

# Vector MICROSAR Profiling

Updated: 08/07/2020



This document and all documents accompanying it are copyrighted by iSYSTEM and all rights are reserved. Duplication of these documents is allowed for personal use. For every other case, written consent from iSYSTEM is required.

Copyright © iSYSTEM, AG.  
All rights reserved.  
All trademarks are property of their respective owners.

iSYSTEM is an ISO 9001 certified company

## Table of Contents

1	Introduction .....	2
2	Generic OS Configuration .....	3
2.1	Configure ORTI Support .....	3
2.2	Setup iTCHi .....	3
2.3	Enable OS Timing Hooks .....	4
2.4	Enable VFB Trace Hooks .....	4
3	Use-Cases .....	5
3.1	Running Task/ISR via Data Trace .....	5
3.2	Running Task/ISR via Instrumentation Trace .....	6
3.3	Task State/Running ISR via Data Trace .....	7
3.4	Task State/Running ISR via Instrumented Data Trace .....	11
3.5	Task State/Running ISR via Instrumentation Trace .....	12
3.6	Runnables via Program Flow Trace .....	13
3.7	Runnables via Instrumented Data Trace .....	14
3.8	Runnables via Instrumentation Trace .....	15
4	Generic Profiler/Trace Configuration .....	16
4.1	Configure OS/RTE Profiling .....	16
4.2	Infineon TriCore Data Trace .....	17
4.3	Renesas RH850 Software Trace .....	20
4.4	BTF Export .....	21
5	Technical Support .....	22
5.1	Online Resources .....	22
5.2	Contact .....	22

## 1 Introduction

In this document, we explain 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 profiling objects (e.g., tasks, ISRs and Runnables) and the trace capabilities available on your microcontroller to properly utilize this resource. If you are not familiar with these topics, consider watching our [Introduction to AUTOSAR Classic Profiling](#) webinar and consult our [Introduction to AUTOSAR Classic Profiling](#) application note and then come back to this document.

Once you know the types of objects you want to record and the available trace techniques available on your microcontroller, you can use *Table 1* to jump to the section within this document that explains that use-case. We recommend that you first read the rest of this introduction, then follow the steps in the *Generic OS Configuration* section, and then consult the chapters for your specific use-cases. You do not have to read the complete document. In each section, we also link to the relevant part in our Vector MICROSAR Profiling webinars in case you prefer video over the textual guide. Watching the videos might also help to resolve unexpected issues.

Table 1: Links to the step by step configuration guides for the different profiling use-cases.

	<b>Running Task/ ISR</b>	<b>Task State/ Running ISR</b>	<b>Runnables</b>
<b>Program Flow Trace</b>			<a href="#">Runnables via Program Flow Trace</a>
<b>Data Trace</b>	<a href="#">Running Task/ISR via Data Trace</a>	<a href="#">Task State/Running ISR via Data Trace</a>	
<b>Instrumented Data Trace</b>		<a href="#">Task State/Running ISR via Instrumented Data Trace</a>	<a href="#">Runnables via Instrumented Data Trace</a>
<b>Instrumentation Trace</b>	<a href="#">Running Task/ISR via Instrumentation Trace</a>	<a href="#">Task State/Running ISR via Instrumentation Trace</a>	<a href="#">Runnables via Instrumentation Trace</a>





Each section follows the same steps. First, you configure the OS or RTE to make the information about the trace objects available for profiling in winIDEA. For instrumentation-based use-cases, you must generate the instrumentation code and recompile the application. Next, you make winIDEA aware of the different profiling objects by creating an iSYSTEM Profiler XML with iTCHi (iSYSTEM Trace Configuration Helper iTCHi). Finally, you configure the hardware trace on the microcontroller to record the OS and RTE objects for the profiling. Depending on the use-case, winIDEA might be able to do this step automatically, though manual configuration leads to a better understanding of the underlying trace logic on the silicon.

## 2 Generic OS Configuration

This chapter explains the configuration steps shared by different use-cases. *Configure ORTI Support* and *Setup iTCHI* are required. The other sub-sections are only relevant for particular use-cases.

### 2.1 Configure ORTI Support

An OSEK Runtime Interface (ORTI) file is mandatory for all Task and ISR based use-cases. To enable ORTI support, do the following steps:

1. In the DaVinci Configurator, open the Basic Editor.  
 [Basic Editor](#)
2. In the Basic Editor, open the OS node.  
 Os
3. Navigate to the *OsOS* node and select the *OsDebug* node.  
 OsOS  
 OsDebug
4. Activate ORTI Debug Support by selecting ORTI\_23\_STANDARD or ORTI\_23\_ADDITIONAL.

After these steps regenerate the OS and you should now have an `Os_Trace.ORT` file in your generated data directory `App1\GenData`.

### 2.2 Setup iTCHI

The iSYSTEM Trace Configuration Helper iTCHI helps the user by automatically generating the iSYSTEM Profiler XML file and instrumentation code. *Figure 1* shows the input and output files for the different iTCHI commands. The ORTI file is a necessary input file, and iTCHI always generates the Profiler XML as an output file. The other fields are use-case specific.

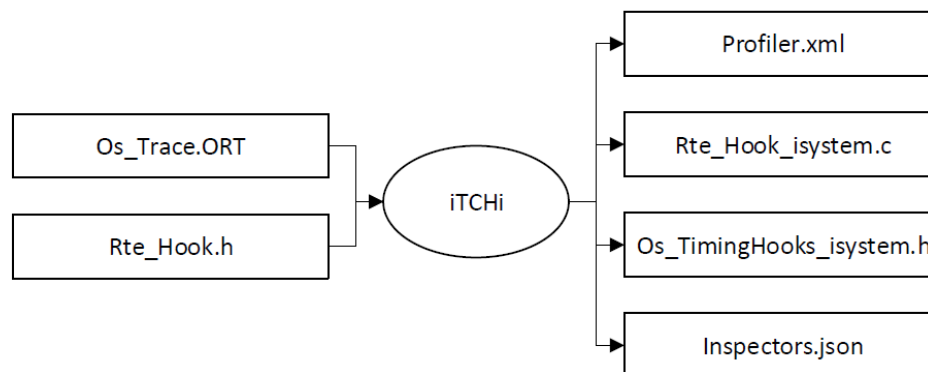





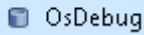


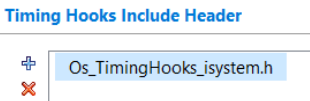
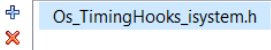

Figure 1: iTCHI helps the user to generate an iSYSTEM Profiler XML file and instrumentation code.

To start using iTCHI, navigate to the `scripts/itchi` directory within your winIDEA installation. In case that directory is not available, download a newer winIDEA version or ask the iSYSTEM Support to provide iTCHI to you. Once you have iTCHI available, the folder contains the iTCHI executable `itchi-bin.exe` and the documentation `readme.html`. Open a command window (terminal) in that directory and run `itchi-bin.exe --help`. This command should output the available iTCHI commands, including a short explanation. Next, generate an empty iTCHI configuration file by running `itchi-bin.exe --write_default_config`. This command creates an empty `itchi.json` in the current directory. It includes empty attributes for the different use-cases. Start by pointing the ORTI file attribute to your ORTI file and specify a Profiler XML file, for example, `profiler.xml`. Keep in mind that iTCHI resolves relative paths relative to the JSON configuration file.

## 2.3 Enable OS Timing Hooks









Timing-Hooks are only available in the following versions of MICROSAR. In generation 6, Timing-Hooks are available in versions 9.01.00 and higher. In gen 7, Timing-Hooks are available in versions 01.07.00 and higher. The version numbers are part of the OS source headers.

Follow these steps to enable the Vector OS Timing-Hooks. The chapter about your specific use-case covers how to implement the hook file.

1. In the DaVinci Configurator, open the Basic Editor.  
 [Basic Editor](#)
2. In the Basic Editor, open the OS node.  
 Os
3. Navigate to the *OsOS* node and select the *OsDebug* node.  
 OsOS  
 OsDebug
4. Locate the Timing Hooks Include Header section and click the add symbol.  
 Timing Hooks Include Header 
5. Double click the new box and call it *Os\_TimingHooks\_ismethod.h*.  
 Timing Hooks Include Header  
 Os\_TimingHooks\_ismethod.h
6. Regenerate the OS .

## 2.4 Enable VFB Trace Hooks

To record Runnables via instrumentation, you must enable the so-called Virtual Function Bus (VFB) trace hooks in DaVinci Configurator. These hooks allow instrumentation of RTE related events such as Runnable starts and returns. The chapter about your specific use-case covers how to implement the hooks with iTCHI.

1. Open *Runtime System General* in the *Runtime System* node.  
 [Runtime System General](#)
2. Expand the RTE tab and select VFB Tracing.  
 RTE  VFB Tracing
3. Enable VFB Tracing.  
 Enable VFB Tracing:  
4. Start the Import *VFB Functions Assistant*.  
[Import VFB Trace Functions Assistant](#)
5. On the first page of the wizard, select the *RTE hook file* to keep the default configuration.
6. On the next page, change the *trace functions to be imported* from *All* to *Selected*.  
 All  
 Selected
7. Select the start and return hook for each Runnable you want to profile:
  - a. `Rte_Runnable_<SWC>_<Runnable>_[Return|Start]`
  - b. For example, if you want to profile the Runnable with the name `Core2_Runnable_5ms` select the respective start and return hooks:  
 Rte\_Runnable\_SWC\_Core2\_SWC\_Core2\_Runnable\_5ms\_Return  
 Rte\_Runnable\_SWC\_Core2\_SWC\_Core2\_Runnable\_5ms\_Start
  - c.  Rte\_Runnable\_SWC\_Core2\_SWC\_Core2\_Runnable\_5ms\_Start
8. Finish the wizard. The selected hooks should now appear in the VFB Trace Functions window.
9. Regenerate the OS and the RTE .
10. You can find the enabled hooks in `Rte_Hook.h`. iTCHI uses this file to implement the hooks.

### 3 Use-Cases

This chapter explains the different OS and RTE profiling use-cases. Refer to *Table 1* to find which options work for your setup. You can combine multiple use-cases with the same iSYSTEM Profiler XML file. For example, it is common to combine data-trace based Task State/Running ISR with instrumentation-based Runnable Profiling. Simply configure the different use-cases one after another using the same iTCHI configuration and Profiler XML file.

#### 3.1 Running Task/ISR via Data Trace

This section explains how to profile the running Task/ISR via data-trace. We also cover this use-case in our [MICROSAR Profiling Webinar](#). We do not use the iSYSTEM Profiler XML in the webinar, but it is the preferred approach. Running Task/ISR Profiling relies on the two ORTI attributes *RUNNINGTASK* and *RUNNINGISR*. Each attribute points to a global variable that winIDEA uses to profile the respective object. The listing below shows that section from the ORTI file. For a multi-core application, there is one attribute pair for each core.

```
RUNNINGTASK = "OsCfg_Trace_OsCore_Core0_Dyn.CurrentTask";
RUNNINGISR2 = "OsCfg_Trace_OsCore_Core0_Dyn.CurrentIsr";
```

Listing 1: ORTI Attributes for single core application.

Start by referencing the ORTI and the Profiler XML file from the iTCHI configuration file. Your configuration file should have the contents shown in the following listing.

```
{
  "orti_file": "Os_Trace.ORTI",
  "profiler_xml_file": "Profiler.xml"
}
```

Listing 2: iTCHI configuration for Running Task/ISR Profiling.

You are now ready to create the Profiler XML file by running `itchi-bin.exe --running_taskisr`. Once you have this file available, you can follow the *Configure OS/RTE Profiling* section.

For single-core applications, you can start recording right away, because winIDEA can do the trace configuration for you. However, for multi-core microcontrollers, winIDEA might not be able to configure the trace automatically. In this case, you must set the trace hardware on the microcontroller to record all necessary variables. You can find the required variables in the Profiler configuration menu under OS Setup, as shown in *Figure 2*. For each ORTI object, winIDEA shows the respective global variable in the Object Info under *Signaling*.

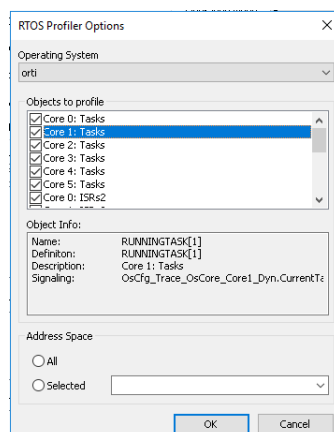


Figure 2: The RTOS Profiler Options menu shows the Running Task and Running ISR objects. For multi-core applications, you must manually configure the data-trace for each of the cores.

Once the application is running and you start a recording, you should see Tasks and ISRs, as shown in *Figure 3*. If you do not see any data, check the trace window for write accesses to the global variables.

In case you cannot see any write events try to configure the data-trace manually. Also, make sure the data section in the profiler timeline is visible and zoomed out far enough.

Two potential errors can occur when you start the recording. First, winIDEA might complain about too many data-areas. This error means winIDEA cannot figure out how to configure the data-trace automatically, and you have to set the hardware trace manually. Second, winIDEA might complain about a missing default IRQ. To resolve this error, open the Profiler XML and rename INVALID\_ISR to NO\_ISR. For a more permanent solution, specify the default\_isr2 attribute in the iTCHi configuration file and re-run iTCHi.

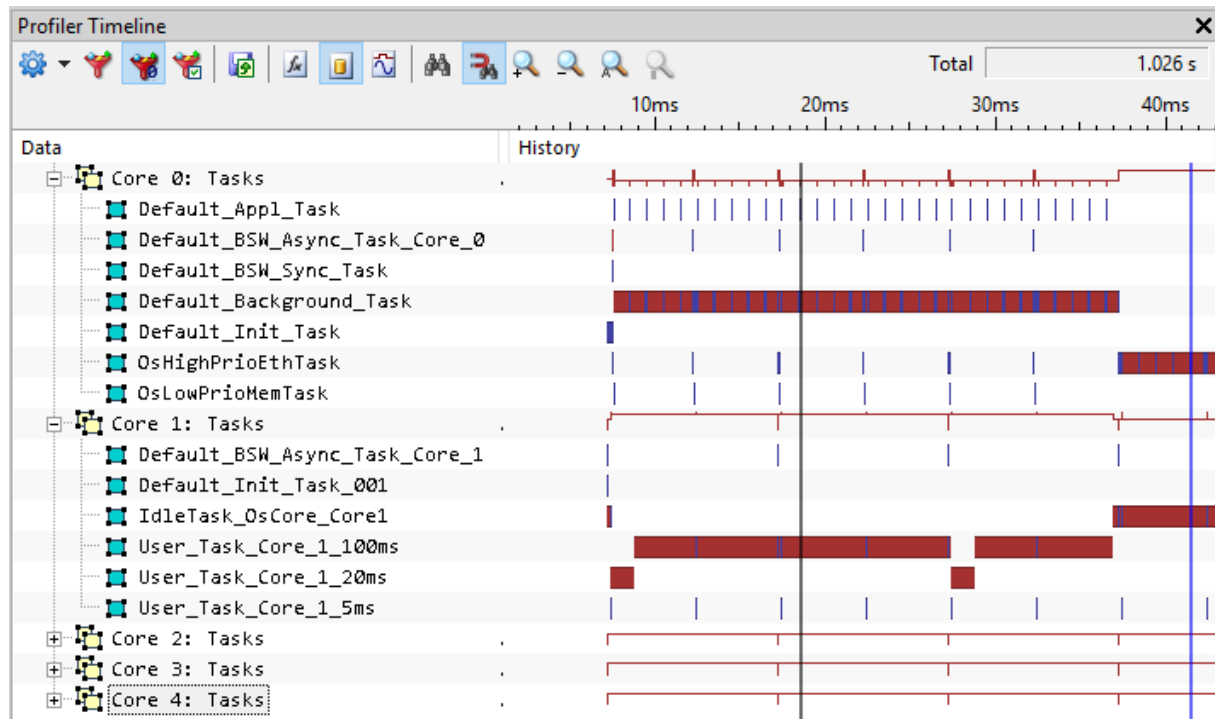


Figure 3: ORTI running Task/ISR trace in the winIDEA Profiler Timeline.

### 3.2 Running Task/ISR via Instrumentation Trace

Some microcontrollers do not provide data, but instrumentation trace. Instrumentation trace means adding instrumentation to the code that generates hardware trace messages. On RH850 based microcontrollers, Renesas call this feature Software Trace. On PowerPC based microcontrollers, it is called Ownership Trace Messages (OTM).

Here, we cover how to record a running Task/ISR trace via Renesas RH850 software trace because it is the use-case most commonly utilized by our customers. Note that this use-case only works for single-core applications. For multi-core applications, please follow *Task State/Running ISR via Instrumented Data Trace* section.

Start by enabling the operating system timing-hooks, as explained in *Enable OS Timing Hooks*. Next, we need to implement the instrumentation for the hooks:

1. Navigate into the `App\GenData` directory and create a new file `Os_TimingHooks_isystem.h`.
2. Copy the content from *Listing 3* into the file and save it.
3. Build the application by running `.\m.bat depend` and `.\m.bat` in the build directory.

The instrumentation code generates a software trace message with the identifier of the newly running thread whenever it changes. Note that thread is an umbrella term for both tasks and ISRs. Consequently, winIDEA can profile both objects by utilizing a single instrumentation hook.

```
asm void isystem_profile_thread(val)
{
%reg val
    mov val, r10
    dbpush r10-r10
}

#define OS_VTH_SCHEDULE(FromThreadId, FromThreadReason, ToThreadId, ToThreadReason,
CallerCoreId) \
{ \
    isystem_profile_thread(ToThreadId); \
}
```

Listing 3: Code for *isystem\_hooks.h* to profile the currently running thread via Renesas software trace.

To make winIDEA aware of the instrumentation, you must generate a Profiler XML. For this specific use-case, manual editing of the Profiler XML is necessary. First, create an iTCHI configuration file with the following content.

```
{
  "orti_file": "Os_Trace.ORT",
  "profiler_xml_file": "Profiler.xml"
}
```

Then run iTCHI by executing `itchi-bin.exe --task_state_instrumentation` in the terminal. Manually edit the newly generated Profiler XML file. Scroll down to the Profiler section and add a new object, as shown in *Listing 3*.

```
<Object>
  <Type>TypeEnum_ThreadMapping</Type>
  <Name>Threads_Instrumentation</Name>
  <Definition>Threads_Instrumentation_Definition</Definition>
  <Description>Threads</Description>
  <Signaling>DBPUSH(10)</Signaling>
  <DefaultValue>NO_THREAD</DefaultValue>
  <Level>Tasks</Level>
</Object>
```

Listing 3: The profiler object to record threads via Renesas RH850 software trace.

When you re-run iTCHI, for example, after you have added new tasks to the OS, iTCHI leaves this object alone, i.e., you only must make this manual change once. You can now add the Profiler XML file to winIDEA, as explained in the *Configure OS/RTE Profiling* section. Make sure you select the *Threads\_Instrumentation* object in the OS Setup menu. Usually, winIDEA can configure software trace automatically, but if you do not see any data, you can follow the *Renesas RH850 Software Trace* section. Congratulations, you now have a Running Thread trace that includes tasks and ISRs.

### 3.3 Task State/Running ISR via Data Trace

Running Task profiling provides no information about the reason for a task context switch. With Task State Profiling, you get the additional information about why a switch has occurred, for example, because of preemption by a higher priority task. This section explains how to profile Task State/Running ISR information with Data Tracing.

Depending on the MICROSAR version you are using, there are two different approaches for how to set up this use-case. Older MICROSAR OS versions have a so-called task state array. That is an array in which each field represents the state of a task. Newer MICROSAR versions have a so-called complex expression for the task state. That means the state information for each task depends on multiple variables. Search for TASK objects in your ORTI variable and examine the STATE attribute to find out which approach is the right one for your application.



### 3.3.1 Task State Simple Variable

If you have an older MICROSAR OS version, your Task objects look as shown in *Listing 4*.

```
TASK osSystemExtendedTask {
    PRIORITY = "osTcbActualPrio[0]";
    vs_HomePriority = "0";
    STATE = "osTcbTaskState[0]";
    /* other attributes here */
}; /* osSystemExtendedTask*/
```

Listing 4: Task object ORTI definition with a simple STATE variable.

For this use-case, you need a minimal iTCHi configuration file.

```
{
    "orti_file": "Os_Trace.ORT",
    "profiler_xml_file": "Profiler.xml"
}
```

You can then generate the Profiler XML file by running `itchi-bin.exe --task_state_single_variable`. Add the Profiler XML to winIDEA and configure the data-trace to record the complete `osTcbTaskState`-Array. You can now start profiling.

Note that there is no state model for ISRs. We still profile ISRs via the running ISR variable. In some older versions of MICROSAR, the ISR variable contains a pointer, as shown in the following listing.

```
RUNNINGISR2 = "osConfigBlock.CcbAddress->LockIsNotNeeded.ossActiveISRID";
```

If this is the case, winIDEA might complain once you start profiling. Usually, there exists a non-pointer symbol that maps to the same address. To find that variable, enter the pointer into the watch window in winIDEA.

```
☐ osConfigBlock.CcbAddress | (Ptr (0x70010C44) = Ptr (osCtrlVarsCore0)
```

You can see that the pointer references the regular symbol `osCtrlVarsCore0`. Update your iTCHi configuration so that iTCHi automatically writes the correct symbol into the Profiler XML file, as shown in the following listing.

```
{
    "orti_file": "Os_Trace.ORT",
    "profiler_xml_file": "Profiler.xml",
    "running_taskisr": {
        "search_replace_list": [
            ["osConfigBlock.CcbAddress->", "osCtrlVarsCore0"]
        ]
    }
}
```

Do not forget to re-run iTCHi with the same flag as before. The Running ISR variable in the Profiler XML file does not contain a pointer anymore. You can profile the Task State and the Running ISR without further issues.

### 3.3.2 Task State Complex Expression

In newer MICROSAR versions, task states have sophisticated STATE attributes consisting of multiple symbols, as shown in *Listing 5*. The winIDEA Analyzer can profile these expressions with the help of [winIDEA Inspectors](#). This use-case is non-trivial, so we recommend watching the [webinar](#) in addition to reading this section.

```
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 */
```

Listing 5: Task object ORTI definition with a complex STATE variable.

First, create an iTCHI configuration file, as shown in *Listing 6*.

```
{
  "orti_file": "Os_Trace.ORT",
  "profiler_xml_file": "Profiler.xml",
  "task_state": {
    "task_to_core_heuristic": true
  },
  "task_state_inspectors": {
    "inspectors_file": "Inspectors.json",
    "constant_variables": {
      "OsCfg_Core_OsCore_Core0_Status_Dyn.OsState": 2,
      "OsCfg_Core_OsCore_Core1_Status_Dyn.OsState": 2
    },
    "parent_area_template": "Data/Core {core_id}: Tasks/{task_name}",
    "default_state": "UNKNOWN"
  }
}
```

Listing 6: iTCHI configuration for task state profiling with complex state variables.

You are familiar with the ORTI and Profiler XML attributes. In addition to them, there are two new objects: *task\_state* and *task\_state\_inspectors*. We set *task\_to\_core\_heuristic* to *true*. That enables iTCHI to infer the task to core mapping from the ORTI file automatically.

The other object, *task\_state\_inspectors*, helps iTCHI to generate a working inspectors file. The profiler XML automatically references this file allowing the profiler to reconstruct the task states.

The only field you update here is *constant\_variables*. If you look back to *Listing 5*, you see that the first part of the expression checks whether the OS application of the task is running (2 maps to the Running state for OS applications), as shown in the following listing.

```
OsCfg_Core_OsCore_Core0_Status_Dyn.OsState == 2
```

The operating system only writes to this variable once at the startup of the application. If we start tracing at a later point, the profiler will never know the current value of the variable. To avoid this problem, we tell iTCHI that it can ignore this variable by setting it to 2 manually, as shown in *Listing 6*. Depending on how many cores your application uses, this variable may have a different name, or there might be multiple variables (one for each OS application). You have to extend the list so that each of them gets a constant value of 2. Make sure to not add a comma for the last pair in the list.

You can now generate the Profiler XML and the Inspectors JSON file by running `itchi-bin.exe --task_state_complex_expression`, and add the Profiler XML file to winIDEA as explained in *Configure OS/RTE Profiling*.

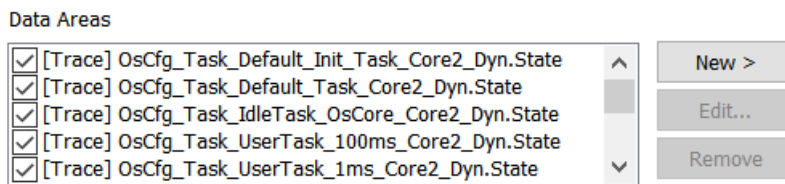
Next, we have to record all the other variables that are part of the state expression. There is one running Task variable for each core that is part of the expression:

```
OsCfg_Trace_OsCore_Core0_Dyn.CurrentTask
9 of 22
```

Additionally, there is a state variable that is part of a structure for each task. The linker allocates the structs to consecutive memory areas. Therefore, the most efficient way to trace the state variables is to configure the data-trace so that it records all accesses to the full list of structs. The configuration depends on the microcontroller in use.

```
OsCfg_Task_Default_Appl_Init_Task_Dyn.State
```

Once the trace contains the write accesses to the state variables, the profiler must display them. The Inspectors can then reference the values to reconstruct the task states. To show the state variables add them in the winIDEA profiler data tab under Data Areas. Also, make sure to enable data profiling. Add the state variables without braces, as shown in the following screenshot.



Since the Analyzer automatically includes the Inspectors file, you can now start a recording, as shown in Figure 4. Note the task-states (indicated by the filter icons) and the data areas for the states. The first thing to check if the inspectors do not change is if the respective data areas are part of the profiler timeline. Also, read the output window to see if there are any issues.

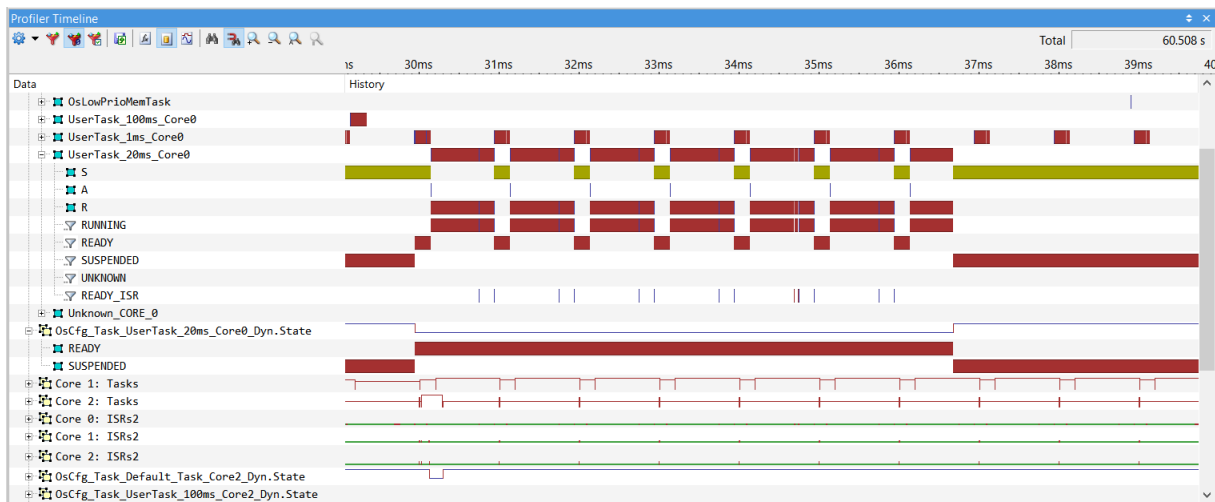


Figure 4: Task state profiling via data tracing for complex expressions. Inspectors (filter icon) reconstruct the state from multiple variables.

### 3.4 Task State/Running ISR via Instrumented Data Trace

This section describes how to record a Task/ISRs State Trace by using the Vector Timing-Hooks. Timing-Hooks are OS macros which, if implemented, execute at points of interest in the scheduling routines of the OS. For more information about Timing-Hooks, refer to the Vector OS reference manual.

Vector MICROSAR OS versions that support the timing-hooks no longer explicitly differentiate between Tasks and ISRs. The umbrella-term thread refers to both types of objects and the same hooks signal information for Tasks and ISRs.

This section explains how to do Task State/Running ISR profiling by instrumenting the Vector OS Timing-Hooks. A video guide for this use-case is also available as a [webinar](#).

Start by enabling the Timing-Hooks as explained in *Enable OS Timing Hooks*. Next, generate the Profiler XML and the implementation of the hooks with iTCHi. Use the following listing as a starting point for your iTCHi configuration file.

```
{
  "orti_file": Os_Trace.ORT",
  "profiler_xml_file": "Profiler.xml",
  "task_state_instrumentation": {
    "template_directory": "./"
  }
}
```

Listing 7: iTCHi configuration for Task State/Running ISR profiling via instrumented data-trace.

You can see that the configuration only includes an additional attribute to indicate the directory to which iTCHi should save the instrumentation files. You can either generate the code into the current directory or directly provide the path to your Vector MICROSAR *GenData* directory.

With the configuration file in place, you can now run `.\itchi-bin.exe --task_state_instrumentation`. This command generates the Profiler XML file as well as two instrumentation files: `Os_TimingHooks_isystem.h` and `Os_TimingHooks_isystem.c`.

The header file should have the same name as the hook implementation file specified in DaVinci Configurator. You can examine the header file to verify that it implements three OS timing-hooks. Each hook includes the core information, the new state of a thread, and the thread identifier. The hook implementation merges this information into a single value and writes it into the global variable `isystem_trace`.

The first hook `OS_VTH_SCHEDULE` signals a thread context change; the currently running thread is preempted, terminated, or goes into the waiting state, and a new thread starts executing. This hook requires two writes, to signal the latest state of the old thread, as well as the thread which is running next. The hook `OS_VTH_ACTIVATION` indicates when a new thread is activated. Finally, `OS_VTH_SETEVENT` means that the OS sets an event for a specific thread. If the state of that respective thread changes, this indicates a state change from the waiting into the ready state.

The instrumentation derives state identifiers from the existing definitions for the *FromThreadReason* and *ToThreadReason* arguments. The OS header file `OsInt.h` contains these defines.

The only purpose of the C-file is to define the global `iSYSTEM` trace variable. You can adapt the definition for your processor architecture. By default, we use the `at` preprocessor-macro to map the variable into global uncached LMU-RAM on Infineon TriCore microcontrollers.

After you have adapted the instrumentation, rebuild the application with the new hook files.

1. Copy the files into the `Appl\GenData` directory.
2. Add `Os_TimingHooks_isystem.c` to one of the makefiles:  
`APP_SOURCE_LST += GenData\Os_TimingHooks_isystem.c`
3. Start a shell in the build directory and execute `.\m.bat`.

You have finished instrumenting the application. Add the Profiler XML file to winIDEA and configure a data-trace for the global trace variable `isystem_trace`. You can now start to profile your application. The result is a Vector MICROSAR Thread state recording, as depicted in *Figure 5*.

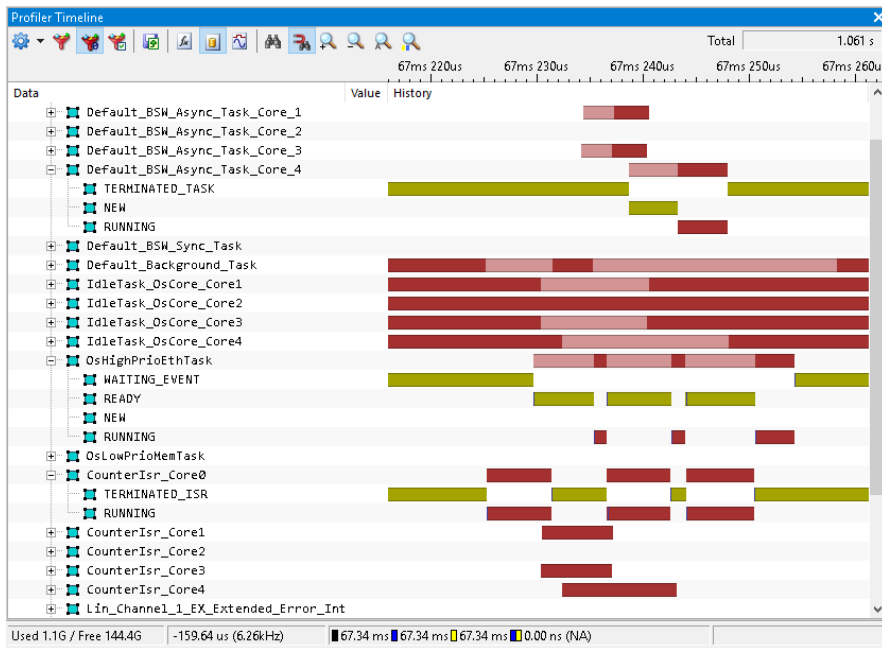


Figure 5: MICROSAR OS Thread State profiling recorded by instrumenting the Vector Timing-Hooks. Notice the initial pending time (NEW) and the different inactive states: READY, WAITING and TERMINATED.

### 3.5 Task State/Running ISR via Instrumentation Trace

Instead of instrumenting the Vector OS Timing-Hooks with writes to a global variable, we can also utilize instrumentation trace. For a visual guide to this use-case, watch this [webinar](#). Follow the previous section Task State/Running ISR via Instrumented Data Trace and come back to this section before you compile the application with the instrumentation file.

You first have to update the instrumentation to use software trace messages. Open `Os_TimingHooks_isystem.h` and scroll to the bottom. You will find a commented out section that includes implementations for the timing-hooks and an assembly function `isystem_profile_thread`. Remove the comments to enable this part of the instrumentation file and remove the other hook implementations in the upper part (the once writing into `isystem_trace`). You can now copy the header file into your `GenData` directory and rebuild the application. You do not have to include the C file because this use-case does not use the global instrumentation variable.

Next, you have to update the Profiler XML to let the profiler know that the instrumentation uses RH850 software trace instead of the instrumentation variable. Search for the `Threads_Definition` object. Then remove the red line with the trace variable and add a new line with the `DBPUSH` string, as shown in the following listing. Be aware, that `iTChI` will override the Profiler XML file if you re-execute it. To avoid this, you can copy the whole object and append `SFT` to the Name and Definition texts, as indicated by the blue color.

```
<Object>
  <Definition>Threads_Definition_SFT</Definition>
  <Name>Threads_Name_SFT</Name>
  <!--Lines removed to save space. -->
  <Expression>isystem_trace</Expression>
  <Signaling>DBPUSH(10)</Signaling>
  <!--Lines removed to save space. -->
</Object>
```

You can now load the update Profiler XML file into winIDEA and start profiling. Usually, winIDEA can configure software trace automatically, but if you do not see any data, you can follow the *Renesas RH850 Software Trace* section.

### 3.6 Runnables via Program Flow Trace

Runnables are functions defined within the RTE. The operating system does not actively manage the execution of Runnables directly but runs Tasks, which then execute the Runnables. There are no variables that indicate the current state of a Runnable. Tracing Runnables via data-trace is therefore not feasible. However, you can profile Runnables without instrumentation via program-flow-trace. For a video guide of this use-case, watch this [webinar](#).

It is possible to record a program-flow-trace and analyze the Runnables within the code-area of the profiler. Explicitly, marking the functions of Runnables has two advantages: the profiler shows the Runnables under a dedicated node in the data section, as shown in *Figure 6*, and you can export the Runnables into a BTF trace.

To profile Runnables, you have to add them to the iTCHI configuration file into a dedicated section, as shown in *Listing 8*. Next, execute `./itchi-bin.exe --runnable_program_flow`. After generating the Profiler XML load, it into winIDEA, configure program-flow-trace for your target and make sure to select Runnables under the profiler OS setup. You can start profiling and should get a result similar to the recording depicted in *Figure 6*.

```
{
  "orti_file": "Os_Trace.ORT",
  "profiler_xml_file": "Profiler.xml",
  "runnable_program_flow": {
    "runnables": [
      "Runnable_Core1_100ms",
      "Runnable_Core1_1ms"
    ]
  }
}
```

Listing 8: iTCHI configuration for profiling Runnables with program-flow-trace.

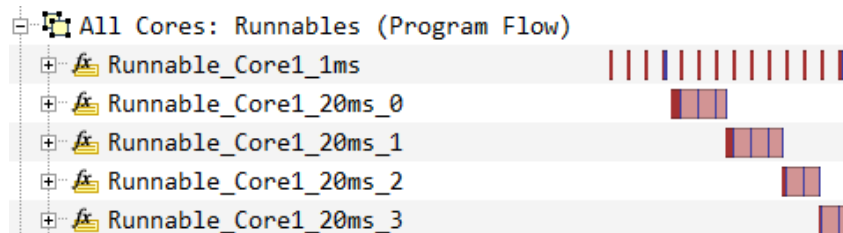


Figure 6: Runnable profiling based on program-flow-trace. The profiler displays Runnables in the data area. Make sure to unselect "hide areas with no activity."

### 3.7 Runnables via Instrumented Data Trace

Tracing of Runnables can be accomplished via program flow trace or utilizing instrumentation. This section describes how to profile Runnables by instrumenting the Virtual Function Bus (VFB) trace hooks. VFB tracing allows tracing the execution of various AUTOSAR RTE objects. The user can decide which events (e.g., Runnable start/return or Data send/receive) to observe. For more information, refer to the RTE technical reference manual. For a video guide of this use-case, watch our [webinar](#). Start by enabling the VFB Runnable hooks for the Runnable you want to profile, as explained in *Enable VFB Trace Hooks*. After you have regenerated the RTE, iTCHi can automatically implement the instrumentation for the hooks. Adapt the configuration in the following listing.

```
{
  "orti_file": "Os_Trace.ORT",
  "profiler_xml_file": "Profiler.xml",
  "runnable_instrumentation": {
    "isystem_vfb_hooks_c": "Rte_VfbHooks_isystem.c",
    "rte_hook_h": "Appl/GenData/Rte_Hook.h",
    "regex": "(FUNC\\(void, RTE_APPPL_CODE\\)
Rte_Runnable_(\\w+)_ (Start|Return)\\([^\n]+\))",
    "trace_variable": "isystem_trace_runnable",
    "template_file": "VfbHooks_template.c"
  }
}
```

Listing 9: iTCHi configuration for profiling Runnables with instrumented data-trace.

There are a couple of attributes for the Runnable instrumentation use-case. You can leave the attributes `regex`, `template_file`, and `trace_variable` as they are. Use `isystem_vfb_hooks_c` to specify the file into which iTCHi generates the instrumentation and `rte_hook_h` to point iTCHi to the `Rte_Hook.h` file generated by the DaVinci Configurator.

You are now ready to run `itchi-bin.exe --runnable_instrumentation`. Open the newly generated instrumentation file and verify that the Runnable hooks are there. You will notice that the generated code relies on some microcontroller and compiler-specific features. The default instrumentation uses the `mfcrr` (move from core register) instruction to get the core identifier. Also, we allocate the `iSYSTEM` Runnable variable into global LMU RAM by using the `at` directive. This hook implementation works for Infineon AURIX microcontrollers.

For other architectures, open the hook template file (`VfbHooks_template.c`) and remove the two red sections, as shown in the following listing. Delete the memory allocation from the definition of the trace variable and remove the core identifier instruction from the hook implementation template. If you are using a multi-core microcontroller, you must replace the code with another command to get the core identifier depending on the architecture you are using.

```
// listing does not show the full template.
volatile uint32 {{trace_variable}} __at(0xB0040200) = 0;

{% for hook in runnable_hooks %}
{{hook.declaration}}
{
  {{trace_variable}} = {{hook.id}} | (__mfcrr(CPU_CORE_ID) << 24);
}
{% endfor %}
```

Listing 10: The default Runnable hook template contains two Infineon AURIX specific sections. Remove the red parts if you are using a different architecture.

You can now regenerate the hook implementation with the command from above. Next, add the hook implementation file (not the template) to your MICROSAR build process and rebuild the application. Add the Profiler XML to winIDEA, select Runnables under OS setup, and record your first Runnable profiling. The result should look as shown in *Figure 7*.

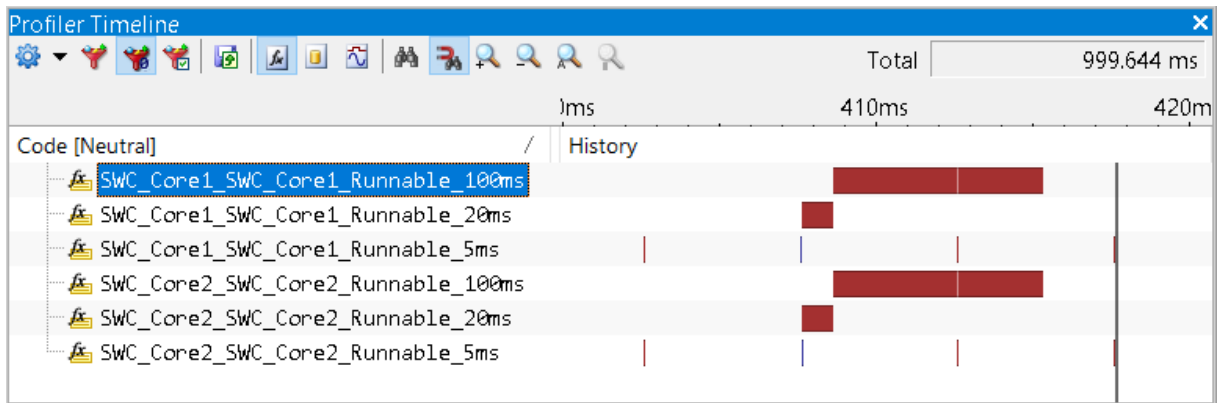


Figure 7: MICROSAR OS Runnable Profiling via VFB Tracing Hooks instrumentation.

### 3.8 Runnables via Instrumentation Trace

Instead of instrumenting the RTE VFB trace hooks with writes to a global variable, we can also utilize instrumentation trace. Follow the previous *section 3.7* and come back to this section before you compile the application with the instrumentation file. You have to make two changes before you can Profile the Runnables via software trace. First, update the hook template file with the software trace instrumentation, regenerate the hook file, and compile the application. Second, update the Profiler XML file to use software trace signaling instead of the global trace variable.

For the first step, open the hook template file (*VfbHooks\_template.c*) and make the changes as shown in the following listing. We remove the trace variable because it is not necessary for software trace. Then, we add an assembly function that uses a DBTAG instruction to send a trace message for a constant value. Finally, we use the assembly function in the hook implementation to signal the Runnable hook identifier. You can now recompile the application with the updated file.

```
// listing does not show the full template.
volatile uint32 {{trace_variable}} __at(0xB0040200) = 0;

# ifndef CPU_CORE_ID
# define CPU_CORE_ID 0xFE1C
# endif

asm void isystem_sft_dbtag(value)
{
  %con value
  dbtag value
}

# if (RTE_VFB_TRACE == 1)
# define RTE_START_SEC_APPL_CODE
# include "MemMap.h" /* PRQA S 5087 */ /* MD_MSR_19.1 */

{% for hook in runnable_hooks %}
{{hook.declaration}}
{
  {{trace_variable}} = {{hook.id}} | (__mfc(CPU_CORE_ID) << 24);
  isystem_sft_dbtag({{hook.id}});
}
{% endfor %}

# endif
```

Now update the Profiler XML file by searching for the Runnable object. Replace the expression for the trace variable with signaling via DBTAG. Reload the Profiler XML into winIDEA, select Runnables under OS setup, and start profiling. The result should look as shown in *Figure 7*.

```
<Expression>isystem_trace_runnable</Expression>
<Signaling>DBTAG</Signaling>
```



## 4 Generic Profiler/Trace Configuration

This chapter shows how to configure OS/RTE awareness using the iSYSTEM Profiler XML file. We also cover trace configurations for standard architectures.

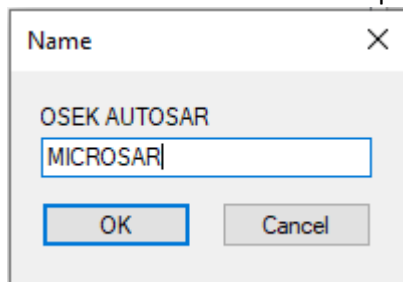
### 4.1 Configure OS/RTE Profiling

This section explains how to add the iSYSTEM Profiler XML generated by iTCHi to winIDEA, and then how to make the winIDEA Analyzer aware of the OS/RTE via the Profiler configuration menu.

1. Add the XML file to winIDEA.
  - a. Go to Debug, Operating Systems:



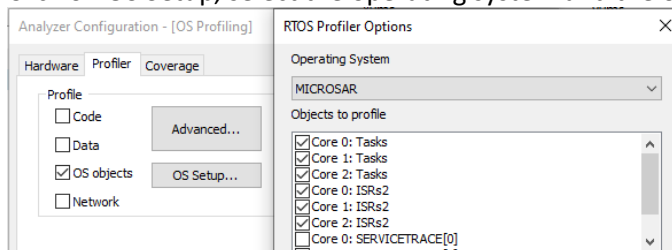
- b. Create a new OSEK AUTOSAR operating system and call it MICROSAR:



- c. Select iSYSTEM XML as file description type and reference your `profiler.xml` file:

Property	Value
Configuration	
RTOS description file type	iSYSTEM XML
RTOS description file location	profiler.xml

- d. Close the menu and Load Symbols or Download to apply the changes:
2. Enable OS/RTE Profiling in the winIDEA Analyzer.
  - a. Go to View, Analyzer, to start the winIDEA Analyzer.
  - b. Create a new Analyzer configuration:
    - c. In the menu, select profiler, unselect Coverage and choose Automatic.
    - d. Open the new configuration via the hammer-icon:
    - e. Make sure profiler is active in the hardware tab:  Profiler
    - f. Switch into the profiler tab, unselect all options except OS objects:  OS objects
    - g. Click on OS Setup, select the operating system and the objects you want to profile:



3. You are now ready to start profiling by clicking the green play button in the Analyzer. If you have multiple objects, winIDEA might give an error saying there are too many data areas. When you get this error, you have to configure the hardware trace manually under the Hardware-tab. You can find out how to do that for different architectures in the following sections.

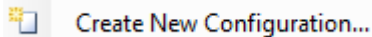
## 4.2 Infineon TriCore Data Trace

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.

### 4.2.1 Basic Configuration

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

1. Create a new winIDEA analyzer configuration.



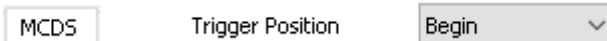
2. Specify a name, select the Profiler checkbox, and choose Manual Trigger Configuration. Confirm with OK.



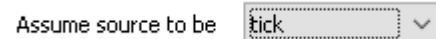
3. In the record-tab, make sure to disable Timer Interpolation.



4. Change to the MCDS Tab and set the Trigger Position to Begin.



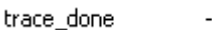
5. Change the Timestamps source to tick.



6. Change to the MCX tab.



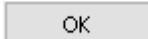
7. Set trace\_done to Never.



8. Set tick\_enable to Always.



9. Click OK to make the configuration permanent.



### 4.2.2 Data Trace Single Variables

This section assumes you have a basic TriCore configuration. Based on that, this section shows how to add data-trace triggers for certain variables.

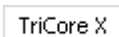
1. Open your analyzer configuration and select Configure.



2. In the MCDS tab, configure POB X so that it observes CPU0 (assuming that core accesses this variable).



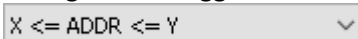
3. Switch to the TriCore X tab.



4. Configuring trace for a specific variable or memory area is a three-step process. First, select the variable with the dtu\_ea\_trig\_\* data trigger. Double-click the DTU (data-trace unit) trigger with the index 0.



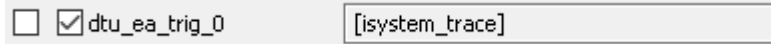
5. Configure the trigger to use work as a range comparator.



6. Select the variable you want to trace and tick the checkbox for the Entire Object. This setting ensures recording of the complete variable, independent of it being a primary type or a complex data type like an array or a struct.



7. Next, you must find an Event that maps to the trigger, enable it, and select the respective trigger from the list.



8. When you close the event configuration dialog, the respective event should look like this:

EVT10 dtu\_ea\_trig\_0

9. Finally, activate dtu\_wdat and dtu\_wadr for the event you have selected. To do so, set the respective Qualifier on Active, the Level on State, and the event you have chosen in the previous step.

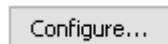


10. Make sure to do this for the data and the address action.

dtu\_wdat           EVT10  
dtu\_wadr           EVT10

This step finishes the configuration for a variable that is accessed from CPU0. If multiple cores access a variable, do the same settings in the SR tab.

1. Open your analyzer configuration and select Configure.



2. In the MCDS tab, select that the LMU is seen by SRI 1.

3. Which SRI slave is seen by SRI1           LMU (LMU SRAM, EMEM) ▼

4. Switch to the SRI tab.



5. Follow the configuration steps starting from step 4 from above but use one of the dtul\_ea\_trig\_\* triggers.

Figure 9 shows a screenshot of a data-trace configuration for accesses to a variable from multiple cores. In total, it is possible to trace up to four variables or memory ranges per observation block. Each trigger maps to a different event, which must then assign to the data and address trace actions, as shown in Figure 8.

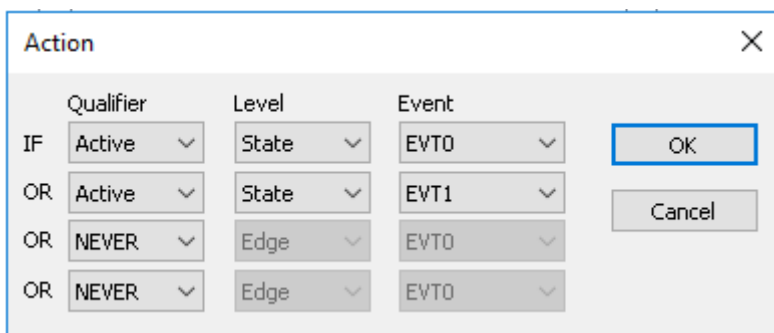


Figure 8: One processor or bus observation block can observe up to four different events.

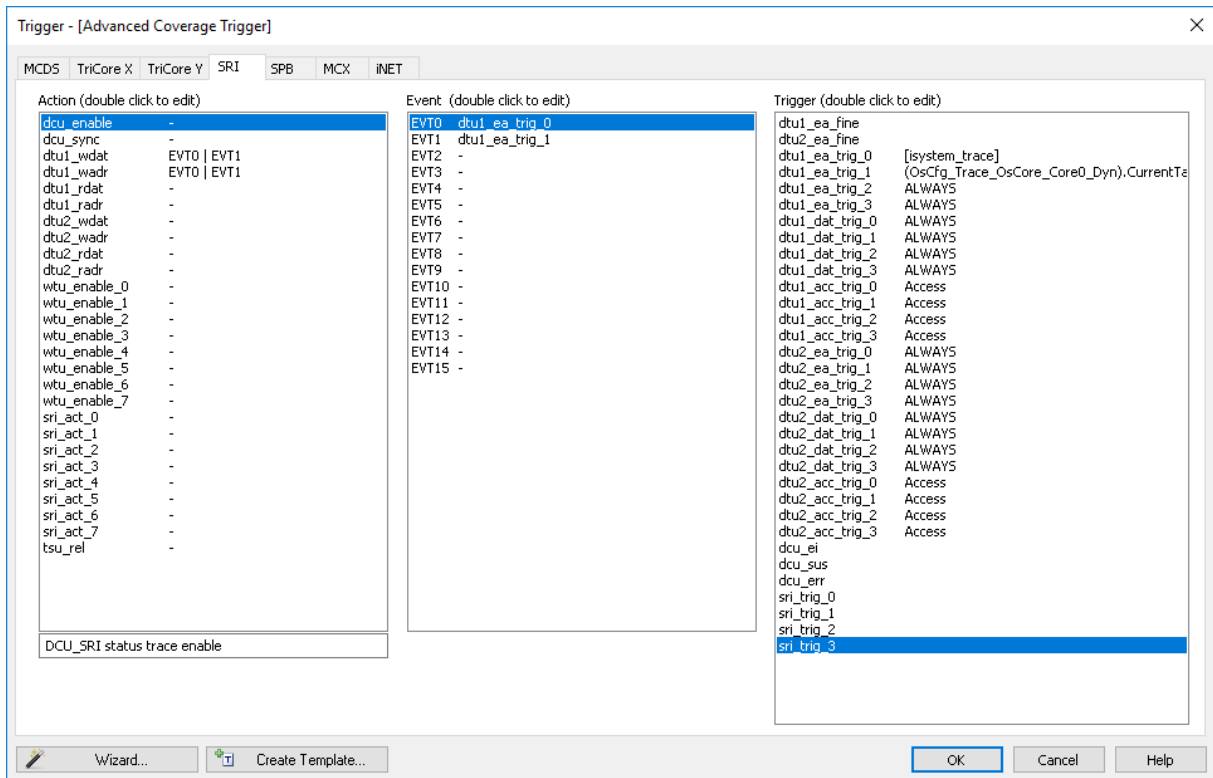


Figure 9: Data trace configuration for memory accesses to the `isystem_trace` variable and a memory range from multiple Infineon TriCore cores.

### 4.2.3 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. Execute the steps from the previous section, except 5 and 6, and configure the trigger, as shown in *Figure 10*.

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 `0xbeef1337` are valid addresses. The Y value must be higher than the X value.

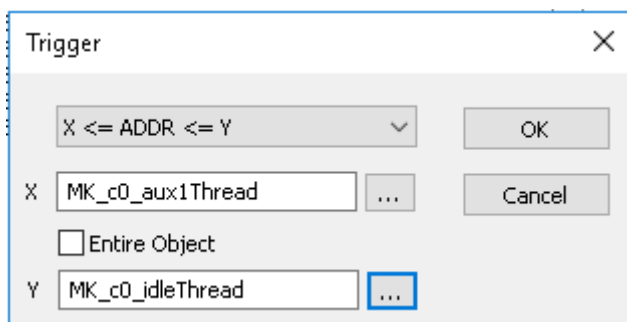


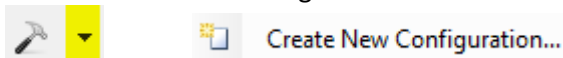
Figure 10: By specifying two objects or addresses, a range-comparator can span larger memory areas.

### 4.3 Renesas RH850 Software Trace

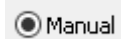
Renesas Software trace is an RH850 specific instrumentation based trace technique. It uses dedicated assembly instruction called `DBCP`, `DBTAG`, and `DBPUSH` to create trace messages at points of interest. The user can decide where and with which arguments to call the respective instructions. The first instruction `DBCP` creates a trace message with the current value of the instruction pointer. The `DBTAG` message creates a message with a constant value (known at compile time), while `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 *section 3.5* for Task State/ISR Profiling and *section 3.7* for Runnable Profiling with instrumentation.

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

1. Create a new configuration by clicking the drop-down arrow next to the hammer symbol and select: Create New Configuration.



2. Specify a name, select the Profiler checkbox, and choose Manual Trigger Configuration. Confirm with OK.



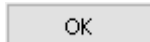
3. Switch to the PE1.



4. The default configuration already contains most of the necessary settings. The only change required is to disable this setting: Record Program trace.



5. Close the hardware configuration by clicking the OK button.



The resulting configuration should look as depicted in Figure 11. 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.

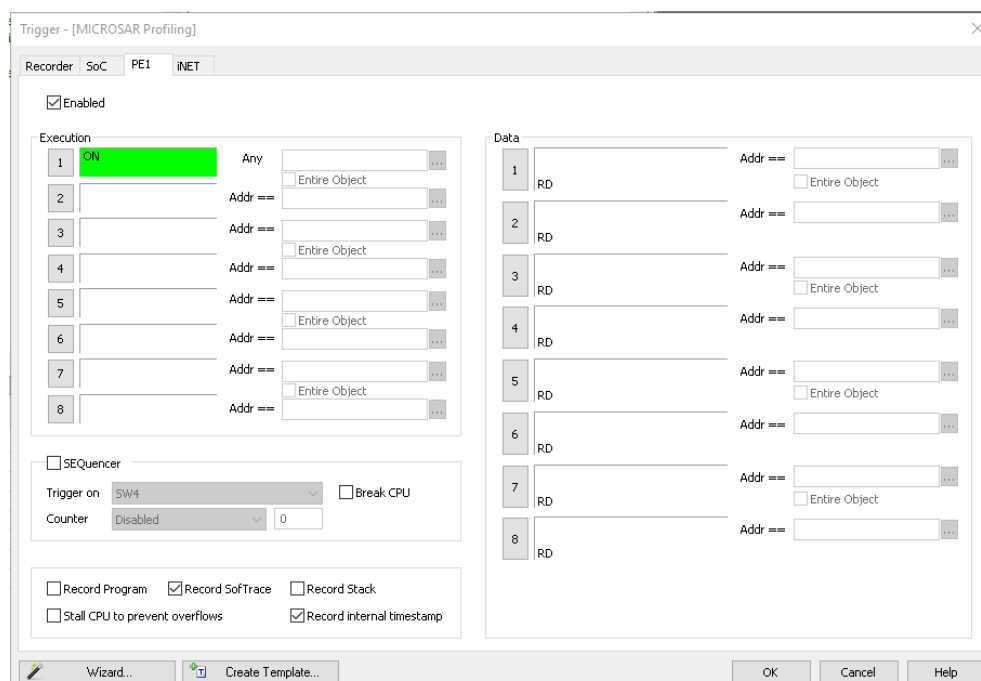



Figure 11: Hardware configuration to record Renesas software trace messages in winIDEA.

## 4.4 BTF Export

The winIDEA Profiler supports the export of traces into the BTF format. BTF is a CSV based trace format that is supported by different timing tool vendors. By using iTCHi, the configuration for BTF export is part of the Profiler XML automatically. Each Task and ISR object should reference a BTF mapping, as shown in the following listing. Note that BTF export only makes sense for task state profiling.

```
<BTFMappingType>TypeEnum_BTFMapping</BTFMappingType>
```

The mapping maps a state to a BTF transition, as shown in *Listing 11*. The Name-tag is the state as displayed in the winIDEA Profiler timeline, and the Value-tag is the respective BTF transition for a change to that state. To export a BTF file, follow these steps:

1. Load symbols  to make sure that the latest iSYSTEM Profiler XML is in use.
2. Record a trace with the necessary configuration to record threads and Runnables.
3. Select the export button in the Profiler timeline, choose BTF export, and export.



4. The result is a BTF trace, as shown in *Figure 12*.

```
<TypeEnum>
  <Name>TypeEnum_BTFMapping</Name>
  <Enum><Name>NEW</Name><Value>Active</Value></Enum>
  <Enum><Name>READY</Name><Value>Ready</Value></Enum>
  <Enum><Name>READY_SYNC</Name><Value>Ready</Value></Enum>
  <Enum><Name>RUNNING</Name><Value>Running</Value></Enum>
  <Enum><Name>WAITING_EVENT</Name><Value>Waiting</Value></Enum>
  <Enum><Name>WAITING_SEM</Name><Value>Waiting</Value></Enum>
  <Enum><Name>READ_ASYNC</Name><Value>Waiting</Value></Enum>
  <Enum><Name>WAITING</Name><Value>Waiting</Value></Enum>
  <Enum><Name>TERMINATED_TASK</Name><Value>Terminated</Value></Enum>
  <Enum><Name>TERMINATED_ISR</Name><Value>Terminated</Value></Enum>
  <Enum><Name>INVALID</Name><Value>Terminated</Value></Enum>
  <Enum><Name>QUARANTINED</Name><Value>Terminated</Value></Enum>
  <Enum><Name>SUSPENDED</Name><Value>Terminated</Value></Enum>
</TypeEnum>
```

Listing 11: Mapping from thread states to BTF state transitions. This mapping is required for the winIDEA Profiler to execute a correct BTF export.

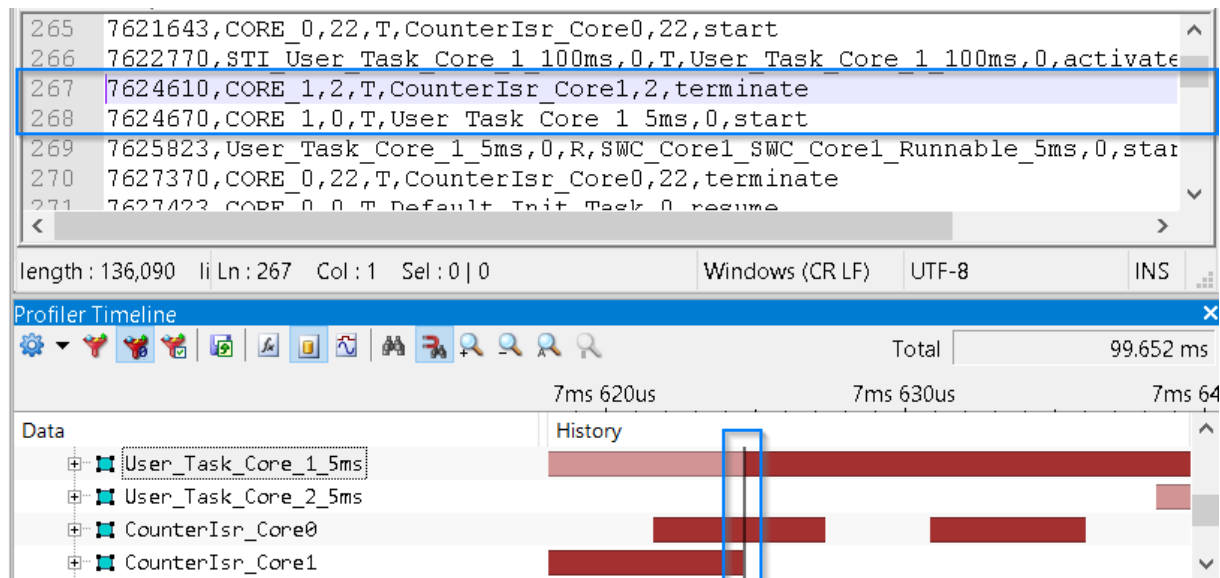


Figure 12: The winIDEA Profiler can export to the BTF format. Multiple timing tool vendors support BTF.

## 5 Technical Support

### 5.1 Online Resources

<p><a href="#">Online Help</a> ▶</p> <p>winIDEA and testIDEA online help</p>	<p><a href="#">Knowledge Base</a> ▶</p> <p>Tips &amp; tricks categorized by issue type and architecture</p>	<p><a href="#">Tutorials</a> ▶</p> <p>From beginner to expert</p>
<p><a href="#">Technical Notes</a> ▶</p> <p>How-tos for winIDEA functionalities with scripts</p>	<p><a href="#">Application Notes</a> ▶</p> <p>How-to notes on advanced use-cases</p>	<p><a href="#">Webinars</a> ▶</p> <p>Technical webinars about ISYSTEM tools with use cases</p>

### 5.2 Contact

Please visit <https://www.isystem.com/contact.html> for contact details.

iSYSTEM makes every effort to ensure the accuracy and reliability of the information provided in this document at the time of publishing. While iSYSTEM reserves the right to make changes to its products and the specifications detailed herein, it does not make any representations or commitments to update this document.

© iSYSTEM. All rights reserved.