *Import VFB Trace Functions Assistant*.
4. Select the *Rte_Hook.h* file via GenData folder of the application project.

- Typically, `Rte_Runnable` and `SchM_Schedulable` hooks are selected.
- Note that iTCHi also requires this file to implement the Runnable hooks.

5. Select the hooks, click *Finish* and generate the RTE.

This process enables the VFB Runnable hooks, which can now be implemented via iTCHi.

Step 2: Generate the Instrumentation

To implement the RTE hooks and update the Profiler XML for Runnables, follow these steps:

1. In the *iTCHi* wizard, make sure your itchi.json file is selected.
2. Ensure your ORTI file and Profiler XML file are specified correctly.

Keep existing settings for Task and ISR tracing as they are.

3. Select Runnable Tracing.
4. Choose `runnable_instrumentation` as the command and press *Next*.

5. Configure the following:

- Under `runnable_instrumentation`, adjust the settings to your project.
- Under `isystem_vfb_hooks_c`, specify the name of the instrumentation file into which iTCHi generates the instrumentation code, for example `Rte_Hook_isystem.c`.
  - Include that file to the build process later to implement the Runnable hooks. To avoid copying the file manually, generate it into the Appl/Source directory of your project.
  - Optional: To edit the hooks template, set `template_file` to `Rte_Hook_isystem_TEMPLATE.c`. iTCHi will then use that template file in the next run. The template uses the Python Jinja2 syntax.
- Under `rte_hook_h`, reference the *Rte_Hook.h* file located in the *Appl/Source* project of the application project.

6. Depending on your microcontroller, pick the right `instrumentation_type`.

Refer to [Trace Configuration](#).

7. Uncheck `software_based_coreid_gen`.

On most relevant architectures, winIDEA can infer the core ID from the trace messages.

8 Click *Generate* to create the instrumentation and Profiler XML file.

runnable_instrumentation	
isystem_vfb_hooks_c	Rte_Hook_isystem.c
rte_hook_h	Rte_Hook.h
rte_xdm	
regex	(FUNC\(void, RTE_APPL_CODE\) Rte_Runnable_(\w+)_(Start|Return)\([^\n]+\))
trace_variable	isystem_trace_runnable
trace_variable_definition	
template_file	Rte_Hook_isystem_TEMPLATE.c // THIS IS OPTIONAL
instrumentation_type	data_trace
software_based_coreid_gen	☐
stm_base_address	0x0
stm_channel	0x0
sft_dbpush_register	11
sft_dbtag	☑

This process generates the *Rte_Hook_isystem.c* instrumentation file, and updates the Profiler XML for Runnable profiling.

Rebuild the application

After enabling and generating the VFB Runnable trace hooks, follow these steps to rebuild your application.

1 Copy the generated *Rte_Hook_isystem.c* into the *Appl\Source* directory.

2 Add *Rte_Hook_isystem.c* to one of the makefiles.

Including the following line in the appropriate section of the makefile:

```
APP_SOURCE_LST += Source\Rte_Hook_isystem.c
```

3 Build your application.

Open a shell or command prompt in the build directory and execute the following command:

```
.\m.bat
```

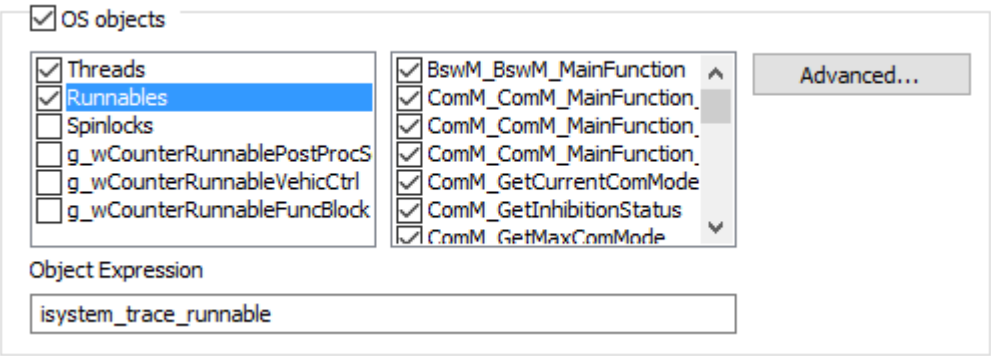
4 Download the instrumented application via winIDEA.

5 For data tracing, add the `isystem_trace_runnable` variable to a Watch window,

This way you can confirm that it changes when you enable real-time updates. For instrumentation tracing, this step does not apply.

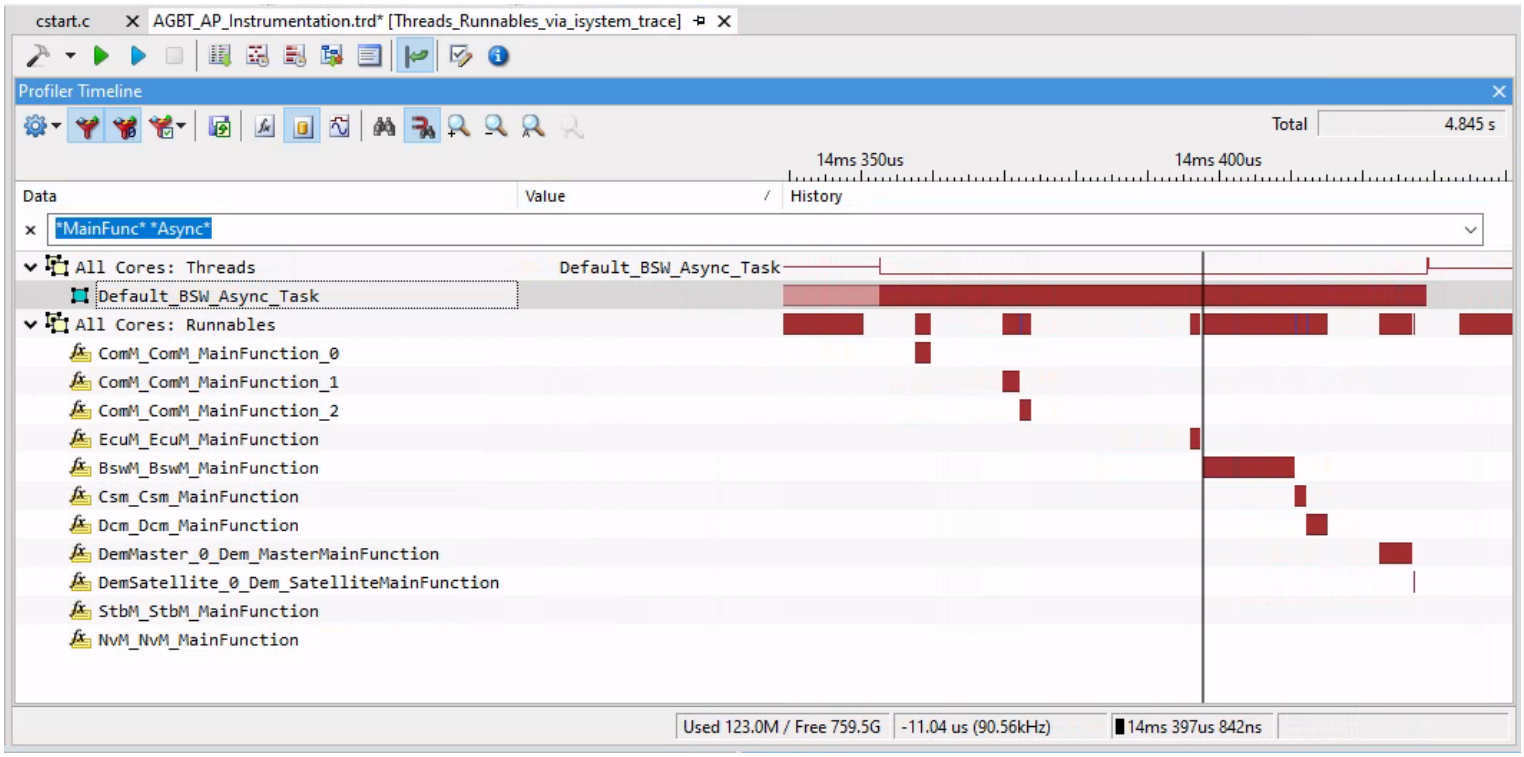
Step 3: Configure winIDEA

1. Add the Profiler XML to winIDEA.
2. Create an Analyzer configuration.
3. Download the application.
4. Check Runnables in addition to Threads or Tasks and ISRs.



5. Start the Analyzer session.

If everything is set up correctly, you should see a trace similar to the one shown in the screenshot below.



Visualization of an OS Task and its Runnables in the winIDEA Analyzer.

> *Some architectures might require manual hardware trigger configuration. If you don't see any data, manually configure the hardware trace to record the* `isystem_trace_runnable` *variable. Also, check that you have configured the correct Analyzer cycle duration.*

Spinlock Profiling

Spinlock Profiling utilizes the Vector OS Timing Hooks. It only works in combination with instrumented Task and ISR profiling.

> ⚠ *Spinlock Profiling is an experimental feature and currently only works with data tracing.*

Step 1: Setup Vector MICROSAR

Spinlock profiling utilizes the OS Timing Hooks, which should already be enabled. If not, follow the section Task State and ISR Profiling via Instrumentation first, before continuing with this section.

Step 2: Setup iTCHi

For Spinlock profiling to work, two changes are necessary. First, in addition to the thread instrumentation, iTCHi also has to generate the Spinlock instrumentation into *OS_TimingHooks_isystem.h*. Second, iTCHi has to update the Profiler XML.

1 Open the iTCHi wizard, and load the existing itchi.json.

2 Check `task_state_instrumentation` and `spinlock_instrumentation` command and

If you haven't configured `task_state_instrumentation` yet, do that first.

3 Press *Next* and make sure that `spinlock_generate_instrumentation` is selected.

In the attribute configuration dialog, there is now a new area spinlock_instrumentation.

4 (optional) Override the definition of the spinlock trace variable.

You can optionally use the attribute `spinlock_trace_variable_definition` to override the definition of the spinlock trace variable. You can use `{{spinlock_trace_variable}}` inside the definition string to be replaced by the attributed defined in `spinlock_trace_variable`.

5 Click *Generate*.

After iTCHi has finished, carefully check the output to confirm that the instrumentation files have been written. If they have not been written because they already exist, delete or rename them and generate again. Rebuild your application as before. In winIDEA, make sure that the spinlock variable exists and that it changes in the watch window if real-time updates are enabled.

Step 3: Setup winIDEA

In the winIDEA Analyzer, open your existing Profiler configuration. In addition to Threads, also enable Spinlocks. Depending on your architecture, manually configure the hardware trace of the spinlock variable and start a new recording.

Running Task and ISR Profiling

Running task and ISR analysis works by data tracing the running task and ISR attribute for each AUTOSAR core.

The **advantage** of this approach is that it does not require instrumentation or special configuration via iTCHi.

The **disadvantage** is that profiling the running object can lead to ambiguous results in the interpretation of the data. For example, if the running task switches from one to another, the reason why the first task stops is unknown to the profiler. This is acceptable for CPU load analysis, but makes other use cases infeasible.

For use cases such as response time requirements verification and event-chain analysis, use a task state profiling approach.

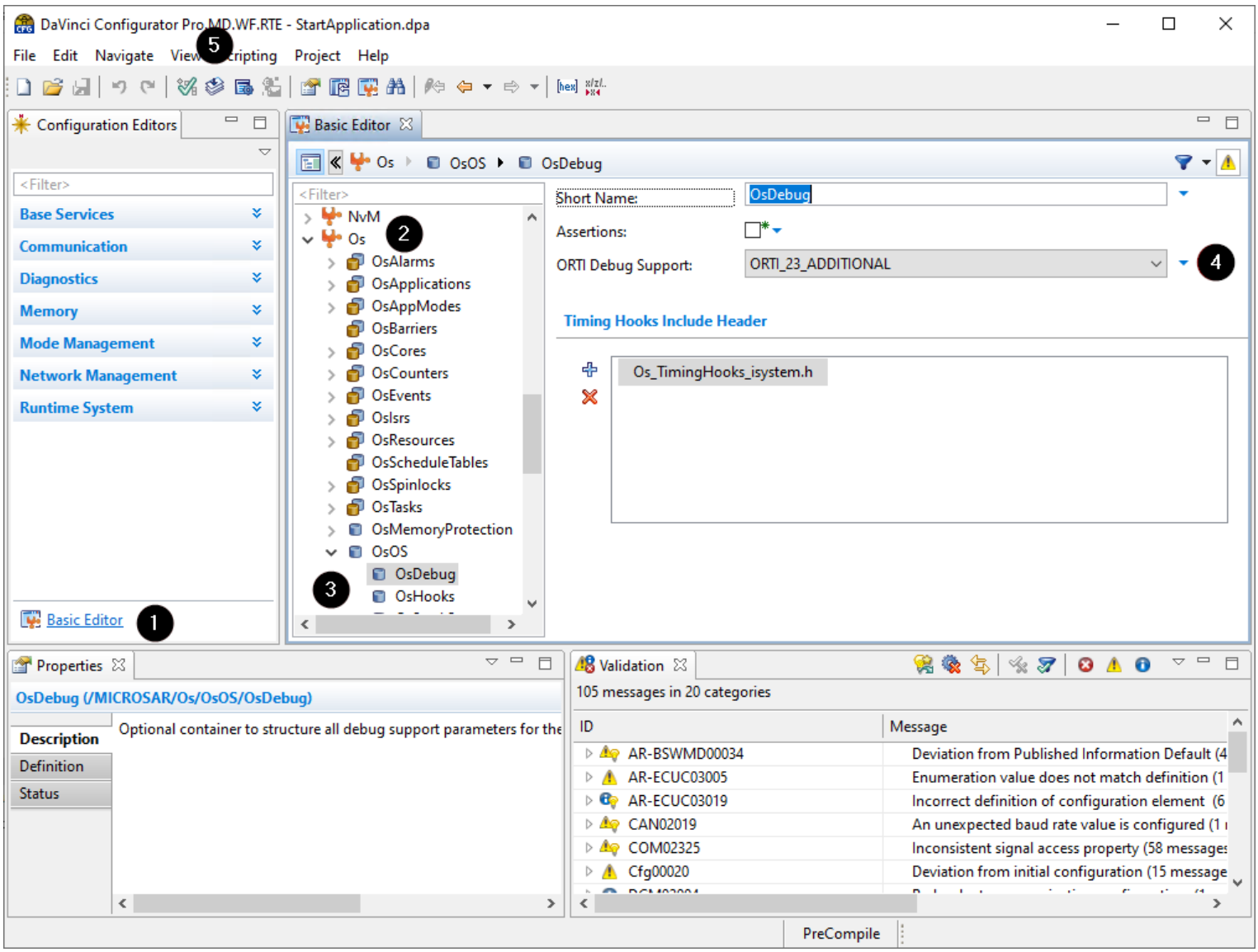
1. Generate an ORTI file.
2. Add ORTI file to winIDEA.
3. Configure winIDEA Analyzer.

Step 1: Configure ORTI Export in DaVinci Configurator

For OS Task and ISR profiling, you need to enable ORTI file generation in your Vector MICROSAR project. Follow these steps:

1. In DaVinci Configurator, open the *Basic Editor*.
2. In the Basic Editor, expand the *OS node*.
3. Navigate to the *OsOS* node and select the *OsDebug* node.
4. Activate ORTI Debug Support by selecting `ORTI_23_STANDARD` or `ORTI_23_ADDITIONAL`.
5. Regenerate the OS.

This process enables the OS ORTI file generation. After generating the OS, you should find a file `Os_Trace.ORT` in your *Appl/GenData* directory.



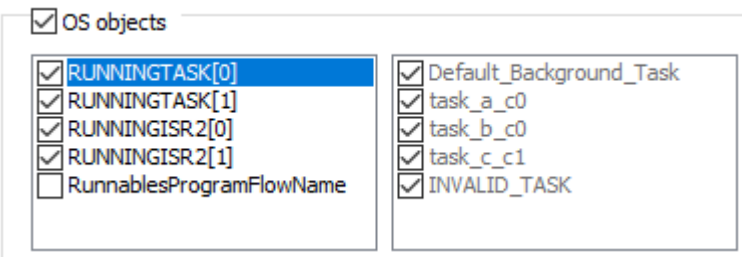
Step 2: Configure winIDEA and start profiling

After generating the ORTI file, follow these steps to add it to winIDEA:

1. Add the generated Profiler XML to winIDEA.
2. Open the Analyzer and create a new trace configuration.

Make sure to add the application that references the ORTI file.

3. Select the RUNNINGTASK and RUNNINGISR2 attributes via Profiler page.



4. Start a new trace recording.

If everything is set up correctly, you should see a trace like the one shown in the screenshot below.

Note that the tasks only have the two states RUNNING and TERMINATED. As mentioned before, this is not sufficient for many use cases where additional states such as WAITING and READY are required.

If you don't see any data or the data does not look plausible, please check the Knowledge Base.
Specifically, a manual trigger configuration may be necessary.

Trace Configuration

Right Instrumentation Technique

Hardware tracing relies on the trace capabilities provided by the microcontroller. Depending on the microcontroller, one of the following trace techniques must be employed:

Architecture	Instrumentation Technique	Additional Information
Infineon AURIX NXP/ST Power Architecture ARM Cortex-M	`data_trace`	In most cases, `data_trace` is the best approach.
Renesas RH850*	`software_trace`	Keep `sft_dbtag` checked (software trace instrumentation will use the more efficient DBTAG instructions. This attribute is not relevant for other instrumentation types)
ARM Cortex-M ARM Cortex-R	`stm_trace`	• Configure `stm_base_address` and `stm_channel`. • STM base address is device-specific.

**The term RH850 software trace can be misleading, as it actually refers to an instrumentation trace technique that utilizes hardware instructions such as DBPUSH and DBTAG. This technique is restricted to recording traces for only one core at a time, which may limit its application for multi-core applications.*

Hardware tracing depends on the capabilities provided by the microcontroller. In doubt, contact the support team if you have questions about the possibilities on a certain microcontroller.

Infineon AURIX

Infineon AURIX Data Trace

These sections explain how to configure data trace for the Infineon TriCore architecture. The basic configuration for all trace use cases is the same, so make sure to follow the steps in the Basic Configuration section.

Basic Configuration

This section gives you a starting point for more complex TriCore configurations. To create a start configuration, execute the following steps.

1 Select Operation Mode via *Hardware / CPU Options / Analyzer / Operation Mode*.

- On-Chip for DAP
- Aurora Trace Port for AGBT

2 Create a new Manual Trace Configuration via *View / Analyzer / Create New Configuration*.

▸ Need help with configuring Manual Trace Configuration?

3 In the *Recorder* page:

a. disable *Timer Interpolation*.

b. select *Upload while sampling* when using a DAP.

4 In the *MCDS* page set:

a. set the EMEM Trigger Position to Begin.

b. assume timestamp source to be tick.

5 Under the *MCX* page set:

a. `trace_done` to *Never.*

b. `tick_enable` to *Always*.

6 Save the configuration.

Data Trace Single Variable

Assuming you have a basic TriCore trace configuration, this section shows how to add a data-trace trigger for a specific variable.

> *This section assumes that you have followed the instructions to map the trace variables into global LMU RAM. If that is not the case and you want to trace a variable from core local scratchpad RAM (e.g., 0x7000'0000 address range), replace BOB with POB X and select a specific core. Then, do the trigger configuration under TriCore X (instead of SRI).*

1. Open your Analyzer Configuration and select *Configure* under *Manual Hardware Trigger*.
2. In the MCDS, configure SRI 1 to observe SRI slave LMU0.
3. Under the SRI, configure data tracing for a specific variable.
4. Specify a *Trigger* for the variable.

a. Double-click an available `dtu_ea_trig` such as `dtu1_ea_trig_0`.

b. Configure the trigger to work as a ranger comparator `X <= ADDR <=Y`.

c. Select the variable (or address) you want to trace and tick the check box for *Entire Object*.

5. Find an *Event* that maps to the trigger, enable it, and tick the respective trigger.
6. Specify the *Action*.

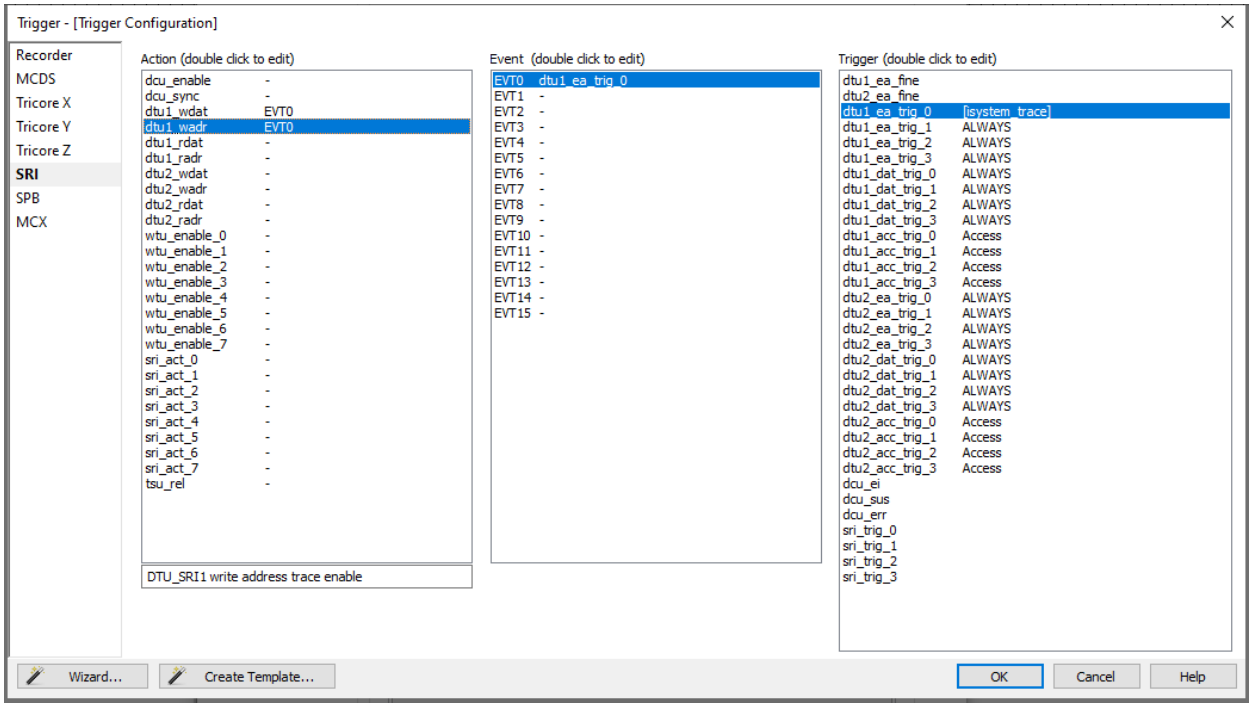
a. Activate `dtu_wdat` and `dtu_wadr` for the event you have selected.

b. Set the respective *Qualifier* on *Active*, the *Level* on *State*, and the event you have chosen in the previous step.

c. Make sure to do this for the data and the address actions.

d. To test this configuration, it's best to first trace a simple global variable that is know to change (such as a counter), and make sure that the write events appear in the trace output.

The following screen shot shows a working configuration for the variable `isystem_trace`.



Data Trace Address Range

Recording a data trace for an address range works similarly to the configuration for a single variable. The difference is that you **specify two variables or addresses** instead of a single variable.

1. Execute the steps from the previous section.
2. Deselect *Entire Object*.
3. Specify a start and end address or symbol.

- When specifying the range via symbols, the first variable, all variables in between, and the last variable are part of the memory range. The only exception is when the Y variable has a complex data type. In that case, it is necessary to expand the complex variable and select the last element. Otherwise, the chip may not record access to the Y variable.
- Instead of specifying symbols, it is also possible to enter addresses directly into the X and Y fields. Specify the raw addresses in hexadecimal form. For example, 0x0 and 0x70002000 are valid addresses. The Y value must be higher than the X value.

ARM STM Trace

STM is an instrumentation trace technique where writes into dedicated channels that are part of so-called Stimulus ports generate data trace messages. To configure STM tracing, follow these steps.

1 Create a new Manual Trace Configuration via *View / Analyzer / Create New Configuration*.

▸ Need help with configuring Manual Trace Configuration?

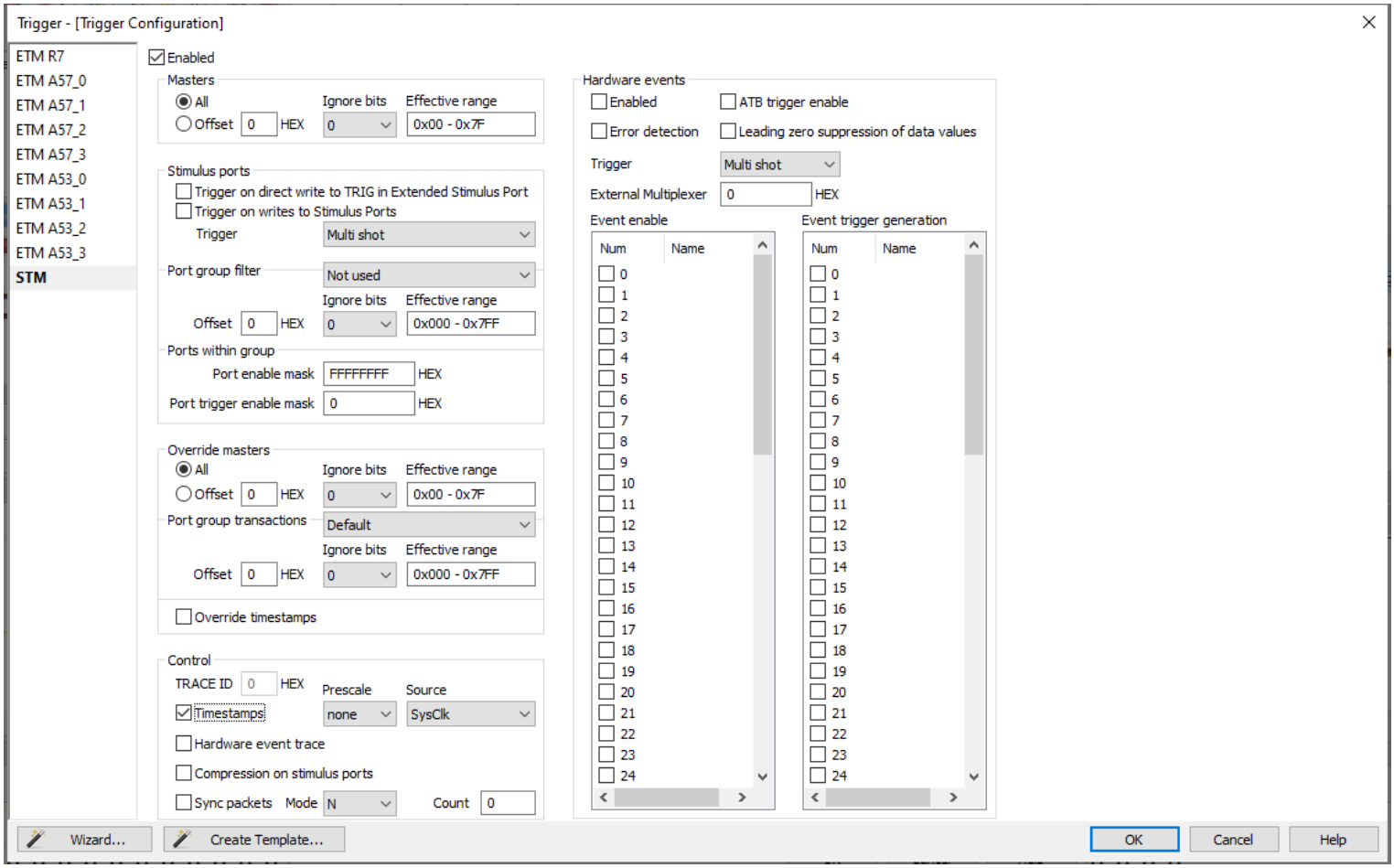
2 In the Trigger configuration menu, open the STM page.

a. Set STM to *Enabled*.

b. Change the *Port enable mask* under *Ports* with group to `FFFFFFFF`.

c. Enable *Timestamps* and set source to *SysClk* for *global timestamps*.

The resulting configuration is shown on the following screen shot. Writes to all STM channels are now recorded.



RH850 Software Trace

Renesas Software trace is an RH850 specific instrumentation-based trace technique. It uses dedicated assembly instruction called DBCP, DBTAG, and DBPUSH to create trace messages at points of interest. You can decide where and with which arguments to call the respective instructions.

- DBCP - Creates a trace message with the current value of the instruction pointer,
- DBTAG - Creates a message with a constant value (known at compile time),
- DBPUSH - Creates signals based on the content of variables (that change during runtime).

This section assumes that the application **already contains software trace assembly instructions**. If this is not the case, refer to the instrumentation trace based sections of this document.

To record software trace messages open winIDEA and the winIDEA Profiler and do the following configuration steps.

1. Select LPD SofTrace under *Hardware / CPU Options / Analyzer / Operation Mode*.
2. Create a new Manual Trace Configuration via *View / Analyzer / Create New Configuration*.

▸ Need help with configuring Manual Trace Configuration?

3. Change Core traced to the core of interest, usually PE1.

Note that Software Trace can only observe one core at a time.

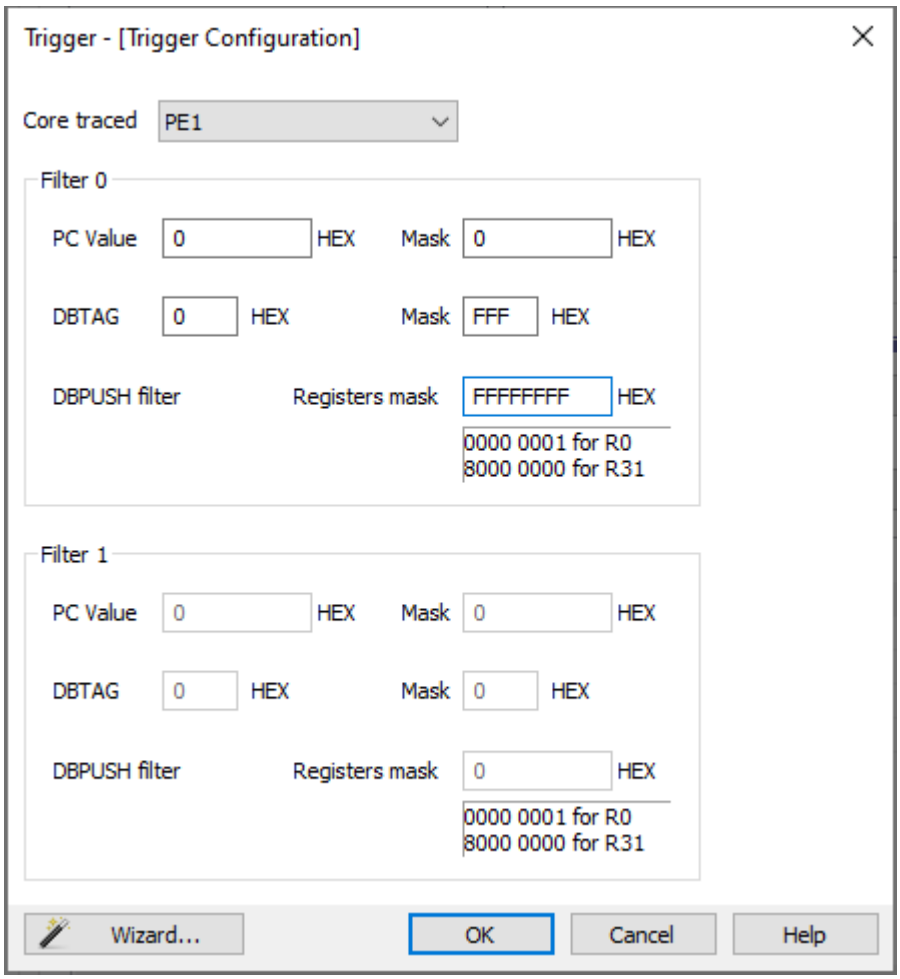
4. To record all DBTAG messages, set value to 0 and Mask to 0xFFF.

All value bits are ignored.

a. Usually, you want to record all values, but using value and filter to limit the amount of trace messages can be helpful in case of overflows.
b. See RH850 SFT Configuration for more information.

5 To record all DBPUSH messages, set the register mask to 0xFFFFFFFF.

The resulting configuration should look as depicted on the following screen shot. The winIDEA Profiler now records Renesas software trace messages. The Profiler interprets the software trace messages based on the information in the Profiler XML file.



NXP/ST Power Architecture

This section explains how to configure data trace for the PowerPC architecture.

1 Select Nexus Trace Port under *Hardware / CPU Options / Analyzer / Operation Mode*.

- PowerPC's On-Chip trace does not provide sufficient buffer sizes for timing analysis.
- If only On-Chip trace is available, an emulation adapter that provides a Nexus Trace Port may be required.

2 Create a new Manual Trace Configuration via *View / Analyzer / Create New Configuration*.

▸ Need help with configuring Manual Trace Configuration?

3 In the Recorder page, disable *Timer Interpolation*.

4 For each CPU on which to record a variable, do the following steps.

a. Navigate to the specific CPU page.
b. Enable trace for that CPU by checking *Enabled*.
c. Under Record, deselect *Program* and select *Data*.
d. Enable a Data Message Controller and specify the name of a variable.
e. Change Access to Data and Control to WR (i.e., trace write accesses only).

The screen shot below shows the correct configuration to record the variable `isystem_trace` from CPU2 (which usually is AUTOSAR core 0).