

iSYSTEM Elektrobit EB tresos AutoCore: Profiling Application Note



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 |
| 1.1 | Trace Objects | 2 |
| 1.2 | Trace Techniques | 3 |
| 2 | Running Task/ISR2 Profiling..... | 4 |
| 2.1 | OS Configuration | 4 |
| 2.2 | OS Running Task/ISR2 Information | 4 |
| 2.3 | winIDEA Profiler Configuration | 7 |
| 2.4 | winIDEA Profiler Visualization..... | 9 |
| 3 | OS Tasks State Trace without Instrumentation | 11 |
| 3.1 | OS Configuration | 11 |
| 3.2 | OS Task State Information | 11 |
| 3.3 | OS ISR2 State Information..... | 11 |
| 3.4 | winIDEA Profiler Configuration | 12 |
| 3.5 | winIDEA Profiler Visualization..... | 17 |
| 4 | OS Task/ISR2 State Trace with Instrumentation..... | 19 |
| 4.1 | OS Configuration | 19 |
| 4.2 | OS Task State Information | 20 |
| 4.3 | OS ISR2 State Information..... | 20 |
| 4.4 | winIDEA Profiler Configuration | 21 |
| 4.5 | winIDEA Profiler Visualization..... | 25 |
| 5 | Processor-specific Trace Configurations..... | 26 |
| 5.1 | Infineon AURIX MCDS | 26 |
| 5.2 | Renesas RH850 Software Trace | 28 |
| 6 | BTF Export..... | 31 |
| 7 | Inspectors | 33 |
| 7.1 | Task Metric Analysis..... | 33 |
| 8 | Technical Support | 35 |
| 8.1 | Online Resources | 35 |
| 8.2 | Contact..... | 35 |

1 Introduction

This document describes how to use the winIDEA Analyzer for the timing analysis of the Elektrotbit EB tresos AutoCore operating system. Timing analysis records and examines the runtime behavior of operating system objects such as tasks and ISRs.

Recording and profiling the data required for timing analysis follows a common pattern independent from the hardware, operating system, and the number or type of objects of interest. Firstly, the objects of interest must be defined. An overview of possible objects is given in the Section Trace Objects. Analyzing the CPU utilization requires recording task and ISR states, whereas analyzing complex event chains may also require recording Runnable events and sender-receiver interfaces. With the list of desired objects, the OS can be configured so that those objects become available to the trace tool. For some objects, like tasks, no configuration may be required. This step is called *Operating System Configuration*.

Next, a hardware trace is recorded. A hardware trace utilizes dedicated microcontroller features that allow data, made available via the OS Configuration, to be recorded and sent off the chip via a dedicated trace interface. The various OS trace concepts are described independent of the underlying processor hardware. Section 5 "Processor-specific Trace Configurations" provides some insights in special trace features of some processor families.

Now, a hardware trace is recorded in winIDEA. A hardware trace contains low level objects that are not directly useful for the timing analysis. It is the job of the winIDEA profiler to analyze this data and transform it into a user-friendly format. This step is called *profiling*. For the profiling step winIDEA must be aware of the underlying OS information encoded in the hardware trace. This information is either provided in the form of an AUTOSAR ORTI file or/and an iSYSTEM proprietary winIDEA Profiler XML file. This step is called *Profiler Configuration*.

1.1 Trace Objects

Depending on the kind of analysis different operating system objects and RTE objects are of interest. This section gives an overview of those objects.

1.1.1 Tasks/ISR2s Running

The most basic type of objects that are usually of interest for the timing analysis are Tasks and ISR2s. Tasks are operating system containers for Runnables. ISR2s are interrupt triggered service routines that are used by the operating system, or can also be used as a container for Runnables. A trace that only contains the information about the currently running Task or ISR is called a Tasks/ISRs Running Trace. Please note that throughout this documents, both notations "ISR" and "ISR2" are identical and both refer to AUTOSAR ISR Category 2 Interrupt Service Routines.

1.1.2 Tasks/ISRs State

A Tasks/ISRs Running trace is not always sufficient for timing analysis. For example, while the information is sufficient to calculate the CPU utilization of a system it is not sufficient to calculate the response time of a given task.

The response time of a task is defined from activation to termination which may include periods during which the task is preempted. With a running task trace the information about activations and preemptions is not available. In this case, a Tasks/ISRs State trace is required. This type of trace records the detailed state of each task and ISR. This means the profiler knows when a task switches from the suspended state into the new state which indicates an activation. Also, the profiler knows when a task switches from running to preempted which indicates a preemption.

1.1.3 Runnables

In AUTOSAR Runnables are special functions defined in the RTE. They are mapped to tasks and executed in the context of those tasks. Runnables are triggered by RTE events such as periodic events and data-received-events. Tracing Runnables becomes important when a more detailed view in the application is required. In theory Runnable tracing can be done independently from Task/ISR tracing. However, whenever not only the currently running Runnable is of interest, but also the information about preemptions and resumes, it is mandatory to record a Task/ISR state trace in conjunction with the Runnable trace. By recording the task and ISR state information the profiler can reconstruct the Runnable preempt and resume events.

1.1.4 RTE Interfaces

In AUTOSAR Runnables are also mapped to software components. Communication between different software components happens via dedicated communication interfaces such as sender/receiver and client/server communication interfaces. For certain use-cases it may be of interest to record those interfaces. For example, a certain signal may have a data age constraint, i.e. a maximum time between the signal being produced by one Runnable and the signal being consumed by another Runnable. By tracing the RTE interfaces which are used to propagate this specific signal through the application the data age constraint can be validated.

1.2 Trace Techniques

In general, there are three trace measurement techniques: software, hybrid, and hardware based tracing. All techniques are meant to examine the runtime behavior of a system. In this section we focus on hardware based trace techniques. Hardware based tracing relies on a dedicated on-chip trace logic which is used to capture events of interest and send it off the chip. Depending on the chip none or more of the following techniques are available. The available trace techniques have a direct influence on which kind of Trace Objects can be recorded or not.

1.2.1 Program Flow Trace

A program flow trace (also called instruction trace) records the instructions that are executed by the CPU. This means a program flow trace shows the complete execution path of an application for the duration of the trace recording. Program flow tracing can be used for debugging, but also for profiling certain trace objects. The most common use-case is to create a Runnable trace based on the function entry and exit information that are part of an instruction trace.

1.2.2 Data Trace

Data tracing records read and write accesses to memory. This allows for monitoring the contents of global variables for the duration of the trace recording. Depending on the architecture comparators or filters are used to define which variables or memory regions should be traced. An example use-case for data tracing is monitoring the OS running task variable to generate a task trace.

1.2.3 Instrumented Data Trace

A normal data trace is not always sufficient to record a trace for all possible trace objects of interest. For example, there is usually no variable that indicates which Runnable is currently executed. In such cases instrumentation can be added to the application to write the desired information into a dedicated variable. This variable can then be recorded by using a regular data trace.

2 Running Task/ISR2 Profiling

This section describes how to record a trace of the currently running OS Task and ISR2.

Recording a Tasks/ISR2s Running Trace is usually done by tracing data objects. Information about these data objects imported into the trace tool via the AUTOSAR ORTI file. This file contains two attributes `RUNNINGTASK` and `RUNNINGISR2` which can be utilized to record the required information via data tracing.

2.1 OS Configuration

To use the ORTI trace feature it needs to be activated in EB tresos Studio . After the respective setting has been changed the OS must be regenerated and recompiled. The generated ORTI file can be found in the directory `"output\generated\ortios.orti"`.

2.2 OS Running Task/ISR2 Information

Based on the ORTI file the hardware must be configured in a way to record the variables which are used by the OS to signal the running task and ISR2. The first step is to extract those variables from the ORTI file. This can be done manually or with the help of winIDEA. If you want to use winIDEA, make sure to do the next step *Profiler Configuration* first and then come back to this section.

To extract the variables of interest the ORTI file is searched for `RUNNINGTASK` and `RUNNINGISR2`. Note that this attribute will appear multiple times. Firstly, in the declaration section of the ORTI file and then again in the information section. For the hardware trace configuration the attribute in the information section is of interest.

For a single core application, the entry in the information section should look as depicted in Listing 1.

```
OS XY
{
  RUNNINGTASK = "OS_taskCurrent";
  RUNNINGISR2 = "OS_isrCurrent";
}
```

Listing 1: ORTI Attributes `RUNNINGTASK` and `RUNNINGISR2` for a single core application.

From the listing the symbolic names of the variables for the `RUNNINGTASK` and `RUNNINGISR2` trace can be retrieved. Based on this information a data trace for single variables can now be configured for the respective microcontroller architecture.

In case a multi-core application is used the entries in the information section look as depicted in Listing 2 **Error! Reference source not found.** There is a separate variable for `RUNNINGTASK` and `RUNNINGISR2` for each core. Accordingly, the OS uses the respective variables to signal the currently running task and ISR for each core individually.

```
/* OS information for Core0 */
RUNNINGTASK[0] = "OS_kernel_ptr[0]->taskCurrent";
RUNNINGISR2[0] = "OS_kernel_ptr[0]->isrCurrent";

/* OS information for Core1 */
RUNNINGTASK[1] = "OS_kernel_ptr[1]->taskCurrent";
RUNNINGISR2[1] = "OS_kernel_ptr[1]->isrCurrent";
```

Listing 2: ORTI Attributes `RUNNINGTASK` and `RUNNINGISR2` for a multi-core application.

The issue with the expression that is used to signal `RUNNINGTASK` and `RUNNINGISR2` is that it represents a dereferenced pointer and the address it points to may change over time. Such an approach is perfectly suitable for a static analysis, i.e. stopping the CPU, reading out the current address location the pointer points to and display the contents in a dedicated OS status view. However, for trace during runtime this is not a viable approach.

Alternatively, the EB tresos AutoCore OS provides other global data objects which can also be utilized for `RUNNINGTASK` and `RUNNINGISR2` tracing.

The `RUNNINGTASK` and `RUNNINGISR2` definitions of the ORTI file can be replaced (overwritten) by an iSYSTEM proprietary Profiler Configuration XML file.

Listing 3 shows a sample iSYSTEM Profiler XML file. It imports the ORTI file generated by EB tresos Studio and overwrites the `RUNNINGTASK` attributes of the ORTI file with the alternative global variables (“Expression”) suitable for hardware data trace.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>emcc_AutoCore_demo</Name>
  <NumCores>2</NumCores>
  <ORTI>os.orti</ORTI>
  <Profiler>
    <Object>
      <Definition>RUNNINGTASK[0]</Definition>
      <Description>TASKs.Core0</Description>
      <Expression>(OS_kernelData_core0).taskCurrent</Expression>
      <Type>OS:RUNNINGTASK</Type>
      <DefaultValue>NO_TASK</DefaultValue>
      <Core>0</Core>
    </Object>
    <Object>
      <Definition>RUNNINGTASK[1]</Definition>
      <Description>TASKs.Core1</Description>
      <Expression>(OS_kernelData_core1).taskCurrent</Expression>
      <Type>OS:RUNNINGTASK</Type>
      <DefaultValue>NO_TASK</DefaultValue>
      <Core>1</Core>
    </Object>
    <Object>
      <Definition>RUNNINGISR2[0]</Definition>
      <Description>ISR2s.Core0</Description>
      <Expression>(OS_kernelData_core0).isrCurrent</Expression>
      <Type>OS:RUNNINGTASK</Type>
      <DefaultValue>NO_ISR</DefaultValue>
      <Core>0</Core>
    </Object>
    <Object>
      <Definition>RUNNINGISR2[1]</Definition>
      <Description>ISR2s.Core1</Description>
      <Expression>(OS_kernelData_core1).isrCurrent</Expression>
      <Type>OS:RUNNINGTASK</Type>
      <DefaultValue>NO_ISR</DefaultValue>
      <Core>1</Core>
    </Object>
  </Profiler>
</OperatingSystem>
```

Listing 3: winIDEA Profiler XML File for Multi-core OS Running Task and ISR2 Profiling on EB tresos AutoCore

If you have already done the Profiler Configuration for the respective ORTI/XML file you can extract the variables from the RTOS Profiler Options in the winIDEA profiler as depicted in Figure 1. Each ORTI attribute or XML object, which is traceable, is shown in this menu. For each entry there is an attribute called *Signaling* which points to the respective variable that is used to signal that attribute. Comparing the listing and the figure shows that the respective variables are the same.

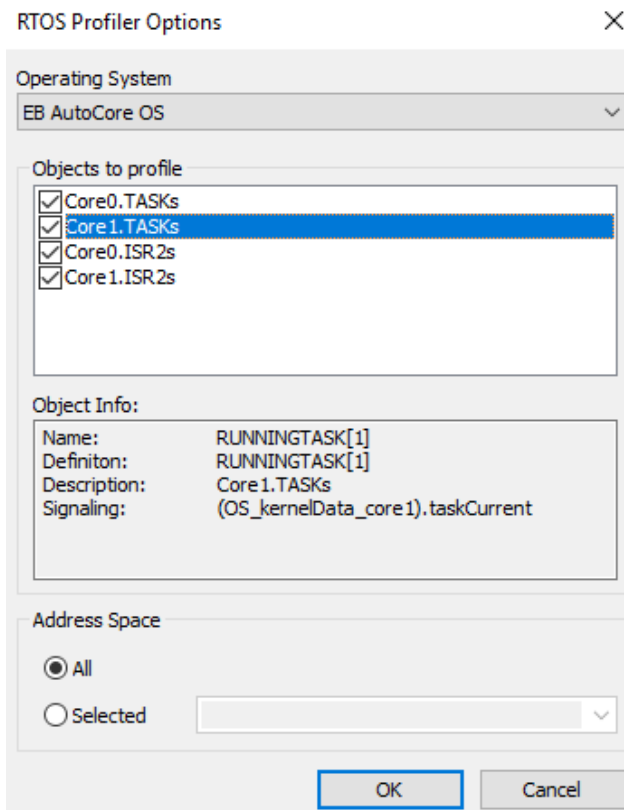


Figure 1: The RTOS Profiler Options menu shows the profiler objects defined in the XML file.

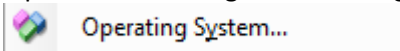
2.3 winIDEA Profiler Configuration

Once the OS and the winIDEA Profiler XML is configured to provide the necessary information and the hardware is configured to record those information, winIDEA must be made aware of this information. This is done in two steps. First, the Profiler XML file is added to the workspace and then the winIDEA profiler is configured to use this information. To add the XML file to the workspace execute the following steps.

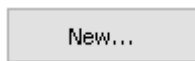
1. Open the Debug menu.

Debug

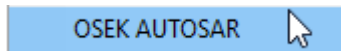
2. Open the OS Configuration Dialog.



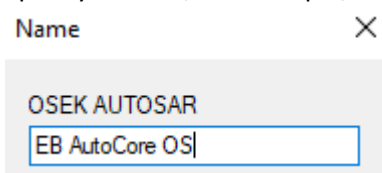
3. Create a new OS Configuration.



4. Select OSEK AUTOSAR OS.



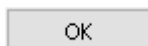
5. Specify a name, for example, *EB AutoCore OS*.



6. Select XML as RTOS description file type.

| Property | Value |
|--------------------------------|------------------|
| [-] Configuration | |
| RTOS description file type | iSYSTEM XML |
| RTOS description file location | EBAutoCoreOS.xml |

7. Select the XML file and click OK.



8. Make sure to load symbols to make the change active.



Next, the winIDEA profiler is configured to use the Profiler XML file, too. To do so execute the following steps.

1. Open the profiler configuration. Make sure it is the same configuration for which data tracing of the XML variables is configured.



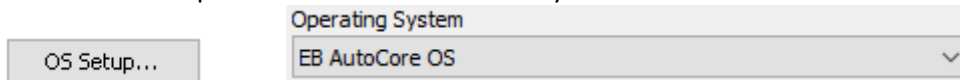
2. Select the hardware tab and make sure that the profiler is activated.



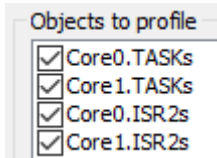
3. Change to the profiler tab and make sure that OS objects are selected.



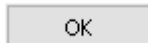
4. Click on OS Setup and select the OS for which you have added the XML file.



5. Select all tasks and ISRs you want to profile. (Again, only those objects for which the signaling variable is record will show up in the profiler timeline.)



- 6.
7. Confirm with OK.



8. Start a new trace recording.



2.4 winIDEA Profiler Visualization

2.4.1 Profiler Timeline



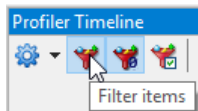
If the application is running you should see the OS objects in the profiler timeline as shown in Figure 2. If nothing is shown check the trace window  if accesses to the signaling variable have been recorded. Also make sure that the data section  of the profiler timeline is selected to be visible.



Figure 2: Running Task/ISR2 Trace in the winIDEA Profiler Timeline.

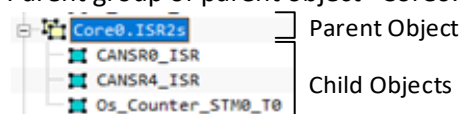
Additional notes to the winIDEA Profiler Timeline in Figure 2:

- As the application comprises a large number of OS tasks, only the tasks and ISR2s of interest have been filtered by means of a string filter ("*" acts a wildcard). The filters view can be enabled via the "Filter items" icon.



- The profiler objects can be re-ordered according to the user requirements by simply dragging-and-dropping by means of the cursor. This is applicable for both so-called parent and child objects. However, child objects can only be moved within their parent group.

Parent group of parent object "Core0.ISR2s":



- The profiler timeline offers three markers: black, blue and yellow.

The black marker is the “main” marker, which can be set to a simple left-mouse click within the timeline.

The blue and the yellow and blue markers can be used to quickly measure the time between two events of interest, such as the time between two instances of the same task.

The blue marker can be set by “Ctrl + left mouse”, the yellow marker is set by “Ctrl + right mouse” click. The absolute location (in time) of the blue and yellow markers is shown in the footer of the profiler timeline, as well as the time difference between the markers.

In Figure 2 the blue/yellow marker pair is used to measure the distance between two instances of the task “T_1MS_0”.

2.4.2 Profiler Statistics

The winIDEA Profiler also calculates statistics for the OS tasks and ISR2s. A sample Profiler Statistics windows is shown in Figure 3.

| Data | Count | Net Time | Net Average Time | Period Average |
|------------------------|-------|----------------------|------------------|--------------------|
| Core0.TASKs | | | | |
| NO_TASK_CORE_0 | 120 | 82.049960 ms 8.20% | 683.749 us 0.07% | 8.355993 ms 0.83% |
| SchMDiagStateTask_20ms | 50 | 4.776540 ms 0.48% | 95.530 us 0.01% | 20.000208 ms 2.00% |
| T_100MS | 272 | 178.419860 ms 17.83% | 655.955 us 0.07% | 3.515531 ms 0.35% |
| T_10MS_0 | 150 | 57.419980 ms 5.74% | 382.799 us 0.04% | 6.649771 ms 0.66% |
| T_1MS_0 | 1020 | 43.157780 ms 4.31% | 42.311 us 0.00% | 980.298 us 0.10% |
| T_20MS | 50 | 21.079970 ms 2.11% | 421.599 us 0.04% | 20.000622 ms 2.00% |
| T_500MS | 123 | 78.513600 ms 7.85% | 638.321 us 0.06% | 5.853225 ms 0.58% |

Figure 3: Sample Profiler Statistics for an OS Running Task Analysis

In the statistics in Figure 3 only the number of recorded task instances (Count), the total net execution time of each task, the average net time and the average period are displayed.

Figure 4 lists all the statistics which can be calculated by the profiler for each profile object.

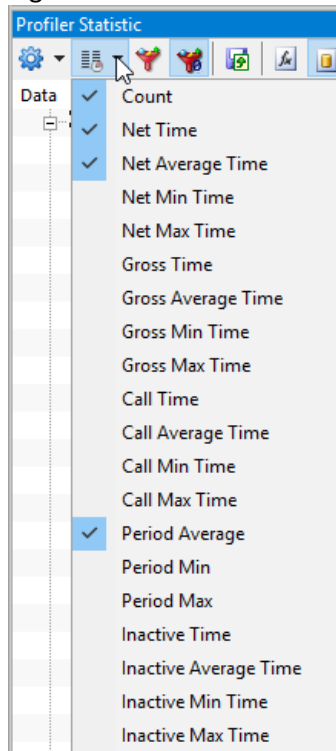


Figure 4: Complete list of calculated Profiler Statistics

3 OS Tasks State Trace without Instrumentation

This section explains how to record a Tasks/ISRs State Trace without the need for OS code instrumentation. Whether this approach is applicable depends on the capabilities of the processor hardware trace logic, i.e. the number of available data trace channels (address comparators) and the bandwidth of the trace interface (such as parallel Nexus trace port, ARM TPIU or AURIX DAP interface).

3.1 OS Configuration

As this approach still requires an ORTI file, generation of an ORTI file needs to be activated in EB tresos Studio. After the respective setting has been changed the OS must be regenerated and recompiled. The generated ORTI file can be found in the directory `output\generated\ort\os.orti`.

3.2 OS Task State Information

Task state tracing is based on the ORTI file task state attribute. To find the required symbols search the ORTI section for `TASK` objects and then for the `STATE` attribute within tasks. This search should yield a Task object definition as shown in Listing 1Listing 4.

```
TASK T_10MS_0
{
  vs_ID = "7";
  STATE = "OS_taskTableBase[7].dynamic->state";
  CURRENTACTIVATIONS = "OS_taskTableBase[7].dynamic->nAct";
  vs_MAXACTIVATIONS = "1";
  STACK = "OS_taskStack0_slot19";
  vs_SHAREDSTACK = "false";
  vs_STACKSIZE = "136";
  vs_ASSIGNEDPRIO = "34";
  vs_REALPRIORITY = "12";
  PRIORITY = "OS_taskTableBase[7].dynamic->prio";
  vs_TYPE = "BASIC";
  vs_USE_HW_FP = "false";
  vs_MEASURE_MAX_RUNTIME = "false";
  vs_TIMING_PROTECTION = "false";
};
```

Listing 4: Task object ORTI definition including STATE attribute

This task state attribute again uses a pointer to a structure element “state”, not really suitable for non-intrusive hardware trace during runtime. There is again (same as e.g. for `RUNNINGTASK`) an alternative global data object available in the AutoCore OS which can be utilized for hardware data trace. This means, the `STATE` definition of the ORTI file also needs to be overwritten by an alternative definition in an iSYSTEM Profiler XML file. Such a `TASKSTATE` definition is shown in Listing 3, for instance the global variable `OS_taskDynamic_core0[1].state` holds the current state of the OS task with index 1 running on core 0.

3.3 OS ISR2 State Information

There exists no state model for OSEK ISRs. Hence, the approach for tracing ISR2 still relies on the `RUNNINGISR2` attribute (as described in Chapter 2 “Running Task/ISR2 Profiling”).

3.4 winIDEA Profiler Configuration

The configuration of the profiler is a bit more complex for this use-case. Since the task state attribute is implemented differently for different operating systems the profiler must be made aware of the task state variables explicitly. A dedicated iSYSTEM Profiler XML file is used for that purpose. Listing 5 shows an example for such a Profiler XML file.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>EB_AutoCore_Demo</Name>
  <NumCores>2</NumCores>

  <ORTI>os.orti</ORTI>

  <Types>
    <TypeEnum>
      <Name>Type_TASKSTATE</Name>
      <Enum><Name>SUSPENDED</Name> <Value>0</Value></Enum>
      <Enum><Name>QUARANTINED</Name> <Value>1</Value></Enum>
      <Enum><Name>NEW</Name> <Value>2</Value></Enum>
      <Enum><Name>READY_SYNC</Name> <Value>3</Value></Enum>
      <Enum><Name>READY_ASYNC</Name> <Value>4</Value></Enum>
      <Enum><Name>RUNNING</Name> <Value>5</Value></Enum>
      <Enum><Name>WAITING</Name> <Value>6</Value></Enum>
    </TypeEnum>

    <TypeEnum>
      <Name>Type_TaskState_MAPPING</Name>
      <Enum><Name>NO_TASK</Name><Value>0xFF</Value></Enum>
      <Enum><Name>TASK_A</Name><Value>0</Value>
        <Property><Name>Expression</Name>
        <Value>OS_taskDynamic_core0[0].state</Value></Property>
        <Property><Name>Core</Name><Value>0</Value></Property>
      </Enum>
      <Enum><Name>TASK_B</Name><Value>1</Value>
        <Property><Name>Expression</Name>
        <Value>OS_taskDynamic_core0[1].state</Value></Property>
        <Property><Name>Core</Name><Value>0</Value></Property>
      </Enum>
      <Enum><Name>TASK_C</Name><Value>2</Value>
        <Property><Name>Expression</Name>
        <Value>OS_taskDynamic_core1[0].state</Value></Property>
        <Property><Name>Core</Name><Value>1</Value></Property>
      </Enum>
    </TypeEnum>

    <TypeEnum>
      <Name>Type_ISRSTATE</Name>
      <Enum><Name>SUSPENDED</Name> <Value>0x0</Value></Enum>
      <Enum><Name>NEW</Name> <Value>0x0</Value></Enum>
      <Enum><Name>RUNNING</Name> <Value>0x0</Value></Enum>
    </TypeEnum>

    <TypeEnum>
      <Name>Type_ISR2_to_BTF_State_Mapping</Name>
      <Enum><Name>Suspended</Name> <Value>Terminated</Value></Enum>
      <Enum><Name>Active</Name> <Value>Active</Value></Enum>
      <Enum><Name>Running</Name> <Value>Running</Value></Enum>
    </TypeEnum>
  </Types>

  <Profiler>
    <Object>
      <Definition>TASKSTATE</Definition>
      <Description>Tasks</Description>
      <Type>Type_TaskState_MAPPING</Type>
    </Object>
  </Profiler>
</OperatingSystem>
```

```

    <Expression>$(EnumType)</Expression>
    <DefaultValue>NO_TASK</DefaultValue>
    <Name>TASKSTATE</Name>
    <Level>Task</Level>
    <TaskState>
      <MaskID>0x0</MaskID> <MaskState>0xFF</MaskState>
      <MaskCore>0x0</MaskCore>
      <Type>Type_TASKSTATE</Type>
</TaskState>
<StateInfo><Name>SUSPENDED</Name><Property>Terminate</Property></StateInfo>
<StateInfo><Name>WAITING</Name><Property>Terminate</Property></StateInfo>
  <StateInfo><Name>RUNNING</Name> <Property>Run</Property></StateInfo>
</TaskState>
</Object>

<Object>
  <Definition>ISRSTATE0</Definition>
  <Description>ISR2s.Core0</Description>
  <Type>OS:RUNNINGISR2</Type>
  <Expression>(OS_kernelData_core0).isrCurrent</Expression>
  <DefaultValue>NO_ISR</DefaultValue>
  <Name>ISRSTATE0</Name>
  <Level>IRQ1</Level>
  <Core>0</Core>
  <TaskState>
    <MaskID>0xFFFFFFFF</MaskID>
    <MaskState>0x0</MaskState>
    <MaskCore>0x0</MaskCore>
    <Type>Type_ISRSTATE</Type>
    <BTFMappingType>Type_ISR2_to_BTF_State_Mapping</BTFMappingType>
    <StateInfo><Name>SUSPENDED</Name><Property>Terminate</Property></StateInfo>
    <StateInfo><Name>RUNNING</Name><Property>Run</Property></StateInfo>
  </TaskState>
</Object>

</Profiler>
</OperatingSystem>

```

Listing 5: Sample iSYSTEM Profiler XML for EB AutoCore OS Task State Profiling without Instrumentation

In the upper section (“Types”) an enumeration type is defined (“Type_TaskState_MAPPING”), which maps a task name, displayed in the winIDEA Profiler to its corresponding state variable in the OS task status/control structure/array.

In the lower section (“Profiler”), a new profiler object is created. It is defined as a “TASKSTATE” object, telling the profiler that this object is used for OS task state reconstruction. The “Type” and “Expression” tags tell the profiler to use the “Type_TaskState_MAPPING” for the task state analysis.

The OS Profiler options only lists the object “Tasks”, as depicted in Figure 5.

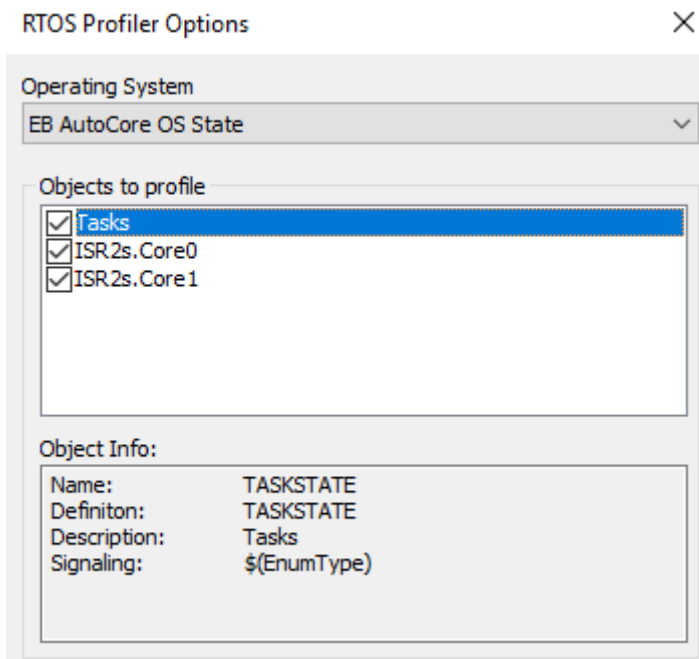
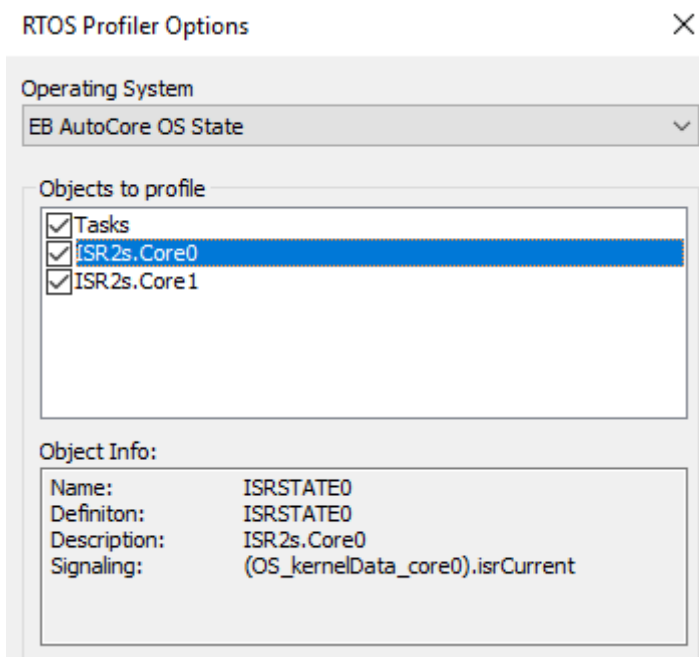


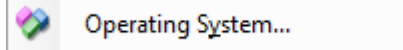
Figure 5: The RTOS Profiler Options menu shows the TASKSTATE profiler object defined in the XML file.



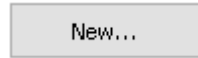
Once the OS is configured to provide the necessary information and the hardware is configured to record that information, winIDEA must be made aware of this information. This is done in two steps. First, the XML file is added to the workspace and then the winIDEA profiler is configured to use this information. To add the XML file to the workspace, execute the following steps.

1. Open the Debug menu.
Debug

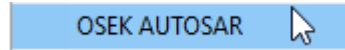
2. Open the OS Configuration Dialog.



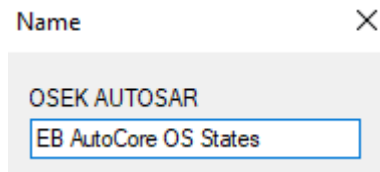
3. Create a new OS Configuration.



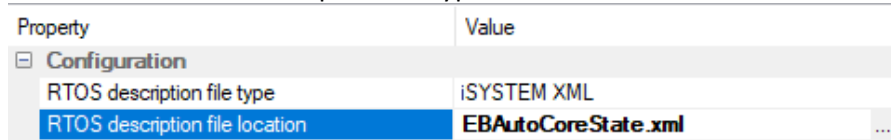
4. Select OSEK AUTOSAR OS.



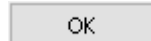
5. Specify a name, for example, *EB AutoCore OS*.



6. Select XML as RTOS description file type.



7. Select the XML file and click OK.



8. Make sure to load symbols to make the change active.



Next, the winIDEA profiler is configured to use the Profiler XML file, too. To do so execute the following steps.

1. Open the profiler configuration. Make sure it is the same configuration for which data tracing of the XML variables is configured.



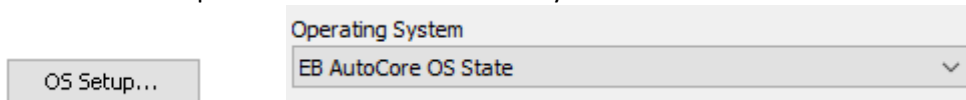
2. Select the hardware tab and make sure that the profiler is activated.



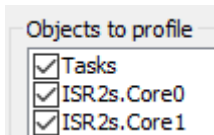
3. Change to the profiler tab and make sure that OS objects are selected.



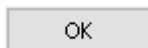
4. Click on OS Setup and select the OS for which you have added the XML file.



5. Select all tasks and ISRs you want to profile. (Again, only those objects for which the signaling variable is record will show up in the profiler timeline.)



6. Confirm with OK.





7. Start a new trace recording.



3.5 winIDEA Profiler Visualization

3.5.1 Profiler Timeline

If the application is running you should see the OS objects in the profiler timeline as shown in Figure 6. If nothing is shown check the trace window  if accesses to the signaling variable have been recorded. Also make sure that the data section  of the profiler timeline is selected to be visible.

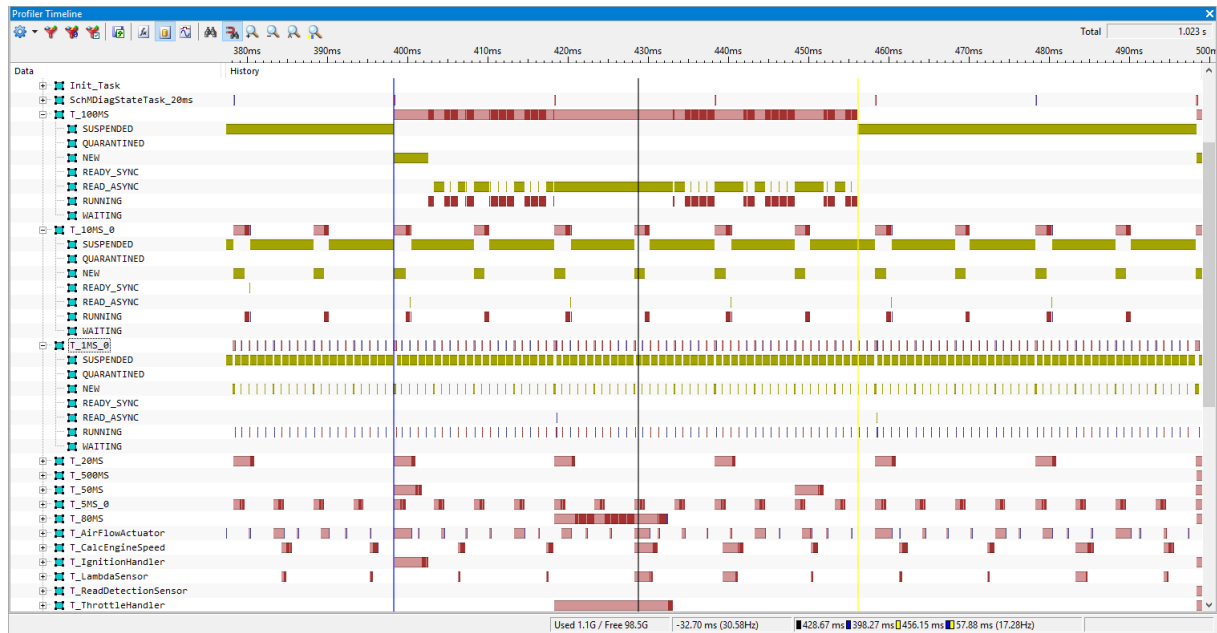
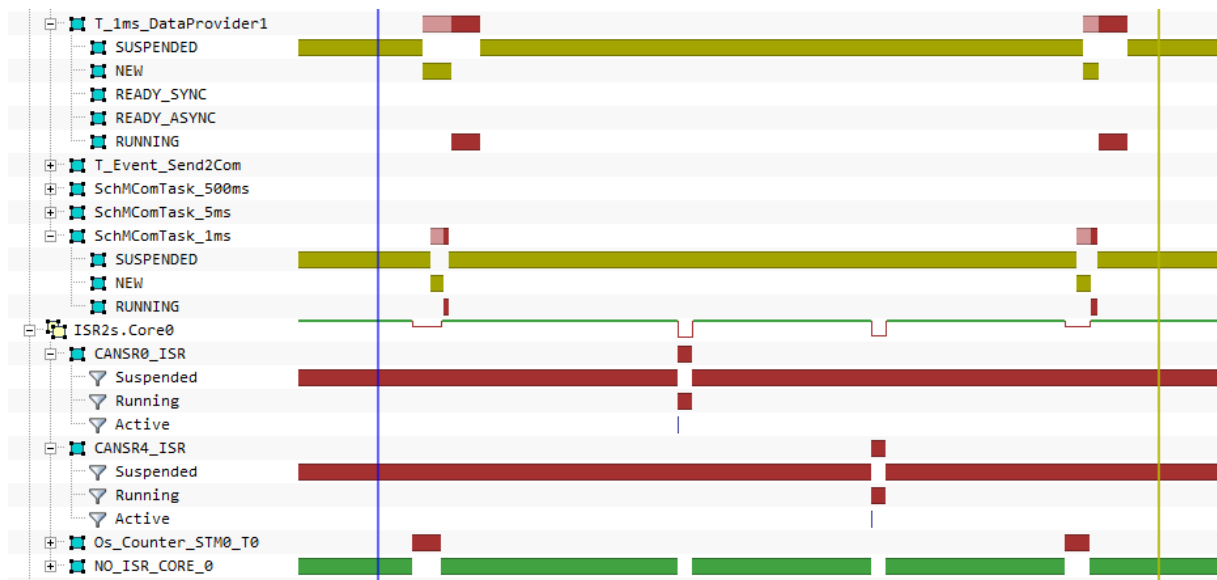


Figure 6: OS Task State Trace (without Instrumentation) in the winIDEA Profiler Timeline.



3.5.2 Profiler Statistics

The winIDEA Profiler calculates statistics for the OS tasks states. Each state of each task is treated as a separate profiler object, and thus the statistics are calculated for each OS task state. A sample Profiler Statistics window is shown in Figure 7. Additional statistics can be calculated by means of iSYSTEM Profiler Inspectors (see).

| Profiler Statistic | | | | | | | | | |
|--------------------|-------|---------------|------------------|----------------|-------|---------------|--------|--|--|
| Data | Count | Net Time | Net Average Time | Period Average | | | | | |
| NO_TASK_CORE_0 | | | | | | | | | |
| RUNNING | 5799 | 98.276150 ms | 9.60% | 16.947 us | 0.00% | 176.335 us | 0.02% | | |
| T_10MS_0 | | | | | | | | | |
| SUSPENDED | 103 | 817.414865 ms | 79.88% | 7.936066 ms | 0.78% | 9.998957 ms | 0.98% | | |
| NEW | 102 | 139.463210 ms | 13.63% | 1.367286 ms | 0.13% | 9.999952 ms | 0.98% | | |
| RUNNING | 154 | 60.828000 ms | 5.94% | 394.987 us | 0.04% | 6.666675 ms | 0.65% | | |
| READ_ASYNC | 45 | 4.578400 ms | 0.45% | 101.742 us | 0.01% | 23.181837 ms | 2.27% | | |
| READY_ASYNC | 7 | 710.310 us | 0.07% | 101.472 us | 0.01% | 133.331608 ms | 13.03% | | |
| T_10MS_1 | | | | | | | | | |
| SUSPENDED | 102 | 866.769411 ms | 84.71% | 8.497739 ms | 0.83% | 10.000352 ms | 0.98% | | |
| RUNNING | 204 | 111.567054 ms | 10.90% | 546.897 us | 0.05% | 4.979155 ms | 0.49% | | |
| NEW | 102 | 29.434964 ms | 2.88% | 288.578 us | 0.03% | 10.000168 ms | 0.98% | | |
| READ_ASYNC | 102 | 7.111540 ms | 0.69% | 69.720 us | 0.01% | 10.000001 ms | 0.98% | | |
| T_100MS | | | | | | | | | |
| SUSPENDED | 10 | 449.944900 ms | 43.97% | 44.994490 ms | 4.40% | 99.880867 ms | 9.76% | | |
| RUNNING | 285 | 202.500290 ms | 19.79% | 710.527 us | 0.07% | 3.579345 ms | 0.35% | | |
| READ_ASYNC | 275 | 276.964165 ms | 27.07% | 1.007142 ms | 0.10% | 3.711752 ms | 0.36% | | |
| NEW | 10 | 88.994750 ms | 8.70% | 8.899475 ms | 0.87% | 99.993053 ms | 9.77% | | |
| T_CalcEngineSpeed | | | | | | | | | |
| T_LambdaSensor | | | | | | | | | |

Figure 7: winIDEA Profiler Statistics for OS Task States

4 OS Task/ISR2 State Trace with Instrumentation

This section describes how to record a Task/ISR2s State Trace by using the EB tresos AutoCore OS Trace Hooks. These Hooks are OS macros which are executed at special points of interest in the OS execution. The macros are already part of the standard EB tresos AutoCore OS, but per default, the macros are defined as “empty”, i.e. do not include any code and therefore do not have any effect. In order to activate the hooks the macro must instead be defined as trace tool specific code. Such instrumentation code gathers the relevant info provided via the macro parameters and stores them with an atomic write operation, into a traceable global variable. This tracing variable also needs to be created by some instrumentation code. All core may share this trace variable as only the actual write operation to this variable is relevant. The variable does not need to store any data.

It is also worthwhile mentioning that such a “hybrid” trace solution (i.e. minimal instrumentation code combined with on-chip trace logic) represents the most efficient trace solution, i.e. has the least performance requirements with regards to number of data trace channels and trace interface bandwidth. In addition, some processor-specific instrumentation trace concepts, such as Renesas RH850 Software Trace or ARM CoreSight System Trace Macrocell (STM), allow for very efficient OS tracing/profiling solutions.

4.1 OS Configuration

In order to activate the trace tool specific trace macro definitions, the following configurations in the EB tresos Studio are needed.

The C header file named “Dbg.h” needs to be created containing the new macro definitions.

```

/* =====
OS task and ISR2 state tracing macro definitions for EB tresos AutoCore
CPU: Infineon TriCore
Compiler: Tasking

(c) iSYSTEM 2017
===== */
#ifndef __DBG_H
#define __DBG_H

#ifndef CPU_CORE_ID
#define CPU_CORE_ID 0xFE1C
#endif

extern unsigned long isystem_os_trace[2];

/* Tasks */
#ifndef OS_TRACE_STATE_TASK
#define OS_TRACE_STATE_TASK(StateId,oldValue,newValue) \
    isystem_os_trace[0] = (StateId) | \
    (newValue << 8) | \
    (__mfcr(CPU_CORE_ID) << 24);
#endif

/* ISR2s */
#ifndef OS_TRACE_STATE_ISR
#define OS_TRACE_STATE_ISR(IsrId,oldValue,newValue) \
    isystem_os_trace[1] = (IsrId) | \
    (newValue << 8) | \
    (__mfcr(CPU_CORE_ID) << 24);
#endif

#endif /* if !defined( DBG_H ) */
/*===== [end of file] =====*/

```

Listing 6: Sample Dbg.h header file containing trace macro definitions for the Infineon TriCore CPU.

The C header file “Dbg.h” needs to be copied into the EB tresos Studio project directory “<project_folder>\output\generated\include”.

In order to include the Dbg.h header file into the overall Build process of EB tresos Studio, the “OsTrace” option within the OS module of the EB tresos Studio ECU project must be enabled.

Also a global variable `unsigned long isystem_os_trace[2]` needs to be created, e.g. within a user-written C source file.

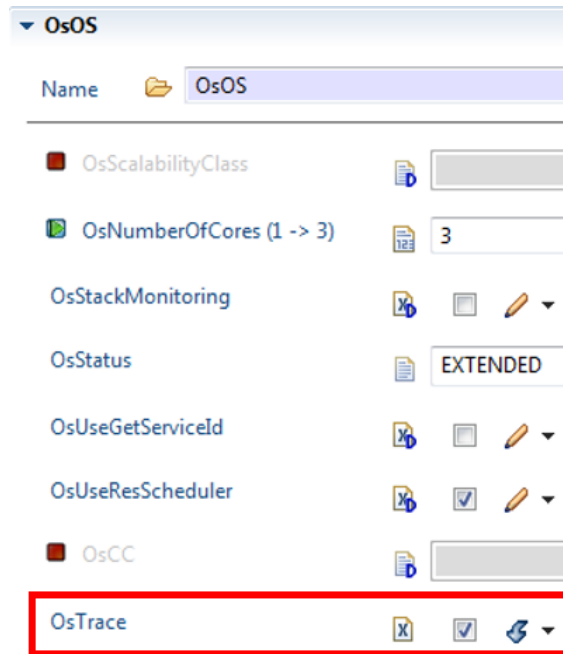


Figure 8: The option “OsTrace” of the OS Module must be enabled to activate the Instrumentation code in “Dbg.h”.

4.2 OS Task State Information

OS task state transitions are signaled by the OS by calling the `OS_TRACE_STATE_TASK` macro.

```
/* Parameters:
   StateId = Task Index
   oldValue = previous Task State (before state transition)
   newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_TASK(StateId,oldValue,newValue) \
    isystem_os_trace[0] = (StateId << 0) | \
                          (newValue << 8) | \
                          (ISYS_COREID << 24);
```

The macro `ISYS_COREID` is typically an intrinsic function used to retrieve the ID of the executing core. On an Infineon AURIX device it would for instance be define as `__mfcr(CPU_CORE_ID)`.

4.3 OS ISR2 State Information

OS ISR2 state transitions are signaled by the OS by calling the `OS_TRACE_STATE_ISR` macro.

```
/* Parameters:
   isrId = ISR2 Index
   oldValue = previous Task State (before state transition)
   newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_ISR(isrId,oldValue,newValue) \
    isystem_os_trace[1] = (isrId << 0) | \
```

```
(newValue << 8) | \
(ISYS_COREID << 24);
```

The macro `iSYS_COREID` is typically an intrinsic function used to retrieve the ID of the executing core. On an Infineon AURIX device it would for instance be define as `__mfcr(CPU_CORE_ID)`.

Note: Grouping the `isystem_os_trace` variables for tasks and ISRs into an array has the advantage that only one data trace channel (one address range comparator) of the on-chip trace logic is needed.

4.4 winIDEA Profiler Configuration

A single global trace variable `isystem_trace` is used to record all task and ISR2 information. Configure winIDEA to record data trace for this variable.

By now the operating system is configured to record all relevant task information into a single variable. Additionally, the hardware is configured to record all write access to this variable. The last step is to inform the profiler about how to interpret the content of this variable. For this purpose, a iSYSTEM Profiler XML as shown in Listing 7 is used.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>EB_AutoCore_Demo</Name>
  <NumCores>2</NumCores>
  <ORTI>os.orti</ORTI>

  <Types>
    <TypeEnum>
      <Name>Type_TASKSTATE</Name>
      <Enum><Name>SUSPENDED</Name> <Value>0</Value></Enum>
      <Enum><Name>QUARANTINED</Name> <Value>1</Value></Enum>
      <Enum><Name>NEW</Name> <Value>2</Value></Enum>
      <Enum><Name>READY_SYNC</Name> <Value>3</Value></Enum>
      <Enum><Name>READY_ASYNC</Name> <Value>4</Value></Enum>
      <Enum><Name>RUNNING</Name> <Value>5</Value></Enum>
      <Enum><Name>WAITING</Name> <Value>6</Value></Enum>
    </TypeEnum>
  </Types>

  <Profiler>
    <Object>
      <Definition>TASKSTATE</Definition>
      <Description>Tasks</Description>
      <Type>OS:vs_SIGNAL_RUNNINGTASK</Type>
      <DefaultValue>NO_TASK</DefaultValue>
      <Name>TASKSTATE</Name>
      <Level>Task</Level>
      <Expression>isystem_os_trace[0]</Expression>
      <TaskState>
        <MaskID>0xFF</MaskID><MaskState>0xFF00</MaskState>
        <MaskCore>0xFF000000</MaskCore>
        <Type>Type_TASKSTATE</Type>
      </TaskState>
    </Object>

    <Object>
      <Definition>ISRSTATE</Definition>
      <Description>ISRs2</Description>
      <Type>OS:vs_SIGNAL_RUNNINGISR2</Type>
      <DefaultValue>NO_ISR</DefaultValue>
      <Name>ISRSTATE</Name>
      <Level>IRQ3</Level>
      <Expression>isystem_os_trace[1]</Expression>
```

```

<TaskState>
  <MaskID>0xFF</MaskID><MaskState>0xFF00</MaskState>
  <MaskCore>0xFF000000</MaskCore>
  <Type>Type_TASKSTATE</Type>
</TaskState>
</Object>
</Profiler>
</OperatingSystem>

```

Listing 7: Sample iSYSTEM Profiler XML file for OS Task and ISR2 State Tracing by means of Instrumentation

Note: In some cases the XML tag `DefaultValue` needs to define with `INVALID_TASK` or `INVALID_ISR` instead of `NO_TASK` or `NO_ISR`, Please refer to the individual ORTI file.

The OS Profiler options lists the objects “Tasks” and “ISR2”, as depicted in Figure 5.

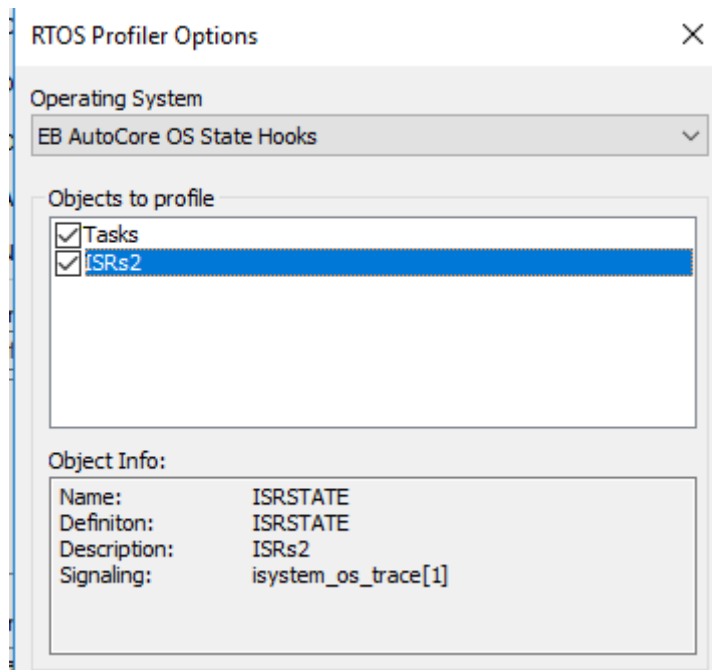


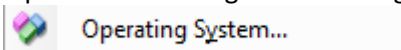
Figure 9: The RTOS Profiler Options menu shows the TASKSTATE and ISRSTATE profiler object defined in the XML file.

Once the OS is configured to provide the necessary information and the hardware is configured to record those information, winIDEA must be made aware of this information. This is done in two steps. First, the XML file is added to the workspace and then the winIDEA profiler is configured to use this information. To add the XML file to the workspace execute the following steps.

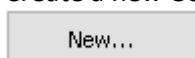
9. Open the Debug menu.



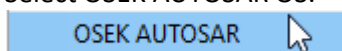
10. Open the OS Configuration Dialog.



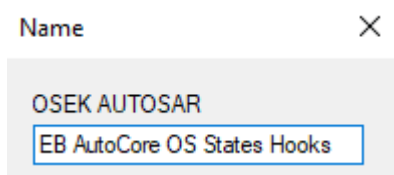
11. Create a new OS Configuration.



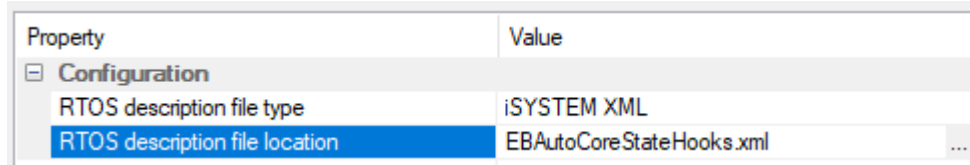
12. Select OSEK AUTOSAR OS.



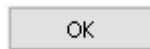
13. Specify a name, for example, *EB AutoCore OS*.



14. Select XML as RTOS description file type.



15. Select the XML file and click OK.



16. Make sure to load symbols to make the change active.



Next, the winIDEA profiler is configured to use the Profiler XML file, too. To do so execute the following steps.

8. Open the profiler configuration. Make sure it is the same configuration for which data tracing of the XML variables is configured.



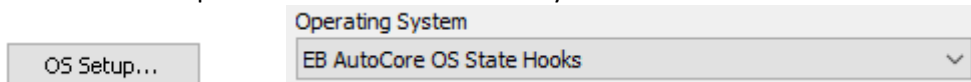
9. Select the hardware tab and make sure that the profiler is activated.



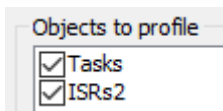
10. Change to the profiler tab and make sure that OS objects are selected.



11. Click on OS Setup and select the OS for which you have added the XML file.

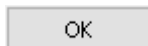


12. Select all tasks and ISRs you want to profile. (Again, only those objects for which the signaling variable is record will show up in the profiler timeline.)



- 13.

14. Confirm with OK.



15. Start a new trace recording.



4.5 winIDEA Profiler Visualization

4.5.1 Profiler Timeline

If the application is running you should see the OS objects in the profiler timeline as shown in Figure 10Figure 2.

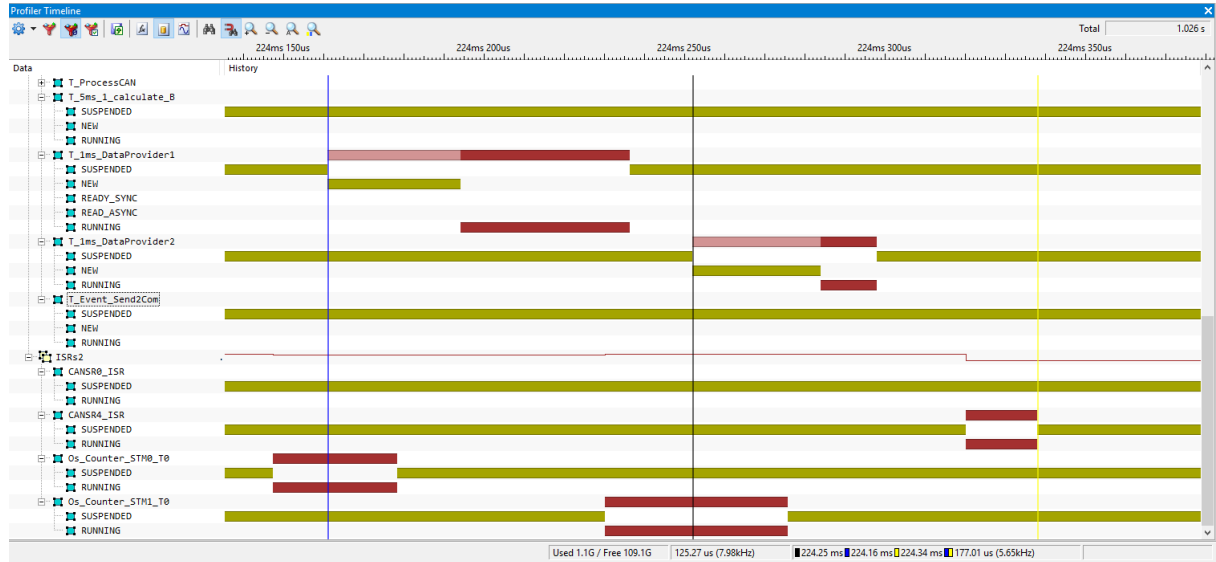


Figure 10: EB tresos AutoCore OS Task and ISR State Trace (via Instrumentation) in the winIDEA Profiler Timeline

5 Processor-specific Trace Configurations

5.1 Infineon AURIX MCDS

5.1.1 DAP Upload-While-Sampling (UWS)

The Infineon AURIX controller family supports two types of trace interfaces, the high-speed AURORA Gigabit Trace (AGBT) interface and the lower-speed Debug Access Port (DAP). Obviously the AGBT interface offers the highest trace bandwidth, but it also requires either the use of emulation adaptors or the application of high-frequency signal routing on the ECU board layout. Both solutions involve quite some effort. Therefore, many trace setups use the DAP interface not only for debug access but also for trace.

Scheduling analysis typically requires trace recording of several seconds. Therefore, it is required to use the DAP interface in a streaming mode. Such trace data streaming can be accomplished by iSYSTEM tools by means of its Upload-While-Sampling (UWS) operating mode, ideally running at the maximum DAP clock speed of 160MHz.

The following winIDEA workspace configuration are need to operate in UWS mode:

- The DAP operation should be set to the maximum performance depending on the target board layout (Mode: DAP Wide, Clock: up to 160MHz).
- The MCDS should be set to Tick time stamping (Time stamps Source: tick).
- The MCX trigger mode should be set to “never” (MCX.Action.trace_done: NEVER).

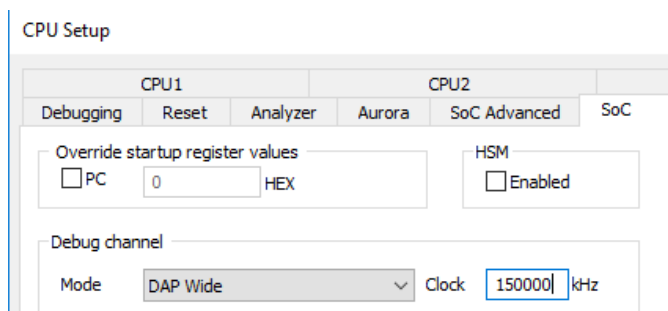


Figure 11: DAP Operation Setting for UWS mode. Up to 160MHz is supported.

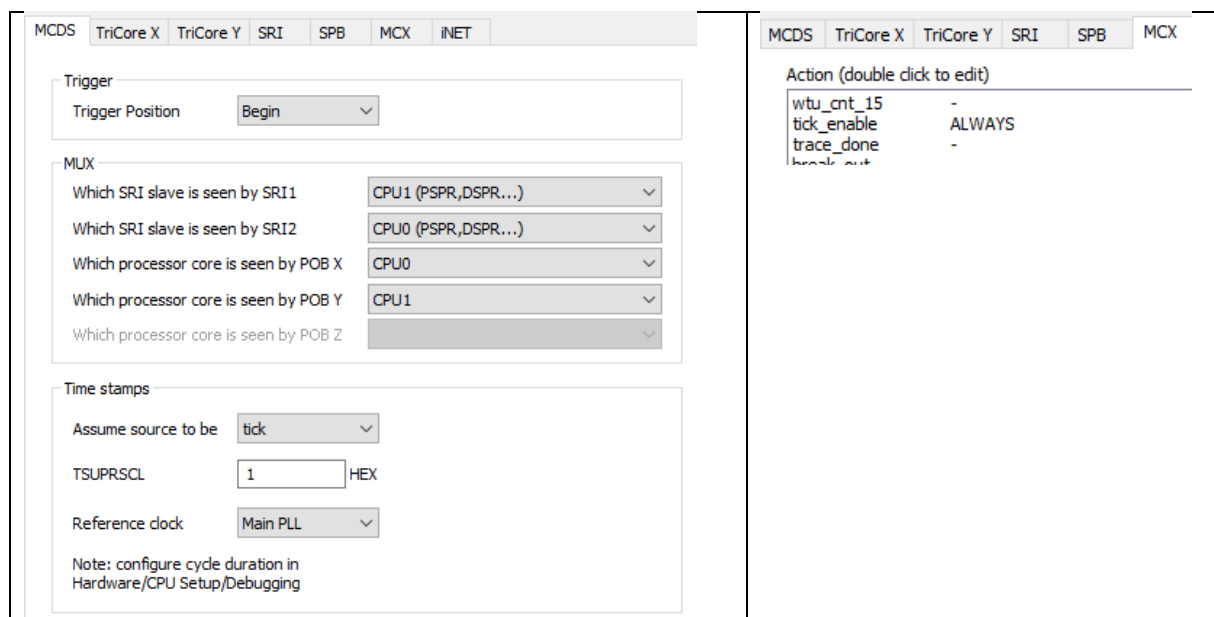


Figure 12: AURIX MCDS Settings for UWS mode

5.1.2 Data Trace by means of Fine Grain Comparator

Especially when using UWS it is very critical to reduce the trace message generation rate to its absolute minimum. The table below ranks the various OS trace concepts according to their trace bandwidth requirements.

| Trace Requirements | Bandwidth | Trace Concept |
|--------------------|-----------|--|
| Lowest | | Running Task / ISR2 Tracing |
| Medium | | Task / ISR2 State Tracing by means of Hook Instrumentation |
| Highest | | Task State Tracing without Instrumentation |

The concept of task state tracing without instrumentation requires the highest bandwidth as many task state variables need to be observed. If the on-chip trace logic only offers a relatively low number of address range comparators, the entire OS task state/control array needs to be observed, leading to the fact that also unnecessary data objects are being traced, wasting quite some trace bandwidth.

The AURIX MCDS offers a very useful feature, called Fine Grain Comparator, to observe a large number of individual data objects. This allows to focus data trace exactly to those objects relevant for task state trace. The underlying concept is a 4kByte RAM-based look-up table. Each bit of this RAM is basically linked to a byte of the address range that is supposed to be observed. So, the 4kByte look-up table RAM can cover 32kByte of “real” memory. If a bit location in the look-up table is set to 1, this means that the corresponding byte location will be traced, i.e. if the CPU performs a data transaction to this location, the data and address can be captured. This concept is ideally suited to cover the OS task state/control array and “pick out” just those data locations relevant to OS task state trace.

Figure 13 shows a sample MCDS trigger configuration in the winIDEA Analyzer.

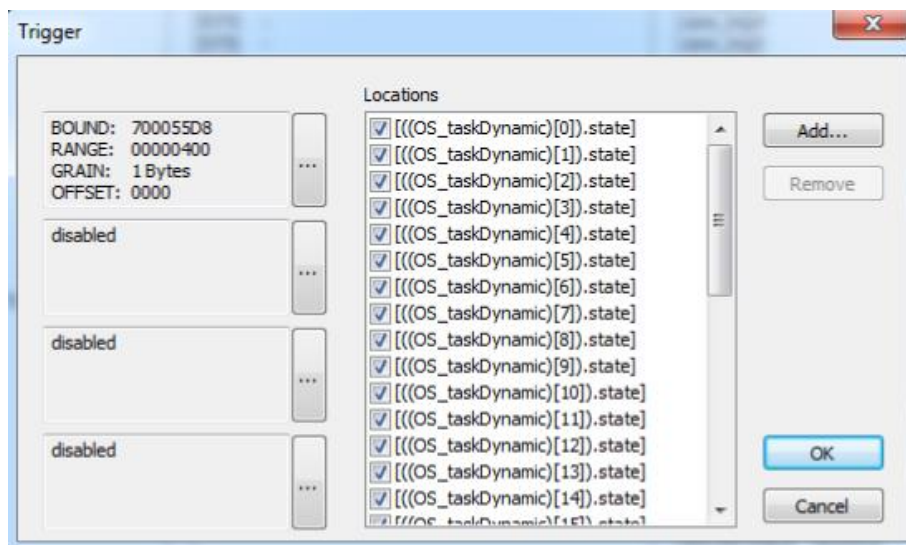


Figure 13: AURIX MCDS Fine-Grain Comparator Configuration to trace OS task state variables of the EB AutoCore OS

5.2 Renesas RH850 Software Trace

RH850 Software Trace allows signaling specific software events to an external hardware trace tool by means of the dedicated CPU instructions `DBTAG #imm10` and `DBPUSH Rx`. When the RH850 CPU executes any of these instructions it causes the on-chip Software Trace Module to generate a Software Trace message. These trace messages can either be stored in an on-chip trace RAM (if implemented) or it can be streamed out via the LPD4 debug port of the RH850 device. The advantage of the Software Trace streaming via the LPD4 interface is the fact that this functionality is available on any RH850 derivative. However, the user also has to be aware of its limitations, which are:

- Trace Bandwidth limitation of the LDP4 interface
- No support of multi-core Software Trace
- Requires code instrumentation

The `DBTAG` instruction can be used to create messages for values which are known at compile time, such as an index of function (Runnable) entries/exits. The `DBPUSH` instruction is suited for signaling events which are only known at run-time, such as OS task or ISR2 state changes.

By combining these approaches, it is possible to profile OS tasks, ISR2s and functions (Runnables). This means the EB tresos AutoCore OS hooks `OS_TRACE_STATE_TASK` and `OS_TRACE_STATE_ISR` are used to generate Software Trace messages via the `DBPUSH` instruction. Additionally, functions of interest can be traced via the `DBTAG` instruction.

Once the instrumentation is part of the application the trace data can be recorded via winIDEA and the resulting trace is visualized in the winIDEA profiler timeline. A dedicated winIDEA profiler XML file is required to make winIDEA aware of how to interpret the recorded data correctly.

5.2.1 RH850 Software Trace Macros

OS task state transitions are signaled by the OS by calling the `OS_TRACE_STATE_TASK` macro, as depicted in Listing 8. In the same fashion, OS ISR2 state transitions are signaled by the `OS_TRACE_STATE_ISR` macro.

```

/* ===== */
/* File name: Dbg.h */
/* Compiler: GHS */
/* ===== */

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

/* Parameters:
    StateId = Task Index
    oldValue = previous Task State (before state transition)
    newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_TASK(StateId,oldValue,newValue) \
    isystem_sft_taskstate( (newValue << 16) | StateId )

asm void isystem_sft_isr2state(val)
{
%reg val
    mov val, r11
    dbpush r11-r11
}

/* Parameters:
    isrId = ISR2 Index
    oldValue = previous Task State (before state transition)
    newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_ISR(isrId,oldValue,newValue) \

```

```
isystem_sft_isr2state( (newValue << 16) | IsrId )
```

Listing 8: Sample OS task and ISR2 state trace macro implementations using RH850 Software Trace

Alternatively the macros can also be defined as functions, as depicted in Listing 9. In some cases, this method is easier to realize. However, it requires an additional C source file to be included into the build process.

```
/* ===== */
/* File name: Dbg.h */
/* ===== */

/* Parameters:
   StateId = Task Index
   oldValue = previous Task State (before state transition)
   newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_TASK(StateId,oldValue,newValue) \
    isystem_sft_taskstate( (newValue << 16) | StateId )

/* Parameters:
   isrId = ISR2 Index
   oldValue = previous Task State (before state transition)
   newValue = next Task State (after the state transition) */
#define OS_TRACE_STATE_TASK(isrId,oldValue,newValue) \
    isystem_sft_isr2state( (newValue << 16) | IsrId)

/* ===== */
/* File name: isystem_sft.c */
/* Compiler: GHS */
/* ===== */

void isystem_sft_taskstate (int value)
{
    __asm volatile ( "mov %0, r10" :: "X" (value) : "r10");
    __DBPUSH(10, 10);
}

void isystem_sft_isr2state (int value)
{
    __asm volatile ( "mov %0, r11" :: "X" (value) : "r11");
    __DBPUSH(11, 11);
}
```

Listing 9: Sample OS Task and ISR2 state trace macro function implementations using RH850 Software Trace

5.2.2 RH850 Software Trace iSYSTEM Profiler XML

Listing 10 shows a sample iSYSTEM Profiler XML file for OS task and ISR2 trace using RH850 Software Trace.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>EB_AutoCore_Demo</Name>
  <NumCores>1</NumCores>
  <ORTI>os.orti</ORTI>

  <Types>
    <TypeEnum>
      <Name>Type_TASKSTATE</Name>
      <Enum><Name>SUSPENDED</Name> <Value>0</Value></Enum>
      <Enum><Name>QUARANTINED</Name> <Value>1</Value></Enum>
```

```

    <Enum><Name>NEW</Name>          <Value>2</Value></Enum>
    <Enum><Name>READY_SYNC</Name>   <Value>3</Value></Enum>
    <Enum><Name>READY_ASYNC</Name>  <Value>4</Value></Enum>
    <Enum><Name>RUNNING</Name>      <Value>5</Value></Enum>
    <Enum><Name>WAITING</Name>      <Value>6</Value></Enum>
  </TypeEnum>
</Types>

<Profiler>

  <Object>
    <Definition>TASKSTATE</Definition>
    <Description>Tasks</Description>
    <Type>OS:vs_SIGNAL_RUNNINGTASK</Type>
    <DefaultValue>NO_TASK</DefaultValue>
    <Name>TASKSTATE</Name>
    <Level>Task</Level>
    <Signaling>DBPUSH(10)</Signaling>
    <TaskState>
      <MaskID>0x0000FFFF</MaskID><MaskState>0x00FF0000</MaskState>
      <MaskCore>0xFF000000</MaskCore>
      <Type>Type_TASKSTATE</Type>
    </TaskState>
  </Object>
  <StateInfo><Name>QUARANTINED</Name><Property>Terminate</Property></StateInfo>
  <StateInfo><Name>SUSPENDED</Name><Property>Terminate</Property></StateInfo>
  <StateInfo><Name>RUNNING</Name><Property>Run</Property></StateInfo>
</TaskState>
</Object>

  <Object>
    <Definition>ISRSTATE</Definition>
    <Description>ISRs2</Description>
    <Type>OS:vs_SIGNAL_RUNNINGISR2</Type>
    <DefaultValue>NO_ISR</DefaultValue>
    <Name>ISRSTATE</Name>
    <Level>IRQ3</Level>
    <Signaling>DBPUSH(11)</Signaling>
    <TaskState>
      <MaskID>0x0000FFFF</MaskID><MaskState>0x00FF0000</MaskState>
      <MaskCore>0xFF000000</MaskCore>
      <Type>Type_TASKSTATE</Type>
      <StateInfo><Name>SUSPENDED</Name><Property>Terminate</Property></StateInfo>
      <StateInfo><Name>RUNNING</Name><Property>Run</Property>
    </TaskState>
  </Object>
</Profiler>
</OperatingSystem>

```

Listing 10: Sample iSYSTEM Profiler XML File for OS Task and ISR2 Trace using RH850 Software Trace


6 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. Before the BTF export is usable the iSYSTEM profiler XML file must be updated. The Profiler supports the export of tasks, ISRs, Runnables and signals. For tasks the following BTF mapping reference must be added to the `TaskState` node.

```
<BTFMappingType>Type_OS_to_BTF_State_Mapping</BTFMappingType>
```

The BTF mapping must be added to the `TypeEnum` section of the XML file. For the EB tresos AutoCore OS the mapping in can be used. The mapping is required to tell winIDEA which OS task state maps to which BTF state. No mapping is required for Runnables and signals. Note that this mapping is intended for Runnables and OS task (and ISR2) *state* tracing. No BTF export is possible for a *running* task/ISR trace.

Once the iSYSTEM Profiler XML is updated the following steps must be executed to export a BTF trace file.

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



4. This generates a BTF trace file which matches the profiler timeline as shown in Figure 14.

```
<!--
=====
      OS Task State Mapping used for the conversion from the EB tresos AutoCore
OS      specific Task State Model to the BTF Task State Model.

      OS Task State          BTF Task State
=====
-->
<TypeEnum>
  <Name>Type_OS_to_BTF_State_Mapping</Name>
  <Enum><Name>SUSPENDED</Name>  <Value>Terminated</Value></Enum>
  <Enum><Name>QUARANTINED</Name> <Value>Terminated</Value></Enum>
  <Enum><Name>NEW</Name>         <Value>Active</Value></Enum>
  <Enum><Name>READY_SYNC</Name>  <Value>Ready</Value></Enum>
  <Enum><Name>READY_ASYNC</Name> <Value>Ready</Value></Enum>
  <Enum><Name>RUNNING</Name>     <Value>Running</Value></Enum>
  <Enum><Name>WAITING</Name>     <Value>Waiting</Value></Enum>
</TypeEnum>
```

Listing 11: Mapping from EB tresos AutoCore OS task states to BTF task states.

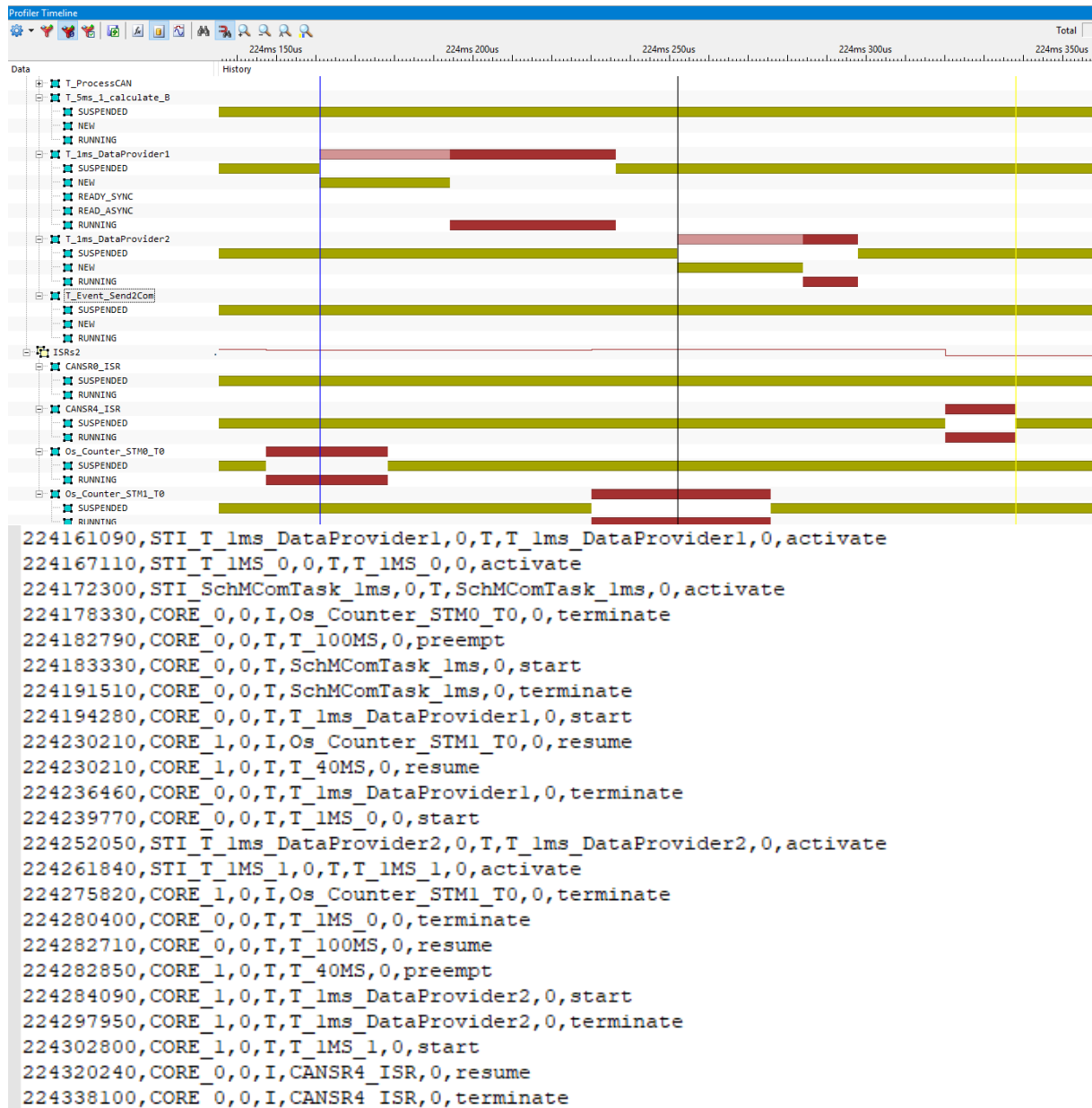


Figure 14: Sample BTF Export of OS tasks and ISR2

Note: The winIDEA Profiler also allows an export of Runnables. However, Runnable trace and profiling is beyond the scope of this Application Note. Please refer to the dedicated Application Note about Runnable trace with EB tresos AutoCore.

7 Inspectors

Inspectors are a winIDEA feature to analyze user-defined metrics in the winIDEA profiler timeline. It allows the creation of new Profiler objects, so called Inspectors, which can change their state depending on different events, such as state changes of other objects and timing parameters. This section demonstrates how inspectors can be used to cover certain advanced timing-analysis use-cases for the EB tresos AutoCore operating system.

7.1 Task Metric Analysis

Inspectors can be used to calculate the metrics defined in the AUTOSAR Timing Extensions Specification. Predefined Inspectors exist for a certain subset of those metrics. The Inspectors are defined in a generic way meaning the metrics are calculated for all tasks in the trace. There is no need to add a separate Inspector for each task and metric.

If you are interested in using those Inspectors, ask your iSYSTEM contact for the respective Inspectors JSON file which can be imported into the winIDEA Profiler to make the metrics available.

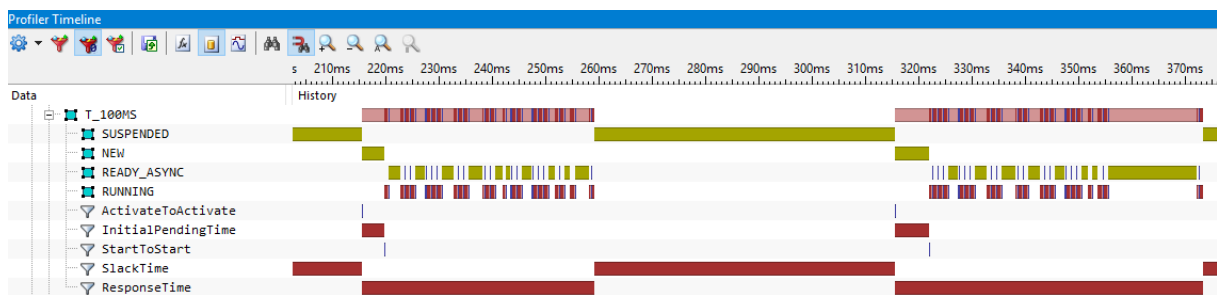


Figure 15: Inspectors to calculate Task Metrics for the Task T-100MS

For a further analysis of the Inspector objects, you can utilize the Properties view of the winIDEA Analyzer. To open the Properties view, select the desired object and press “Alt + Enter”.

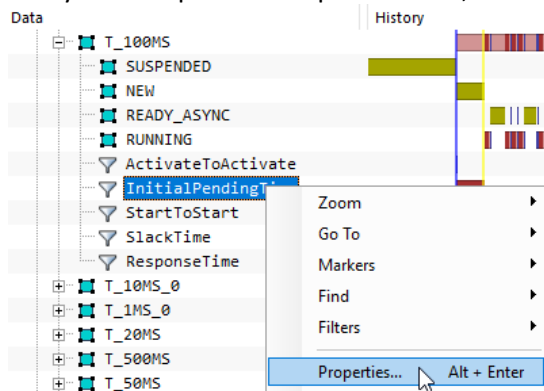


Figure 16: Opening the “Properties” View for a Profiler Inspector Object

For the task metric “ActivateToActivate”, the relevant object statistic is “Period”.

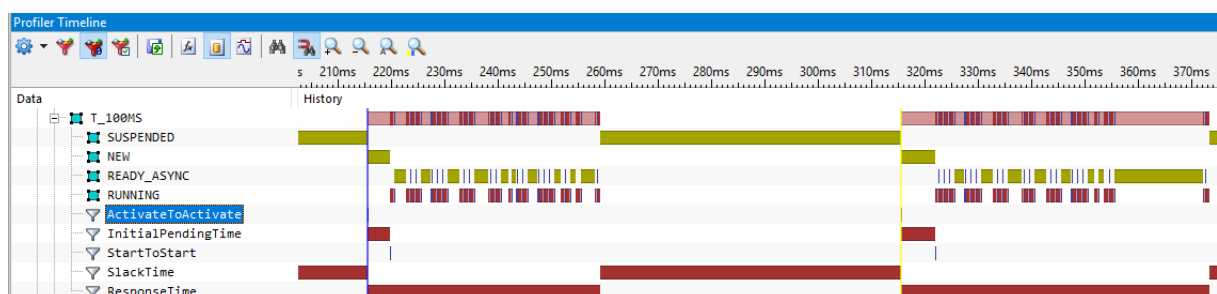


Figure 17: Inspector Object “ActivateToActivate” for the Task T_100MS

The Properties view provides the measurements for average, maximum and minimum period (i.e. “ActivateToActivate” time) along with the time (and link “->”) to its occurrence.

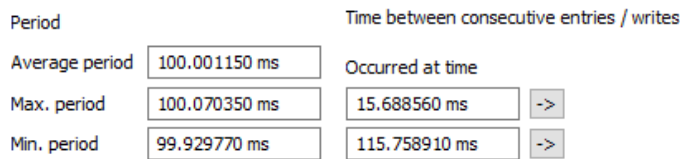


Figure 18: Period Properties for the “ActivateToActivate” Inspector Object

For the task metric “InitialPendingTime”, the relevant object statistic is “Net Time”.

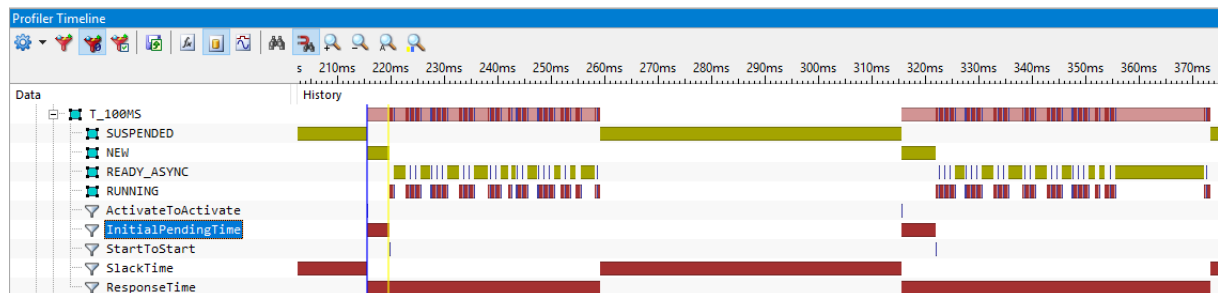


Figure 19: Inspector Object “InitialPendingTime” for the Task T_100MS

The Properties view provides the measurements for average, maximum and minimum Net Time (i.e. “InitialPendingTime”) along with the time (and link “->”) to its occurrence.

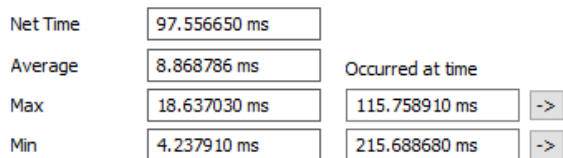


Figure 20: Net Time Properties for the “InitialPendingTime” Inspector Object

8 Technical Support

8.1 Online Resources

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

8.2 Contact

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

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

© iSYSTEM. All rights reserved.