ETAS RTA-OS Profiling

Update Date: 02/01/2021

Table of Contents

1 Introduction

In this document, we explain how to profile and analyze ETAS RTA-OS-based AUTOSAR applications' timing-behavior. 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 consulting our Introduction to AUTOSAR Classic Profiling application note. Then, return 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 ETAS RTA-OS Profiling webinar if 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.

	Running Task/ ISR	**Task State/ Running ISR**	**Runnables**
Program Flow Trace			Runnables via Program Flow Trace
Data Trace	Running Task/ISR2 via Data Trace	Task State/Running ISR via Data Trace	
Instrumentation Trace	Running Task/ISR2 via Software Trace		

Each section follows the same steps. First, you configure the OS to configure the trace objects available for profiling in winIDEA. 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 microcontroller's hardware trace module to record the OS and RTE objects. Depending on the use-case, winIDEA might automatically do this step, though manual configuration leads to a better understanding of the silicon's underlying trace logic. We explain manual trace configuration for some architectures in the *Generic Profiler/Trace Configuration* section.

2 Generic OS Configuration

This chapter explains the configuration steps shared by different use-cases. *Configure ORTI Support* is a mandatory step for all use-cases. The ORTI file provides iTCHi and winIDEA with information about the OS Tasks and ISRs. You do not have to use iTCHi for the *Running Task/ISR2 via Data Trace* use-case, but the other use-cases require to *Setup iTCHi*.

2.1 Configure ORTI Support

You can enable ORTI support in the ETAS RTA-OS configuration tool, as explained in their [user guide].

1. Use rtaoscfg to enable ORTI debugger support. You can find the configuration in the *Target Specific* settings.
2. 

3. Set the *ORTI/Lauterbach* setting to *TRUE*.
4. 

5. Now build the RTA-OS library, which instruments the Kernel with ORTI support. This step also generates the ORTI file with the name `<projectname>.orti`.

Remember the name of the ORTI file since you will need it for all use-cases.

2.2 Setup iTCHi

The iSYSTEM Trace Configuration Helper iTCHi helps the user automatically generate 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.

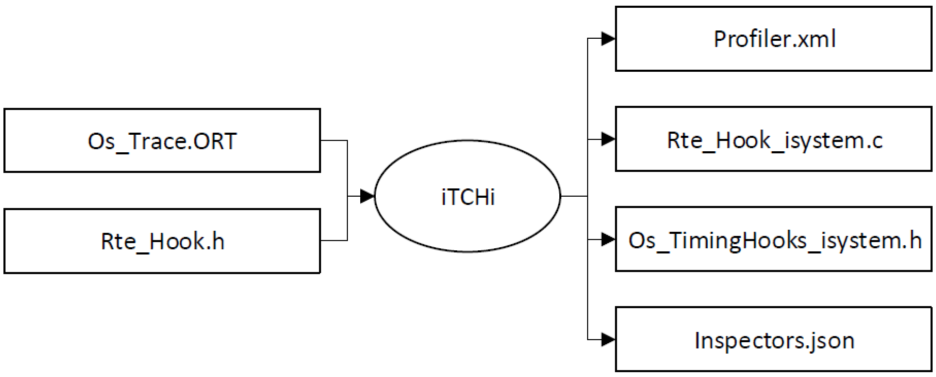


Figure 1: iTCHi helps the user to generate an iSYSTEM Profiler XML file and instrumentation code.

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.

3 Use-Cases

This section explains the configuration for the different Profiling use-cases in detail. You can also watch the iSYSTEM webinar about ETAS RTA-OS profiling for a visual guide to these use-cases. We will also include a link to the webinar with the specific timestamp for use-cases covered by the webinar.

3.1 Running Task/ISR2 via Data Trace

This section describes how to record a trace of the currently running OS Task and ISR2. An explanation of this use-case is also available as a video. The ORTI file specifies two attributes, `RUNNINGTASK` and `RUNNINGISR2`, for each core of the application. These attributes point to global variables that we can record via data-trace. The winIDEA analyzer can then profile the Running Task/ISR based on the trace messages and the information in the ORTI file. *Listing 1* shows these attributes for a single-core application and *Listing 2* for a multi-core application.
After generating the ORTI file, as explained in *Configure ORTI Support*, you add this file to winIEA as described in the section *Configure OS/RTE Profiling*. Make sure you select *ORTI* as file description type and point winIDEA to the correct ORTI file for your project, as depicted in *Figure 2*. Next, open the analyzer, enable profiling OS objects, and select Tasks and ISRs2 under OS setup.
Start a new recording by pressing the green play button. You might get an error message if winIDEA is not able to configure the data-trace manually. In this case, find the `RUNNINGTASK` and `RUNNINGISR2` variables in the ORTI file and set a data-trace for them manually. You can also find the signaling variables by looking at the objects in the OS setup dialog, as shown in *Figure 3*. Once you have configured data-trace for these two variables, you can start another recording. The result should be a running Task/ISR trace, as shown in *Figure 4*.

```
OS XY
{
  RUNNINGTASK = "Os_RunningTask";
  RUNNINGISR2 = "Os_RunningISR";
}
```
Listing 1: ORTI Attributes RUNNINGTASK and RUNNINGISR2 for a single core application.

```
/* OS information for Core0 */
OS Core0_ProjectName {
  RUNNINGTASK = "Os_ControlledCoreInfo[0U].RunningTask";
  RUNNINGISR2 = "Os_ControlledCoreInfo[0U].RunningISR";
  /* lines removed for clarity */
};

/* OS information for Core1 */
OS Core1_ProjectName {
  RUNNINGTASK = "Os_ControlledCoreInfo[1U].RunningTask";
  RUNNINGISR2 = "Os_ControlledCoreInfo[1U].RunningISR";
  /* lines removed for clarity */
};
```
Listing 2: ORTI Attributes RUNNINGITASK and RUNNINGISR2 for a multi-core application.

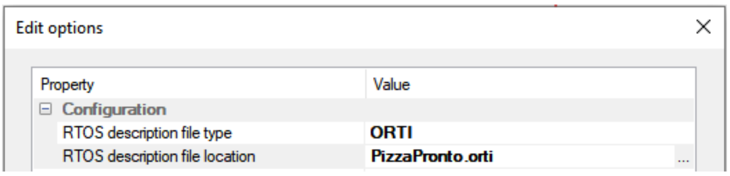


Figure 2: Operating system configuration for Running Task/ISR profiling of the application with the curious name PizzaPronto.

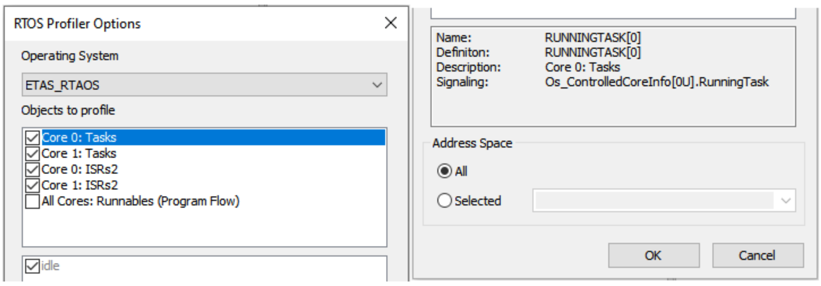


Figure 3: The RTOS Profiler Options menu shows the profiler objects defined in the ORTI file.

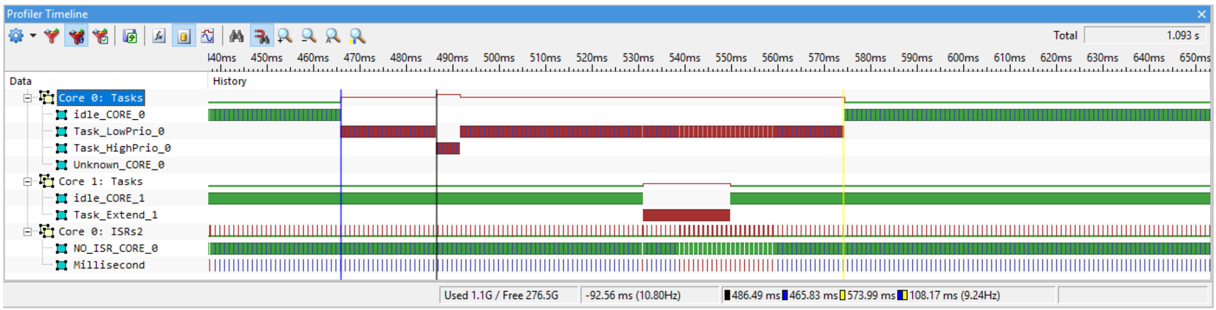


Figure 4: Running Task/ISR2 recording in the winIDEA Profiler Timeline.

3.2 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 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. For a visual guide to this use-case, watch our webinar.

Figure 5 shows the task state model for the ETAS RTA-OS. ORTI has an attribute to signal the current state of each task, as depicted in *Listing 3*. By tracing write accesses of each variable that is part of the state expression, we can reconstruct the states via so-called winIDEA Analyzer Inspectors. Inspectors are unique Profiler objects that change their state based on the value of other entities in the timeline. Our trace configuration helper iTCHi can automatically generate the Inspectors, and you only have to make sure that the relevant variables are part of the profiler timeline.

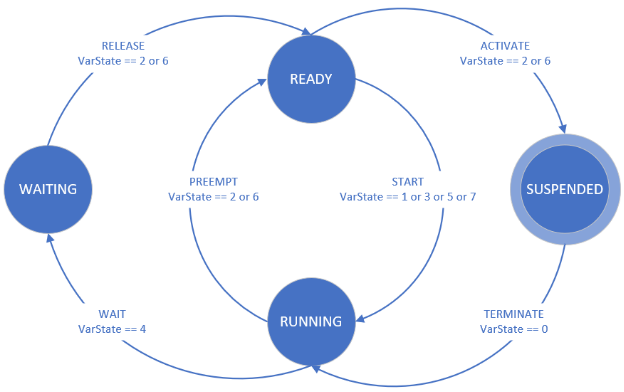


Figure 5: ETAS RTA-OS task state model and state transition. The number for each change shows the value the state variable takes for each state.

```
TASK Task_LowPrio_0 {
  /* additional attributes removed for clarity */
  STATE = "(((Os_ControlledCoreInfo[0U].RunningTask == Os_const_tasks[0]) * 1) & 1)
          + ((Os_ControlledCoreInfo[0U].ReadyTasks.p0 & 0x1) << 1)
          + ((Os_ControlledCoreInfo[0U].WaitingTasks.p0 & 0x1) << 2)";
};
```

Listing 3: Task object ORTI definition, including STATE attribute.

```
{
    "orti_file": "projectname.orti",
    "profiler_xml_file": "profiler.xml",
    "task_state": {
        "task_to_core_mapping": {
            "WriteCode": 0,
            "OrderPizza": 0,
            "EatPizza": 1
        }
    },
    "task_state_inspectors": {
        "constant_variables": {},
        "default_state": "SUSPENDED",
        "inspectors_file": "TaskStateInspectors.json",
        "parent_area_template": "Data/Core {core_id}: Tasks/{task_name}"
    }
}
```

Listing 4: iTCHi configuration for Task State/running ISR Profiling. Manual specification of the task to core mapping is mandatory for multi-core applications is compulsory for multi-core systems. For single-core microcontrollers, iTCHi automatically defaults to 0.

Start the configuration by setting up iTCHi, as explained in the *Setup iTCHi* section. Next, update the iTCHi configuration, as shown in *Listing 4*. There is a `task_to_core_mapping` attribute to specify the mapping from tasks to cores. For single-core systems, you can simply remove the content from this part so that you have empty curly brackets. However, for multi-core applications, you have to specify the mapping for each task manually. For tasks not listed in this section, iTCHi automatically defaults to *0*. The mapping is necessary for the Inspectors to reference the correct Profiler objects. Unfortunately, there is no way for iTCHi to figure this out automatically.
After specifying the mappings, make sure that the configuration points to the correct ORTI and Profiler XML file. 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, configure a data-trace for all variables that are part of the task state expression and for the running ISR variable. For *Listing 3,* you would set data-trace for these variables:

```
Os_ControlledCoreInfo[0U].RunningTask
Os_ControlledCoreInfo[0U].ReadyTasks.p0
Os_ControlledCoreInfo[0U].WaitingTasks.p0
Os_ControlledCoreInfo[0U].RunningISR
```

Of course, the variables may be different for your application. Make sure you search for the Task STATE attributes and select the correct variables. If you are using a Cortex-M based device, like Traveo-II or S32K14x, refer to *Configure Cortex-M Data Trace* for instructions on how to configure data-tracing. For other devices, consult our *Online Resources*.
Once you have configured trace recording for the variables, you must tell the Profiler to display them in the data areas timeline. The Inspectors reference these areas to reconstruct the task states. Open the profiler configuration and add the data areas, as shown in *Figure 7*. This configuration is a little bit tricky, as you can see in *Figure 6*. The description is the string referenced in the ORTI file, and you must copy it so that the Inspectors can find it. The location is the name of the variable, as referred to by winIDEA. Simply search for the variable in the symbol browser and select it.

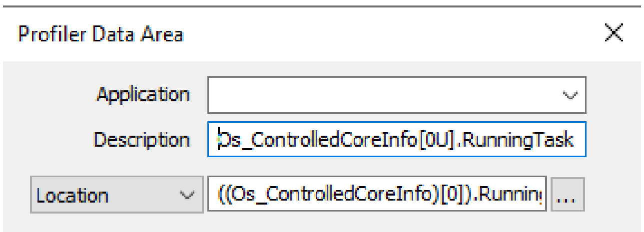


Figure 6: Data area for running task variable. The location references the variable in winIDEA syntax with braces. The description references the variable in the same way as in the ORTI task state expression.

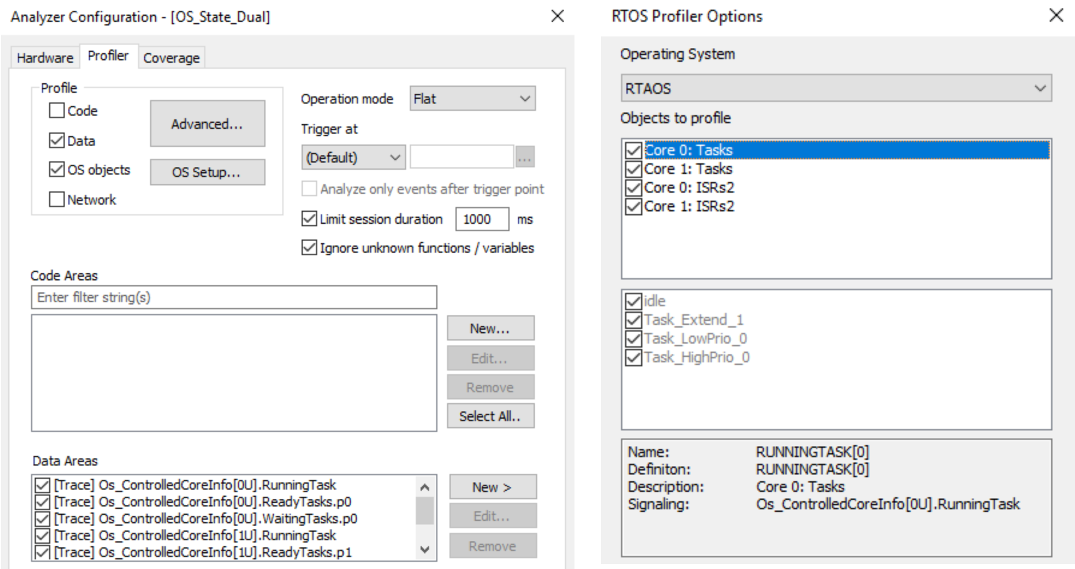


Figure 7: Task state profiling requires adding all variables that are part of the complex expression explicitely to the winIDEA data area. It is also necessary to select the Task and ISR objects under OS setup and to enable OS profiling.

Wit the data areas in place, you are now ready to start a recording by pressing the green button. If the application is running, you should see the OS objects in the profiler timeline, as shown in Figure 8. Under the OS objects, there should be additional areas showing the different states mentioned before in *Figure 5*. If there is no data, check the trace window if it shows write accesses to the signaling variables. Also, make sure that the data section of the profiler timeline is visible. In case you see the tasks and ISRs, but not the task states, check the output window after the trace recording. It should display an error if the Inspectors are unable to reference a variable required for the analysis.

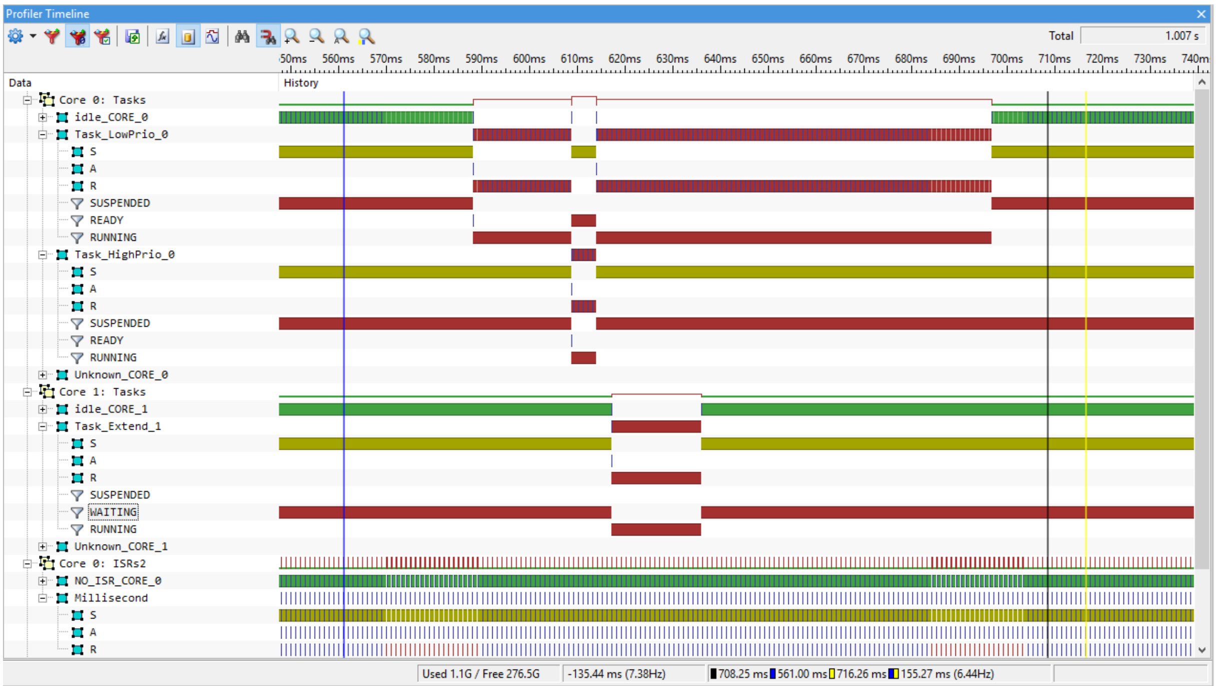


Figure 8: Task State/Running ISR Profiling based on Inspectors in the winIDEA Profiler Timeline.

3.3 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 9*, 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 5*. 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 9*.

```
{
  "orti_file": "projectname.orti",
  "profiler_xml_file": "profiler.xml",
  "runnable_program_flow": {
    "runnables": [
       "Runnable_Core1_100ms",
       "Runnable_Core1_1ms"
    ]
  }
}
```
Listing 5: iTCHi configuration for profiling Runnables with program-flow-trace.

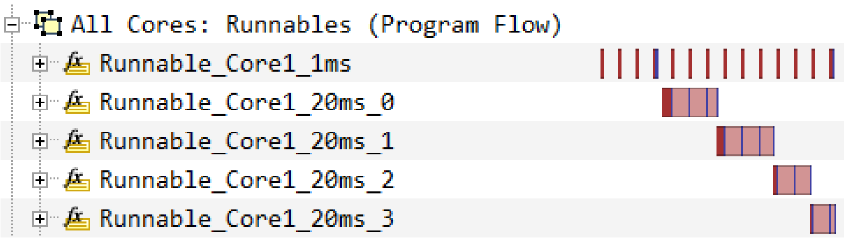


Figure 9: Runnable profiling based on program-flow-trace. The Profiler displays Runnables in the data area. Make sure to unselect "hide areas with no activity."

3.4 Running Task/ISR2 via Software Trace

You can use the ETAS RTA-OS tasks-hooks to add instrumentation on targets that do not have data-trace available.

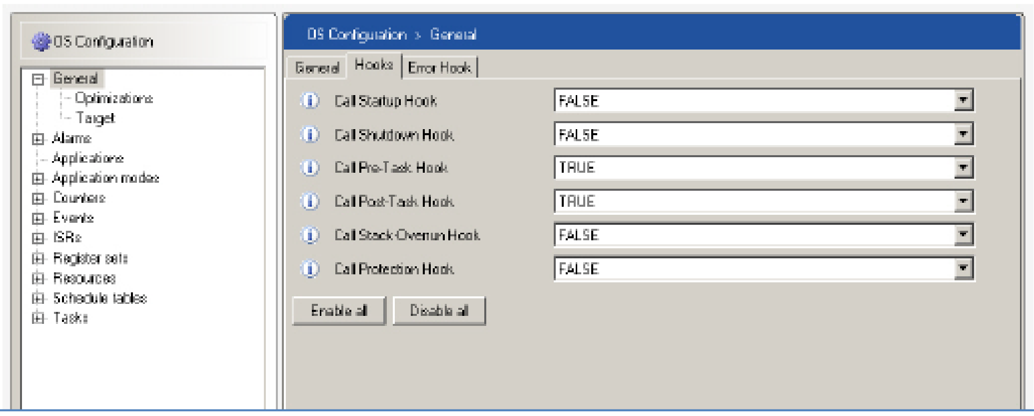
4.13 Pre and Post Task Hooks

Suppose that you need to execute some code before each task starts and/or after each task ends, for example to profile a trace of execution. You can do this using the PreTask and PostTask hooks provided by AUTOSAR OS.

The PreTask Hook is called by RTA-OS whenever a task moves into the running state. This means that the PreTask Hook will also be called whenever a task is resumed after preemption.

The PostTask Hook is called by RTA-OS whenever a task moves out of the running state. The PostTask Hook will be called when the task terminates and each time a task is preempted.

You can activate the hooks by setting the Pre- and Post-Task Hook attributes to *TRUE*, as shown on the following screenshot.



Once you regenerate the code, the pre and post-task-hooks should be part of the generated OS code. The following screenshot shows the template for the PreTaskHook.

```
#ifdef OS_PRETASKHOOK
FUNC(void, OS_CALLOUT_CODE) PreTaskHook (void)
{
  /* Your code */
}
#endif /* OS_PRETASKHOOK */
```

You can now instrument the hooks with the following code. The first two blocks show the hook-implementation, and the fifth and sixth block defines the instrumentation function. For ISRs, you have to use the ETAS-specific hooks Os_Cbk_ISRStart and Os_Cbk_ISREnd, as shown in blocks three and four in the listing. Please be aware that the software trace functions with the DBPUSH-instruction are compiler-specific. You might have to update them depending on the compiler you are using. Please, contact your compiler vendor if you run into issues.

```
FUNC(void, OS_CALLOUT_CODE) PreTaskHook(void)
{
  TaskType CurrentTask;
  GetTaskID(&CurrentTask);
  isystem_profile_task((int)CurrentTask);
}

FUNC(void, OS_CALLOUT_CODE) PostTaskHook(void)
{
    isystem_profile_task((int)0);
}

FUNC(void, OS_CALLOUT_CODE) Os_Cbk_ISRStart(ISRType isr)
{
  isystem_profile_isr2((int)isr);
}

FUNC(void, OS_CALLOUT_CODE) Os_Cbk_ISREnd(ISRType isr)
{
  isystem_profile_isr2((int)0);
}

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

asm void isystem_profile_isr2(val)
{
%reg val
  mov val, r11
  dbpush r11-r11
}
```

The last step is to create a Profiler XML file manually. You must copy the XML file into the same directory as the ORTI file. Note that the XML file references the ORTI file *RTAOS.orti*. The ORTI file reference is necessary so that the winIDEA Profiler can resolve the Task/ISR name to ID mapping.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>OSEK</Name>
  <NumCores>1</NumCores>
  <ORTI>RTAOS.orti</ORTI>
  <Types>
  </Types>
  <Profiler>
    <Object>
      <Name>RUNNINGTASK</Name>
      <Definition>RUNNINGTASK</Definition>
      <Description>RUNNINGTASK</Description>
      <Type>OS:RUNNINGTASK</Type>
      <Signaling>DBPUSH(10)</Signaling>
      <Level>Task</Level>
    </Object>
    <Object>
      <Name>RUNNINGISR2</Name>
      <Definition>RUNNINGISR2</Definition>
      <Description>RUNNINGISR2</Description>
      <Type>OS:RUNNINGISR2</Type>
      <Signaling>DBPUSH(11)</Signaling>
      <Level>IRQ3</Level>
    </Object>
  </Profiler>
</OperatingSystem>
```

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.

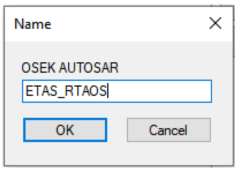
4.1 Configure OS/RTE Profiling

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

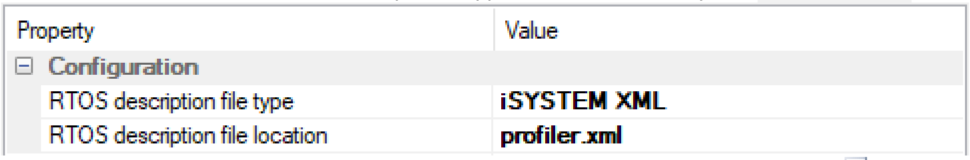
1. Add the XML file to winIDEA.
    a. Go to Debug, Operating Systems:

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

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

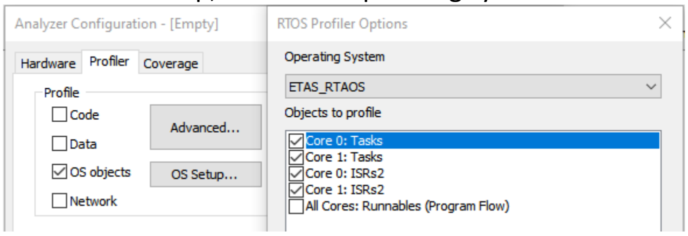
    

Property	Value
⊟ Configuration	
RTOS description file type	**iSYSTEM XML**
RTOS description file location	**profiler.xml**

    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:

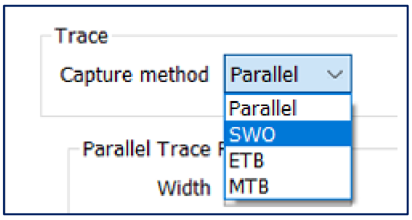
    c. In the menu, select Profiler, unselect Coverage, and choose Automatic.
    d. Open the new configuration via the hammer-icon:
    e. Make sure Profiler is active in the hardware tab: ☑ Profiler
    f. Switch into the profiler tab, unselect all options except OS objects: ☑ OS objects
    g. Click on OS Setup, select the operating system and the objects you want to profile:

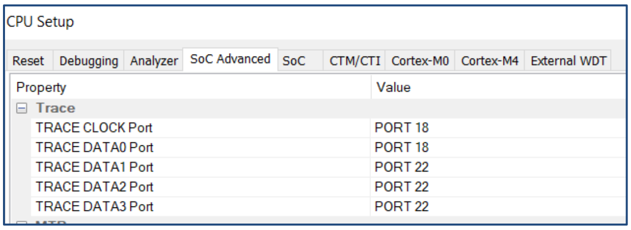
3. You are now ready to start profiling by clicking the green play button in the analyzer. If you have multiple objects, winIDEA might give an error saying there are too many data areas. When you get this error, you have to configure the hardware trace manually under the Hardware-tab. You can find out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

4.2 Configure Cortex-M Data Trace

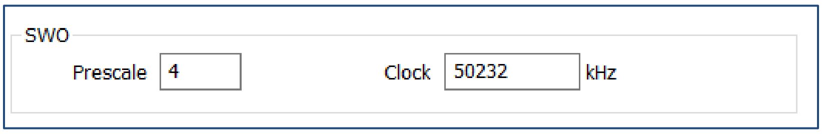
In this section, we explain how to configure data-trace for Cortex-M based microcontrollers, such as Traveo-II and S32K. Before you can start tracing on a Cortex-M based device, you have to select the trace interface, either single wire output (SWO) or parallel trace, under Hardware > CPU Options > SoC:



For parallel trace, you must configure the trace ports depending on your PCB layout. Also, make sure to run the trace line calibration under Hardware > Tools > Trace Line Calibration, while the target is running.

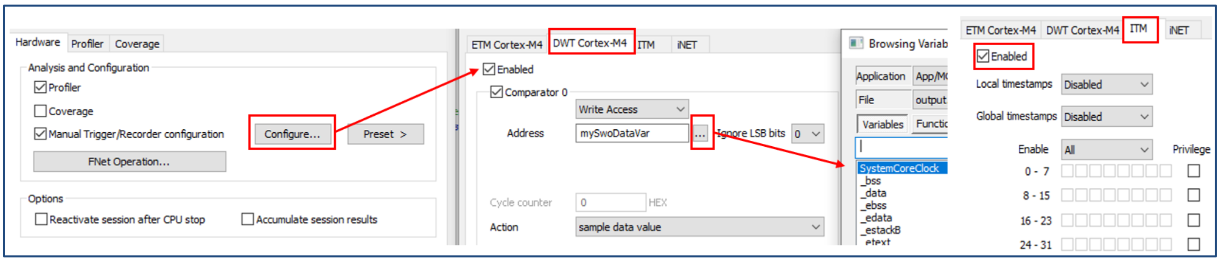


In case you are using the SWO trace, configure the SWO Prescaler and CPU Clock, so that (Clock / (Prescale + 1)) is smaller than 10MHz.



You are now ready to start profiling. Go to the winIDEA analyzer and create a new manual analyzer configuration.

To configure data-trace for a specific variable, open the trace configuration (by click on the hammer symbol), enable manual trigger/recorder configure, and select configure. Under DWT (data watch trace), enable DWT, then use the first available comparator. Choose *Write Access* and specify the variable you want to record. Next, select *sample data value* under *Action*. Next, make sure also to enable the ITM (instruction trace microcell) since it is necessary to record DWT trace messages. Use the four comparators to record all variables required for task state tracing, as shown in *Figure 10*. Do not forget to add the variables to the data area, as explained in *section 3.2*.



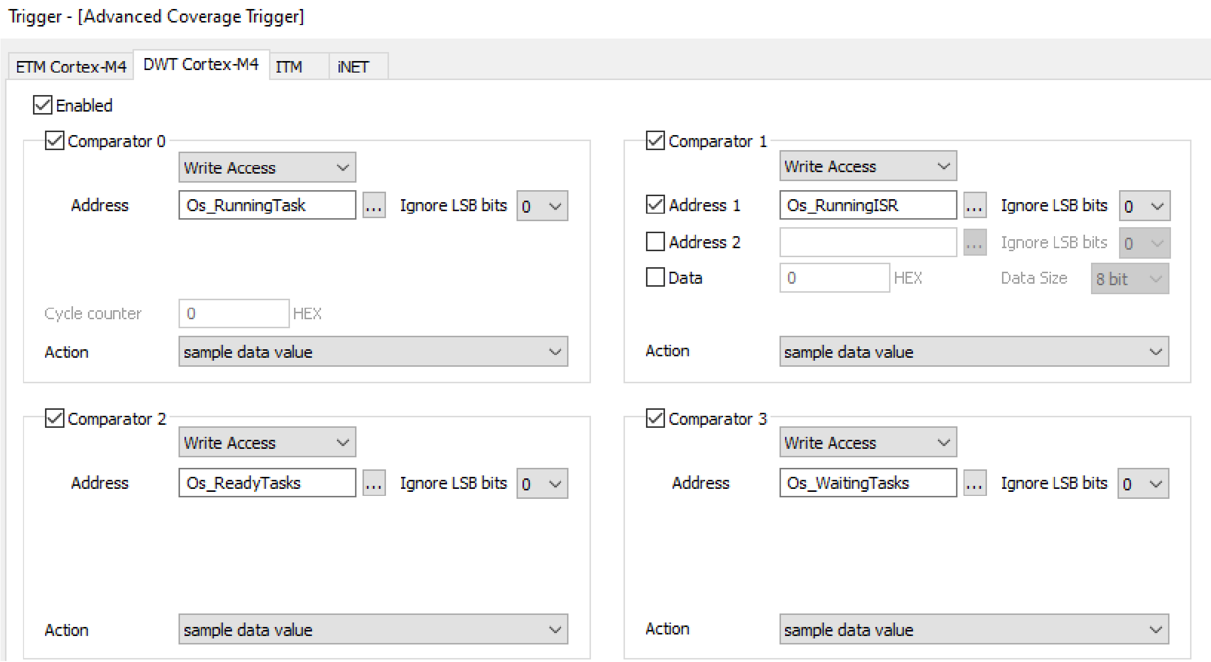


Figure 10: Recording of all variables necessary for task state/running ISR profiling via the DWT module.

4.3 Configure 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 6*. 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 11*.

```
<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 6: Mapping from thread states to BTF state transitions. This mapping is required for the winIDEA Profiler to execute a correct BTF export.

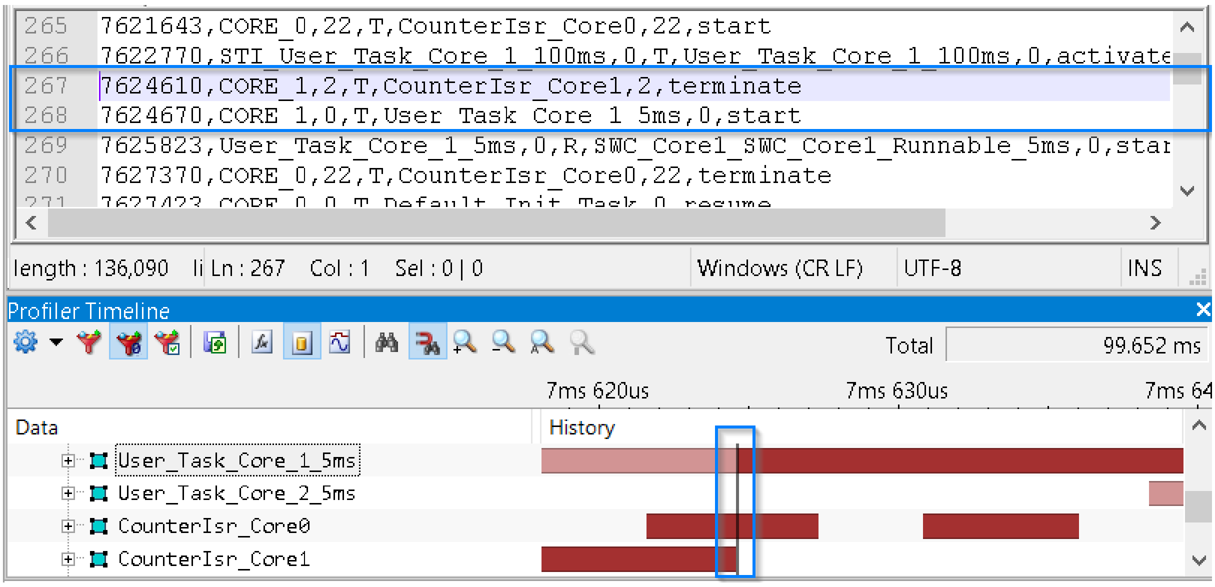


Figure 11: The winIDEA Profiler can export to the BTF format. Multiple timing tool vendors support BTF.

5 Technical Support

5.1 Online Resources

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

5.2 Contact

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