Elektrobit EB tresos Runnable Profiling

Publish Date: 08/09/2018

Table of Contents

1 Introduction

This document describes how to use the winIDEA Analyzer for the timing analysis of AUTOSAR Runnables of the two Elektrobit AUTOSAR solutions “EB tresos AutoCore” or “EB tresos Safety”.

1.1 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.
For more information about OS profiling, please refer to the dedicated Application Notes “Elektrobit EB tresos AutoCore OS Profiling” and “Elektrobit EB tresos Safety OS Profiling”.

1.2 Runnable Tracing

In general, there are three trace measurement techniques: software, hybrid and pure hardware based tracing. All techniques are meant to examine the runtime behavior of a system. In this document we focus on hardware and hybrid trace techniques. These trace techniques rely 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.

In general, the iSYSTEM Analyzer supports two concepts for Runnable profiling.

- Runnable Profiling by means of on-chip Program Trace
- Runnable Profiling by means of Data Trace combined with Instrumentation of the RTE Virtual Function Bus (VFB) Tracing Hooks

Which one is the most suitable depends on the AUTOSAR environment and the processor in use.

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 Instrumented Data Trace

Pure hardware 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 (i.e. RTE) to write the desired information into a dedicated variable. This variable can then be observed efficiently by hardware data trace. Such a trace approach is also called “hybrid”, as it combines the advantages of instrumentation, i.e. the extraction of only the relevant information already on-chip and the performance of the on-chip hardware trace logic.

2 Runnable Profiling by means of Program Trace

This concept is based on full (unconditional) program trace. The program trace logic of the processor emits a trace message whenever the core executes specific branch instructions. Based on those trace messages the trace analyzer can reconstruct the entire flow of the program code, down to each individual assembly instruction (i.e. instruction trace). The reconstructed program flow can subsequently be used by the profiler to reconstruct a function-level profile of the entire executed program.
Some of these functions can be treated as AUTOSAR Runnables, i.e. the profiler creates a dedicated profiler object for those functions the user has declared as Runnables.

The advantage of this concept is that Runnable Profiling can be done on unmodified/non-instrumented code. However, the disadvantage is that the trace message generation rate of a full program trace often exceeds the available bandwidth of the processor trace interface (such as a parallel Nexus trace port, ARM CoreSight TPIU or AURIX DAP interface).

2.1 iSYSTEM Profiler Configuration

Typically, run-time analysis on AUTOSAR based applications is done by means of the ORTI file. However, the ORTI file only covers objects of the AUTOSAR OS, such as tasks and ISR2s.
AUTOSAR Runnable are implemented within the RTE, i.e. are outside the scope of the ORTI file.
iSYSTEM uses a proprietary XML file format to describe the tracing/profiling related aspects of OSes, hypervisors or AUTOSAR RTE objects which are subject to timing/scheduling to the winIDEA Profiler (iSYSTEM Profiler XML file).
Such a Profiler XML file can be used to basically extend the ORTI file, with RTE related information such as Runnables.
The figure below shows a sample profiler XML file used for Runnable profiling based on program trace.

```
    <TypeEnum>
        <Name>Type_RUNNABLE_MAPPING</Name>
        <Enum><Name>SWC_ModifyEcho_ModifyEcho</Name>     <Value>&amp;amp;SWC_ModifyEcho_ModifyEcho</Value></Enum>
        <Enum><Name>SWC_CyclicCounter_SetCounter</Name> <Value>&amp;amp;SWC_CyclicCounter_SetCounter</Value></Enum>
        <Enum><Name>SWC_CyclicCounter_Cyclic</Name>      <Value>&amp;amp;SWC_CyclicCounter_Cyclic</Value></Enum>
    </TypeEnum>

</Types>

<Profiler>

   <Object>
     <Definition>RUN</Definition>
     <Description>Runnables</Description>
     <Type>Type_RUNNABLE_MAPPING</Type>
     <Level>Runnable</Level>
     <Expression></Expression>
     <Signaling>Exec</Signaling>
     <DefaultValue>0</DefaultValue>
   </Object>
```

Figure 1: Sample iSYSTEM Profiler XML File for Runnable Profiling by means of Program Trace

In the upper section (“Types”) an enumeration type is defined (“Type_Runnable_MAPPING”), which maps a Runnable name, displayed in the winIDEA Profiler to its corresponding symbolic name in the ELF file.
Optionally, the ELF file name can be attached to the function symbolic name, e.g. “SWC_ModifyEcho_ModifyEcho,,TRICORE_TC27_simple_demo_can_rte.elf”.

In the lower section (“Profiler”), a new profiler object is created. It is defined as a “RUN” object using “Exec” signaling, telling the profiler that this object is used for “Runnable Profiling” based on Function Profiling in Entry/Exit mode of the analyzer. The “Type” tag tell the profiler to use the enum type “Type_Runnable_MAPPING” for the Runnable naming.

In a typical use-case, these XML file section will actually be part of a more comprehensive Profiler XML file also used for OS task (and ISR) state tracing.

The trace hardware of the processor needs to be configured for unconditional program trace. Please refer to Section 0 “

Hardware Trace Configurations" for the processor-specific settings.

The Profiler only needs to be configured for OS objects profiling. Code profiling is not needed.

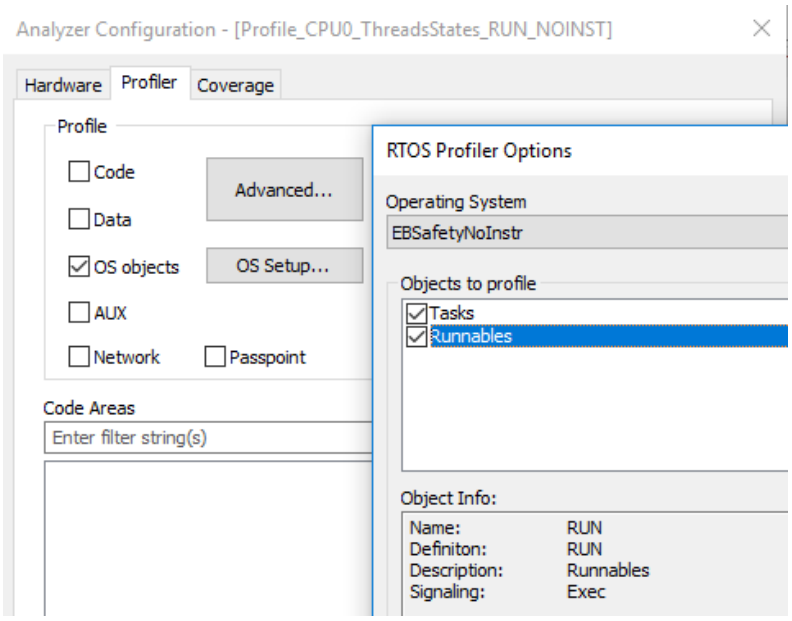
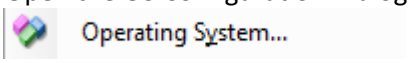


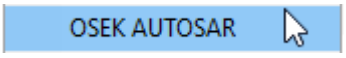
Figure 2: Profiler Configuration for Runnable Profiling based on Program Trace

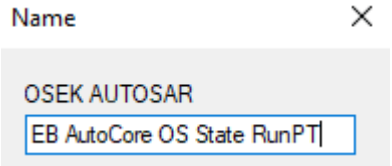
But first, the Profiler XML file has to be added to the winIDEA workspace and then the profiler is configured to use this information. To add the XML file to the workspace execute the following steps.

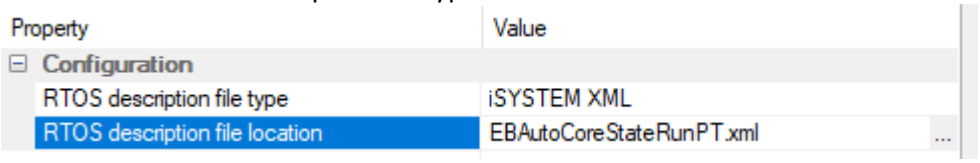
1. Open the Debug menu.
   Debug

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.
   

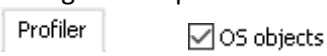
7. Select the XML file and click OK.
   

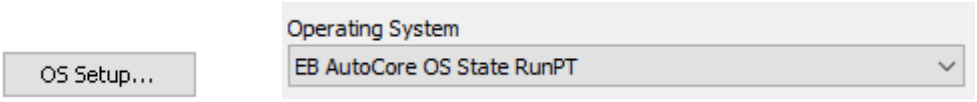
8. Make sure to load symbols to make the change active.
   

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

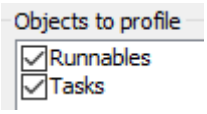
1. Open the profiler configuration. Make sure it is the same configuration for which data tracing of the XML variables is configured.


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. 

7. Confirm with OK.


8. Start a new trace recording.


2.2 Hardware Trace Configurations

For most processors the on-chip trace logic needs to be configured for unconditional program trace.

2.2.1 ARM ETM (e.g. Cortex R7)

The ARM Embedded Trace Macro Cell (ETM) allows for efficient instruction tracing. Whether ETM instruction trace can be applied for typically long-term OS and Runnable profiling mainly depends on the available bandwidth of the trace port, such as a parallel trace port (TPIU) or a High-Speed Serial Trace Port (HSSTP).
Figure 3 shows an unconditional instruction trace configuration of an ETM version 4 (ETMv4), implemented for instance in an ARM Cortex R7 device.

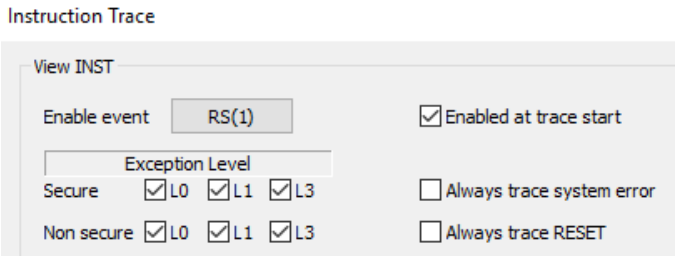


Figure 3: Program (i.e. Instruction) Trace enable on a Cortex R7 ETMv4

2.2.2 PowerPC

The on-chip trace hardware of PowerPC based architectures follow the Nexus Debug & Trace standard. They support a very efficient variant of Nexus program trace messages, so-called “Branch History Messages”. This trace messaging type is very suitable for function, i.e. Runnable, profiling.
Figure 4 shows an unconditional program trace configuration of PowerPC based device.

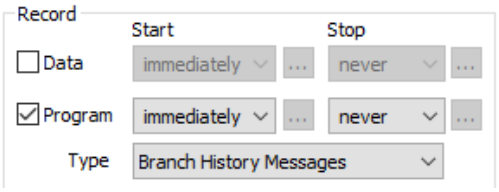


Figure 4: Program Trace enable on a PowerPC Nexus Trace Module

2.2.3 Infineon AURIX MCDS

The AURIX MCDS on-chip trace logic supports two approaches for function (i.e. Runnable) trace.

Program Trace
This is a full instruction trace, based on trace message generation upon the execution of specific branch instructions. This approach requires a high trace interface bandwidth, e.g. AGBT interface, especially when used on multi-core AURIX devices.
Figure 5 shows how to enable unconditional program trace of AURIX MCDS Processor Observation Block X (POB Action.ptu_enable = ALWAYS).

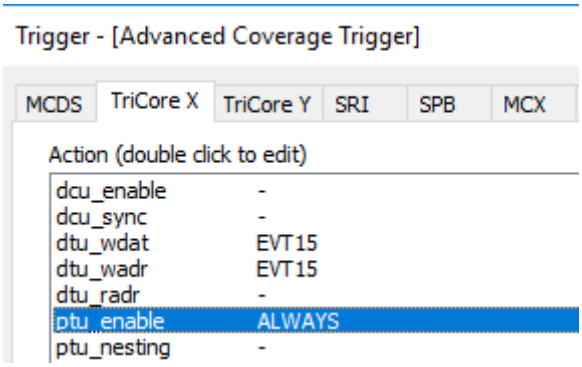


Figure 5: AURIX MCDS Full Program Trace enable

Compact Function Trace (CFT)
This is a Function trace, based on trace message generation upon function call/return instruction execution. This approach requires a consistent function call/return sequence for all functions. This may not always be the case due to compiler optimizations such as function chaining which replaces a function call instruction by a jump instruction.
However, CFT required much less trace interface bandwidth compared to full instruction trace.

Figure 6 shows how to enable unconditional Compact Function trace of an AURIX MCDS Processor Observation Block X (POB Action.ptu_nesting = ALWAYS).

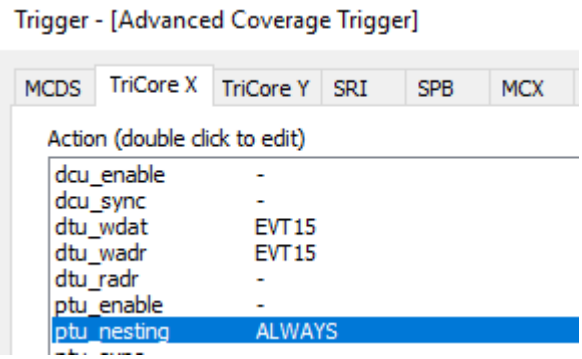


Figure 6: AURIX MCDS Compact Function Trace enable

2.3 winIDEA Profiler Visualization

2.3.1 Profiler Timeline

If the application is running you should see the OS objects in the profiler timeline as shown in Figure 7. 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.

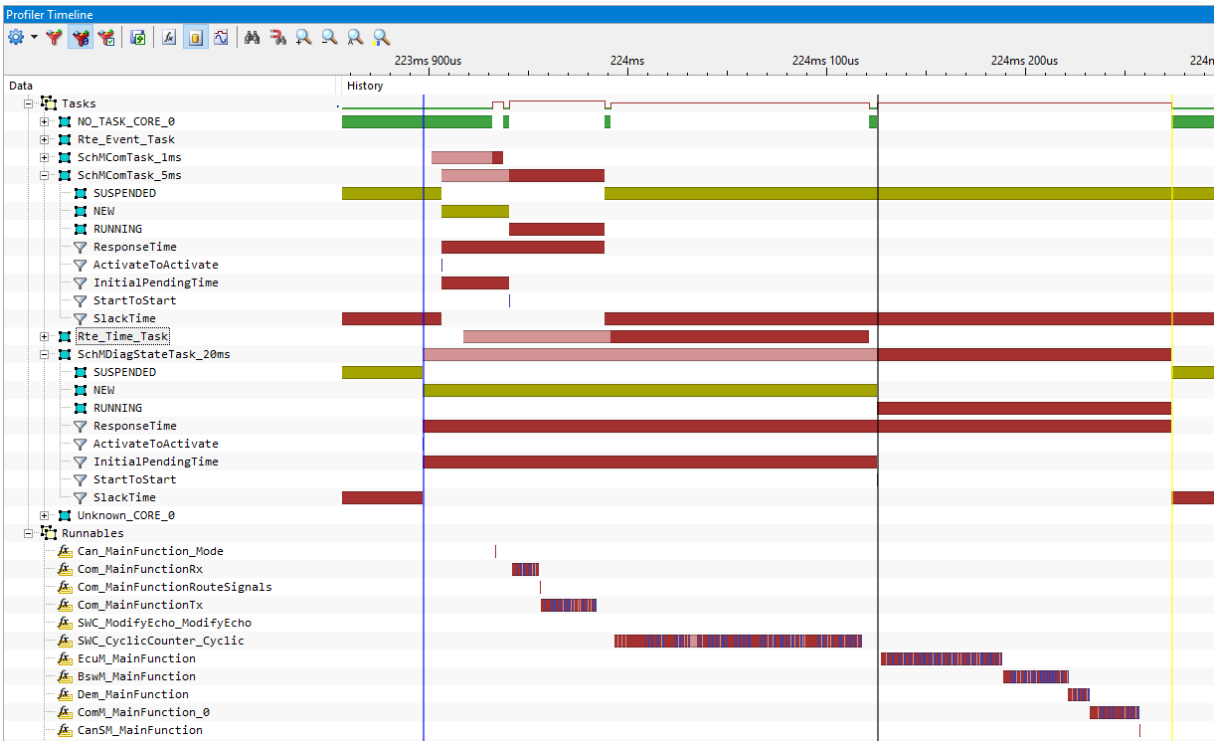
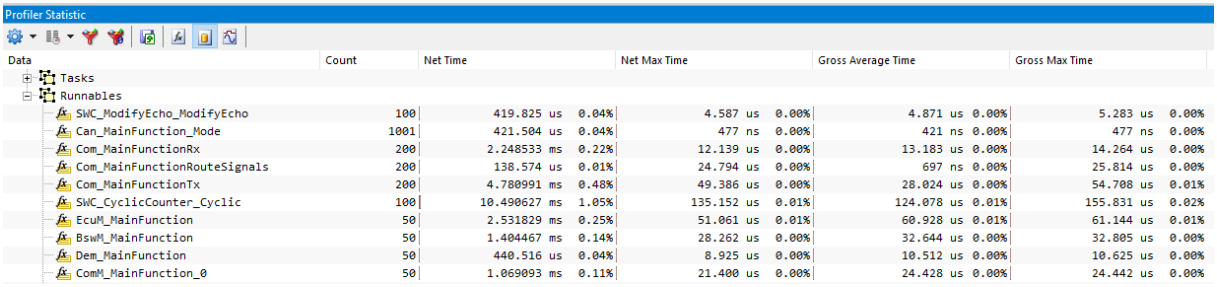


Figure 7: OS Task State and Runnable Trace in the winIDEA Profiler Timeline.

2.3.2 Profiler Statistics

The winIDEA Profiler also calculates statistics for the Runnables. A sample Profiler Statistics windows is shown in Figure 8.



Data	Count	Net Time	Net Max Time	Gross Average Time	Gross Max Time
Tasks					
Runnables					
SWC_ModifyEcho_ModifyEcho	100	419.825 us 0.04%	4.587 us 0.00%	4.871 us 0.00%	5.283 us 0.00%
Can_MainFunction_Mode	1001	421.504 us 0.04%	477 ns 0.00%	421 ns 0.00%	477 ns 0.00%
Com_MainFunctionRx	200	2.248533 ms 0.22%	12.139 us 0.00%	13.183 us 0.00%	14.264 us 0.00%
Com_MainFunctionRouteSignals	200	138.574 us 0.01%	24.794 us 0.00%	697 ns 0.00%	25.814 us 0.00%
Com_MainFunctionTx	200	4.780991 ms 0.48%	49.386 us 0.00%	28.024 us 0.00%	54.708 us 0.01%
SWC_CyclicCounter_Cyclic	100	10.490627 ms 1.05%	135.152 us 0.01%	124.078 us 0.01%	155.831 us 0.02%
EcuM_MainFunction	50	2.531829 ms 0.25%	51.061 us 0.01%	60.928 us 0.01%	61.144 us 0.01%
BswM_MainFunction	50	1.404467 ms 0.14%	28.262 us 0.00%	32.644 us 0.00%	32.805 us 0.00%
Dem_MainFunction	50	440.516 us 0.04%	8.925 us 0.00%	10.512 us 0.00%	10.625 us 0.00%
ComM_MainFunction_0	50	1.069093 ms 0.11%	21.400 us 0.00%	24.428 us 0.00%	24.442 us 0.00%

Figure 8: Sample Profiler Statistics for Runnables

In the statistics in Figure 8 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 9 lists all the statistics which can be calculated by the profiler for each profile object.

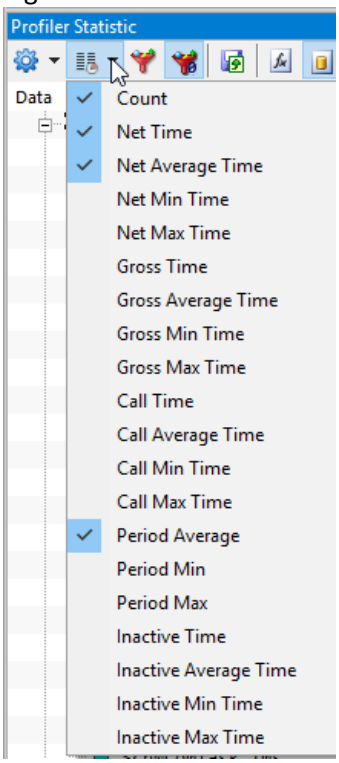


Figure 9: Complete list of calculated Profiler Statistics

Another possibility to get a summary of all statistics for a profile object such as a task or a Runnable is to open the “Properties…” view. It can be opened by selecting the desired profiler object and then hit “Alt + Enter” (or right mouse click), as shown in .

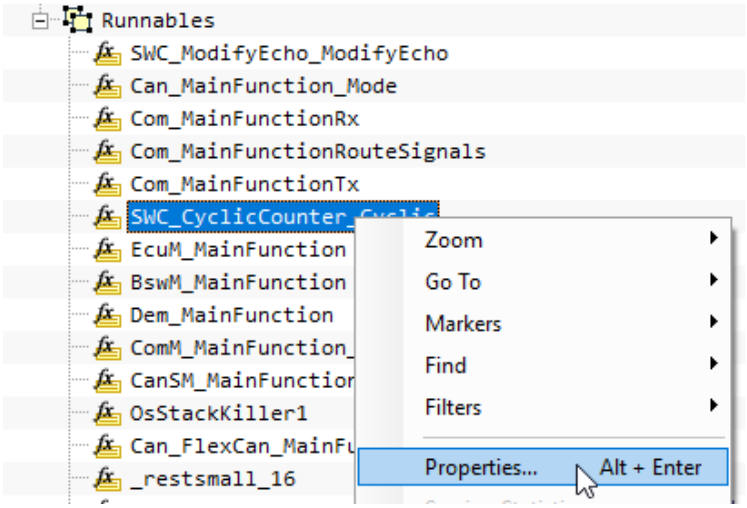


Figure 10: Opening the Properties View for a selected Profiler Object, for instance the Runnable SWC_CyclicCounter_Cyclic

Figure 11: Properties View for the Runnable “SWC_CyclicCounter_CyclicFigure 11 shows a sample Properties view for the Runnable “SWC_CyclicCounter_Cyclic”.
For each metric the total, average, maximum and minimum time is measured, along with the time (and link “->” of its occurrence.

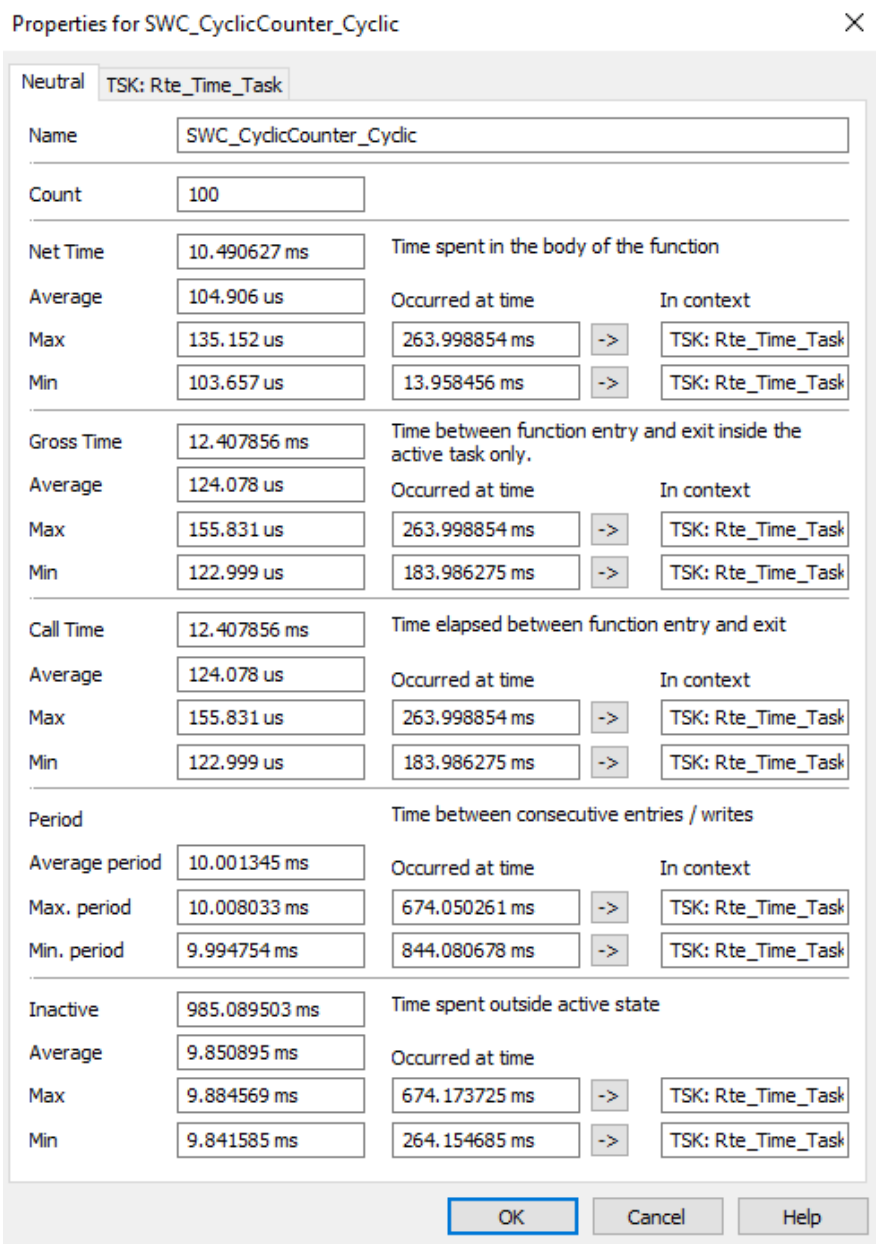


Figure 11: Properties View for the Runnable “SWC_CyclicCounter_Cyclic”

3 Runnable Profiling via RTE VFB Tracing Hook Instrumentation

Tracing of Runnables can be accomplished via program flow trace or by means of instrumentation. This section describes how to record a Runnable aware trace by instrumenting the Virtual Function Bus (VFB) trace hooks. VFB tracing allows the interaction between the AUTOSAR software components to be traced. The user can decide which events like Runnable Start and Return or Sender/Receiver Send should be traced.
This section explains how to activate Runnable hooks in the EB tresos Studio and generate empty hooks by using template code generation. The hooks are then filled with the necessary instrumentation code. All Runnable events are written into a single global variable which can be recorded via data tracing. Finally, the winIDEA Profiler is configured to interpret the global variable as a Runnable trace via an iSYSTEM Profiler XML file.

3.1 RTE Configuration

3.1.1 RTE VFB Trace Configuration

At first, the VFB Runnable hooks for all Runnables that should be recorded must be activated in the EB tresos Studio. This needs to be done in the RTE Editor.

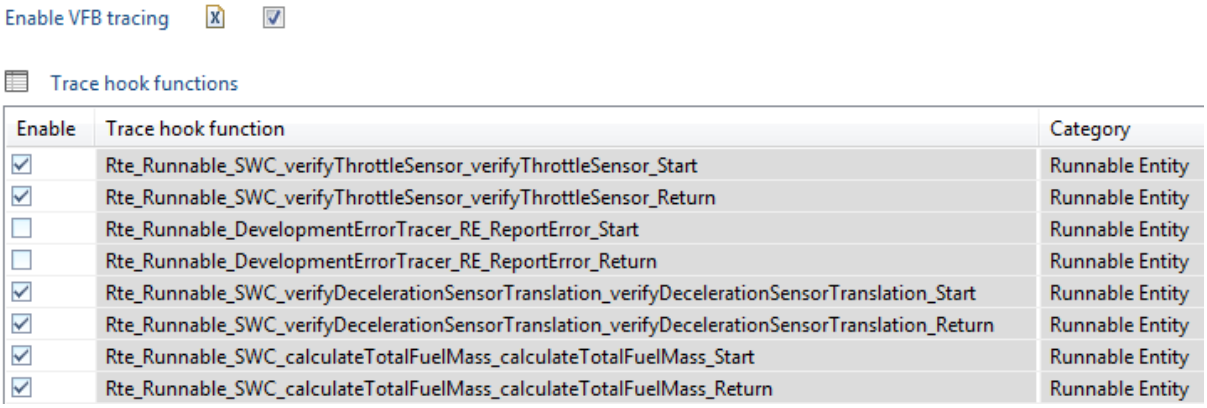


Figure 12: VFB Trace Hook Function Selection in the EB tresos RTE Editor

Listing 1 shows a sample OS task generated by the RTE Generator. The task T_Event_Send2Com only a single Runnable “R_Send2Com”. The Runnable call is surrounded the corresponding start and return hook function calls.

```
TASK(T_Event_Send2Com)
{
  /* scenario A1/B1 (B1) */
  Rte_Task_Dispatch(T_Event_Send2Com);
  {
    /* RunnableEntity R_Send2Com */
    Rte_Runnable_SWC_Send2Com_R_Send2Com_Start();
    R_Send2Com();
    Rte_Runnable_SWC_Send2Com_R_Send2Com_Return();
  }
  Rte_Task_EndHook(T_Event_Send2Com);
  (void)TerminateTask();

} /* TASK(T_Event_Send2Com) */
```

Listing 1: Sample OS Task generated by the RTE Generator including start and return hook for the Runnable R_Send2Com();

The RTE Generator generates a separate hooks header file for each Software Component (SWC). This header file defines the default hook function as “void”. Only if there is already a definition of a “real” hook function the header file declares the hook function as “extern” and assumes the actual implementation in a C source file provided by the user.

```
/*  ========================================================== */
/* File: ouput\generated\include\Rte_SWC_Send2Com_Hook.h  */
/*  ========================================================== */
#if defined(Rte_Runnable_SWC_Send2Com_R_Send2Com_Start) && (RTE_VFB_TRACE == FALSE)
#undef Rte_Runnable_SWC_Send2Com_R_Send2Com_Start
#endif
#if defined(Rte_Runnable_SWC_Send2Com_R_Send2Com_Start)
#undef Rte_Runnable_SWC_Send2Com_R_Send2Com_Start
extern FUNC(void, RTE_APPL_CODE) Rte_Runnable_SWC_Send2Com_R_Send2Com_Start(void);
#else
#define Rte_Runnable_SWC_Send2Com_R_Send2Com_Start() ((void)0)
#endif /* Rte_Runnable_SWC_Send2Com_R_Send2Com_Start */
#if defined(Rte_Runnable_SWC_Send2Com_R_Send2Com_Return) && (RTE_VFB_TRACE ==
FALSE)
#undef Rte_Runnable_SWC_Send2Com_R_Send2Com_Return
#endif
#if defined(Rte_Runnable_SWC_Send2Com_R_Send2Com_Return)
#undef Rte_Runnable_SWC_Send2Com_R_Send2Com_Return
extern FUNC(void, RTE_APPL_CODE) Rte_Runnable_SWC_Send2Com_R_Send2Com_Return(void);
#else
#define Rte_Runnable_SWC_Send2Com_R_Send2Com_Return() ((void)0)
#endif /* Rte_Runnable_SWC_Send2Com_R_Send2Com_Return */
```

Listing 2: Example VFB trace hooks template generated by EB tresos Studio for the SWC_Send2Com_R_Send2Com Runnable.

3.1.2 RTE VFB Trace Hook Functions

To implement the hooks, it is necessary to define a unique positive integer for each Runnable. The number 0 is reserved to indicate a Runnable exit and is used in every return-hook. Once you have generated a mapping the instrumentation can be added to the source code.
Note that for a multi-core application, the Runnable-ID itself is not sufficient, but the core number must be written into the global variable as well. The following code works for TriCore microcontrollers where the core ID can be accessed via the *Move From Core Register* (`mfcr`) command. For single core applications, the part after the *exclusive-or* can be removed. For other architectures the code to get the core ID must be adapted.

```
#define CPU_CORE_ID 0xFE1C

isystem_trace_runnable = <runnable_id> | (__mfcr(CPU_CORE_ID) << 24);
```

Using this code snippet, the implementation of the hooks would look as shown in Listing 3. The global variable `isystem_trace_runnable` must be defined at the beginning of the hooks file as a 32-bit integer.

```
/* ========================================
   File: isystemRunnableTrace.c
   iSYSTEM Runnble Trace Hook Functions
   CPU: TriCore
   Trace Concept: Data Trace
   ======================================== */
unsigned long isystem_runnable_trace;

/* Runnable Hooks */

/* Start hook of Runnable R_Send2Com (ID = 0x01). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Start(void) {
      isystem_runnable_trace = 0x01 | (__mfcr(CPU_CORE_ID) << 24); }


/* Return hook of Runnable R_Send2Com (common return ID = 0x00). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Return(void) {
      isystem_runnable_trace = 0 | (__mfcr(CPU_CORE_ID) << 24); }
```

Listing 3: Sample Implementation of Runnable hooks for an Infineon TriCore microcontroller.

For multiple Runnables the implementation is identical except the Runnable Start ID being unique for each Start hook.

3.1.3 iSYSTEM Profiler XML File

Typically, run-time analysis on AUTOSAR based applications is done by means of the ORTI file. However, the ORTI file only covers objects of the AUTOSAR OS, such as tasks and ISR2s.
AUTOSAR Runnable are implemented within the RTE, i.e. are outside the scope of the ORTI file.
iSYSTEM uses a proprietary XML file format to describe the tracing/profiling related aspects of OSes, hypervisors or AUTOSAR RTE objects which are subject to timing/scheduling to the winIDEA Profiler (iSYSTEM Profiler XML file).
Such a Profiler XML file can be used to basically extend the ORTI file, with RTE related information such as Runnables.
The figure below shows a sample profiler XML file used for Runnable profiling based on data trace of the global variable `isystem_runnable_trace` of the VFB trace hook instrumentation.

```
  <TypeEnum>
    <Name>Types_RUNNABLE_MAPPING</Name>
    <Enum><Name>R_calculate_B</Name>                    <Value>1</Value></Enum>
    <Enum><Name>R_Send2Com</Name>                       <Value>2</Value></Enum>
    <Enum><Name>calculateThrottleSensor</Name>          <Value>3</Value></Enum>
    <Enum><Name>verifyBrakePedalSensorVoter</Name>      <Value>4</Value></Enum>
    <Enum><Name>processThrottleSensor</Name>            <Value>5</Value></Enum>
  </TypeEnum>

</Types>

<Profiler>
    <Object>
      <Definition>RUN</Definition>
      <Description>Runnables</Description>
      <Type>Type_RUNNABLE_MAPPING</Type>
      <Level>Runnable</Level>
      <Expression>isystem_runnable_trace</Expression>
      <Signaling></Signaling>
      <DefaultValue>0</DefaultValue>
      <Runnable>
        <MaskID>0xffffff</MaskID>
        <MaskCore>0xFF000000</MaskCore>
        <ExitValue>0</ExitValue>
      </Runnable>
    </Object>
```

Figure 13: Sample iSYSTEM Profiler XML File for Runnable Profiling based on Data Trace of the VFB Trace Hook Variable isystem_runnable_trace

In the upper section (“Types”) an enumeration type is defined (“Type_Runnable_MAPPING”), which maps a Runnable name, displayed in the winIDEA Profiler to its corresponding ID used by the VFB instrumentation hook functions.

In the lower section (“Profiler”), a new profiler object is created. It is defined as a “RUN” object using data tracing of the global variable (i.e. Expression) `isystem_runnable_trace` to signal Runnable ID and core ID. The “Type” tag tell the profiler to use the enum type “Type_Runnable_MAPPING” for the Runnable naming.
Alternatively to data trace, also processor-specific variants of instrumentation trace may be used for Runnable tracing. Such instrumentation trace techniques are for instance, RH850 Software Trace or ARM CoreSight ITM or STM (see also).

In a typical use-case, these XML file section will actually be part of a more comprehensive Profiler XML file also used for OS task (and ISR) state tracing.

The trace hardware of the processor needs to be configured for unconditional program trace. Please refer to Section 0 “

Hardware Trace Configurations" for the processor-specific settings.

The Profiler only needs to be configured for OS objects profiling. Code profiling is not needed.

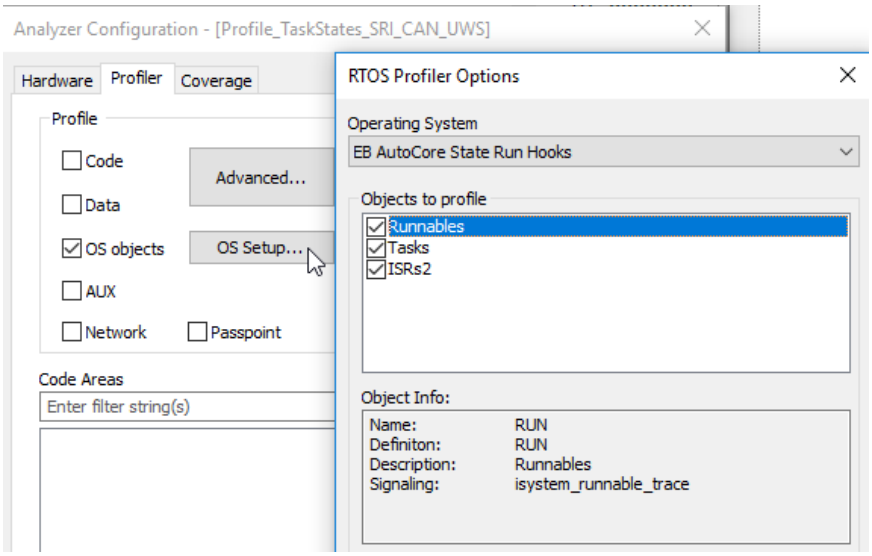
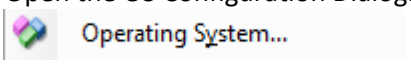


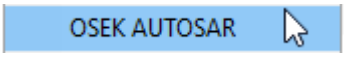
Figure 14: Profiler Configuration for Runnable Profiling based on Program Trace

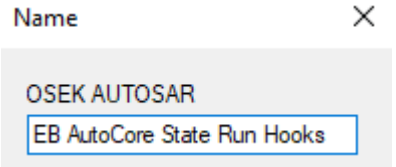
But first, the Profiler XML file has to be added to the winIDEA workspace and then the profiler is configured to use this information. To add the XML file to the workspace execute the following steps.

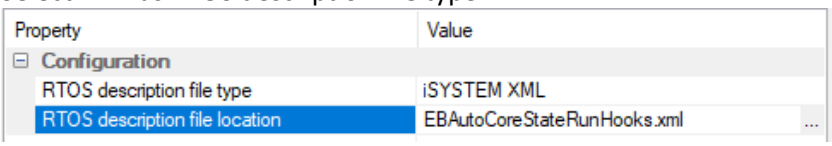
9. Open the Debug menu.
   Debug

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.
    

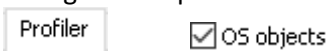
15. Select the XML file and click OK.

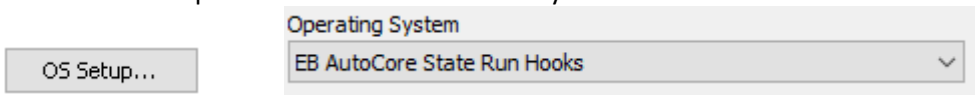
16. Make sure to load symbols to make the change active.


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

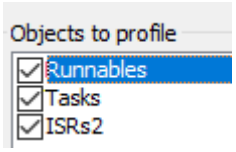
9. Open the profiler configuration. Make sure it is the same configuration for which data tracing of the XML variables is configured.


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


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


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


13. 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.)

14. 

15. Confirm with OK.


16. Start a new trace recording.


3.2 Hardware Trace Configurations

For most processors the on-chip trace logic needs to be configured for data trace of the Runnable trace variable used by the instrumentation hooks, e.g. the global variable `isystem_runnable_trace`.
However, some processors do not provide a suitable data trace infrastructure. In such cases alternative trace concepts need to be applied.
Examples for such processors are some Renesas RH850 derivatives or some higher-end Cortex A or R based devices.

3.2.1 ARM CoreSight System Trace Macrocell (STM)

Most processors based on Cortex A and/or R core do not offer efficient data tracing capabilities. For instance on a Cortex R7 core, data trace is only available in conjunction with instruction trace and thus generates massive amount of trace data not really needed for pure Runnable profiling. Moreover, most Cortex A cores do not offer data trace at all. In such cases, the ARM System Trace Macrocell (STM) allows for efficient instrumentation tracing.

The concept behind STM trace is that a core can perform data write transactions to a memory mapped area of the STM, residing on the AXI bus of the processor. This memory mapped area, called the Stimulus Port, is divided into multiple so-called Channels (256 bytes per channel). A write transaction to such a STM Stimulus Port Channel causes the STM to emit a STM message via the hardware trace port. The Channel number is encoded in the STM message can be used to differentiate different messages types, such as trace messages for task state signaling or for Runnable entry/exit signaling.

In the sample VFB trace hook implementation shown in Listing 4, the STM Channel 1 is used by the R7 core of a RCAR M3 processor to signal Runnable start and return hooks executed by the R7 core.

```
/* ========================================
   File: isystemRunnableTrace.c
   iSYSTEM Runnble Trace Hook Functions
   CPU: RCAR M3, Cortex R7
   Trace Concept: STM, Channel 1
   ======================================== */
#define STM32_DTS(ch) *(volatile unsigned int*)(0xE9000010 + (ch*0x100))
#define STM_TRACE_R7_RUN(value)  do { STM32_DTS(0x1) = value; } while(0)

/* Runnable Hooks */

/* Start hook of Runnable R_Send2Com (ID = 0x01). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Start(void) {
      STM_TRACE_R7_RUN(0x01); }


/* Return hook of Runnable R_Send2Com (common return ID = 0x00). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Return(void) {
      STM_TRACE_R7_RUN(0x00);  }
```

Listing 4: Sample Implementation of Runnable hooks using ARM CoreSight STM Trace with on a Renesas RCAR M3 Cortex R7.

The iSYSTEM Profiler XML needs to be adjusted to use STM trace for Runnable signaling. A sample iSYSTEM Profiler XML is given Listing 5.

```
<Profiler>
    <Object>
      <Definition>RUN</Definition>
      <Description>Runnables</Description>
      <Type>Type_RUNNABLE_MAPPING</Type>
      <Level>Runnable</Level>
      <Expression></Expression>
      <Signaling>STM(0x001)(</Signaling>
      <DefaultValue>0</DefaultValue>
      <Runnable>
        <MaskID>0xffffff</MaskID>
        <MaskCore>0xFF000000</MaskCore>
        <ExitValue>0</ExitValue>
      </Runnable>
    </Object>
```

Listing 5: iSYSTEM Profiler XML file for Runnable Tracing using ARM CoreSight STM Trace

3.2.2 RH850 Software Trace

Although all RH850 micro controllers implement a debug and trace logic that is compliant with the Nexus Class 3+ standard (which also includes data access trace), some derivates do not offer a trace port to allow for trace data streaming to a hardware trace tool. Instead, only a relatively small on-chip trace RAM is available, not always sufficient for AUTOSAR OS and Runnable timing analysis. The alternative solution in these cases is the so-called 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

As the `DBTAG` instruction takes an immediate value as input argument it can be used to create messages for values which are known at compile time, such as indices of function (Runnable) entries/exits. The `DBPUSH` instruction uses a core register as input argument and thus is suited for signaling events which are only known at run-time, such OS task or ISR2 state changes.

In the sample VFB trace hook implementation in Listing 6 shows how the `DBTAG` instruction can be used to signal a Runnable ID in a Start and Return hook function.

```
/* ========================================
   File: isystemRunnableTrace.c
   iSYSTEM Runnble Trace Hook Functions
   CPU: RH850
   Trace Concecpt: Software Trace DBTAG
   Compiler: GHS
   ======================================== */
/* DBTAG macro, Value passed as max 10-bit immediate value. */
asm void isystem_sft_dbtag(value)
{
%con value
  dbtag value
}

/* Runnable Hooks */
/* Start hook of Runnable R_Send2Com (ID = 0x01). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Start(void) {
      isystem_sft_dbtag(0x001); }


/* Return hook of Runnable R_Send2Com (common return ID = 0x00). */
void Rte_Runnable_SWC_Send2Com_R_Send2Com_Return(void) {
      isystem_sft_dbtag(0x000);  }
```

Listing 6: Sample Implementation of Runnable hooks using RH850 Software Trace.

The iSYSTEM Profiler XML needs to be adjusted to use RH850 Software Trace for Runnable signaling. A sample iSYSTEM Profiler XML is given in Listing 7.

```
<Profiler>
    <Object>
      <Definition>RUN</Definition>
      <Description>Runnables</Description>
      <Type>Type_RUNNABLE_MAPPING</Type>
      <Level>Runnable</Level>
      <Expression></Expression>
      <Signaling>DBTAG</Signaling>
      <DefaultValue>0</DefaultValue>
      <Runnable>
        <MaskID>0x00FFFFFF</MaskID>
        <MaskCore>0xFF000000</MaskCore>
        <ExitValue>0</ExitValue>
      </Runnable>
    </Object>
```

Listing 7: iSYSTEM Profiler XML file for Runnable Tracing using RH850 Software Instruction DBTAG

3.3 winIDEA Profiler Visualization

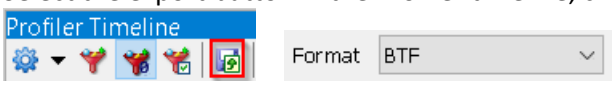
The Runnables trace by means of VFB trace hook instrumentation are visualized in the iSYSTEM Profiler in the same way as described in Section 2.3.

4 BTF Export

The winIDEA Profiler supports the export of traces into the BTF format. BTF is a CSV based trace format that is supported by different timing tool vendors. Before the BTF export is usable the iSYSTEM profiler XML file must be configured accordingly. The Profiler supports the export of tasks, ISRs, Runnables and signals. More details on the iSYSTEM Profiler XML configuration can be found in the Application Note "AN_iSYSTEM_EB_AutoCore_Profiling" or "AN_iSYSTEM_EB_SafetyOS2_Profling", respectively.

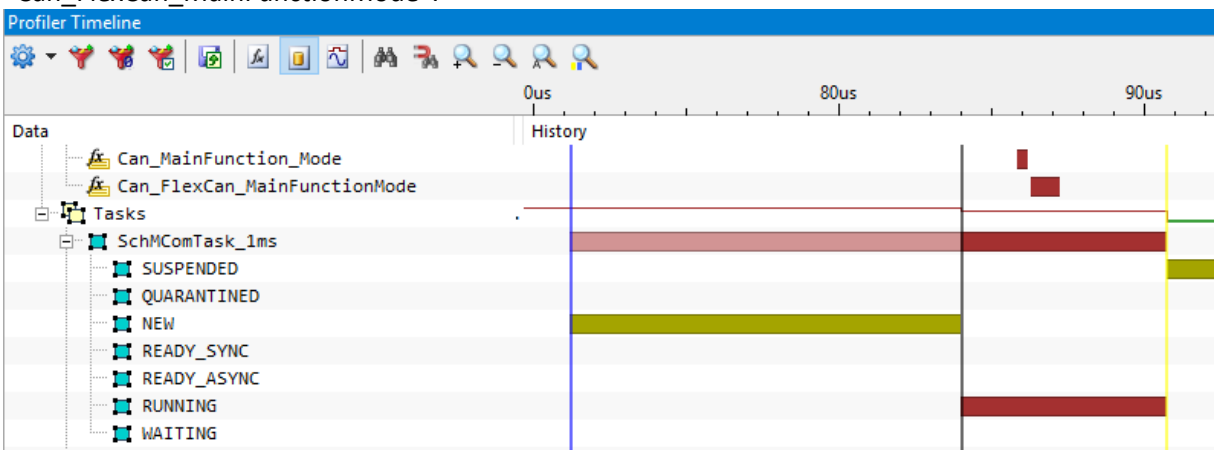
Once the iSYSTEM Profiler XML is updated the following steps must be executed to export a BTF trace file.

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 **Error! Reference source not found.**.

Figure 15 shows a small section of a sample BTF Export, covering the response time of the task "SchMComTask_1ms" including the Runnables mapped into this task, "Can_MainFunction_Mode" and "Can_FlexCan_MainFunctionMode".




```
71232,STI_SchMComTask_1ms,0,T,SchMComTask_1ms,0,activate
84032,CORE_0,0,T,SchMComTask_1ms,0,start
85857,SchMComTask_1ms,0,R,Can_MainFunction_Mode,0,start
86249,SchMComTask_1ms,0,R,Can_MainFunction_Mode,0,terminate
86347,SchMComTask_1ms,0,R,Can_FlexCan_MainFunctionMode,,PA_MPC574XG_simple_demo_can_rte.elf,0,start
87251,SchMComTask_1ms,0,R,Can_FlexCan_MainFunctionMode,,PA_MPC574XG_simple_demo_can_rte.elf,0,terminate
90760,CORE_0,0,T,SchMComTask_1ms,0,terminate
```

Figure 15: Sample BTF Export of OS tasks and Runnables

.

5 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 AUTOSAR Runnables.

5.1 Runnable Call-Time Timing Constraint

An Inspector can for instance be used to check whether a Runnable exceeds its timing budget in terms of elapsed time between Runnable Entry and Exit, i.e. its execution time including all sub-function calls and task/ISR preemptions (may also be called Response Time).

Once such an Inspector has been configured, additional Profiler objects will appear in the Profiler Timeline (and also in the Statistics). Inspector Objects can be identified by the  symbol.
An Inspector Object can be seen as a Finite State Machine (FSM), which changes its state upon user-defined events.

Figure 16 shows an Inspector called “SWC_Cyclic_CallTime_Constraint” which has the states “WithinConstraint” and “Violation”. The FSM enters the “WithinConstraint” state upon entry into the Runnable. It exits this state back to its default state upon the exit of the Runnable. However, in case the Call Time exceeds the specified timing budget (in this example 123500 ns), the state changes to “Violation”.

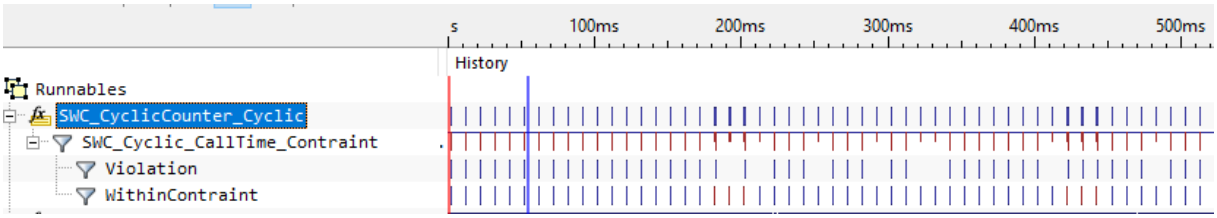


Figure 16: iSYSTEM Profiler Timeline of the Runnable “SWC_CyclicCounter_Counter” including a Profiler Inspector which checks for Violations of a specified Call Time Timing Constraint.

The trace recording in Figure 16 spans of a time period of 500ms and it can easily be recognized that timing violation have occurred.
In addition, the “Properties” view of the “Violation” object (Inspector state “Violation”) can be open as shown in Figure 17.

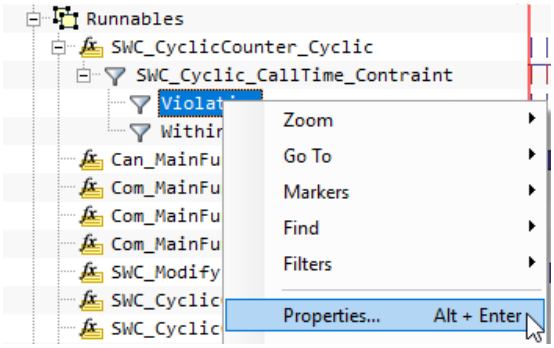


Figure 17: Opening the “Properties” View for the Profiler Object “Violation”

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

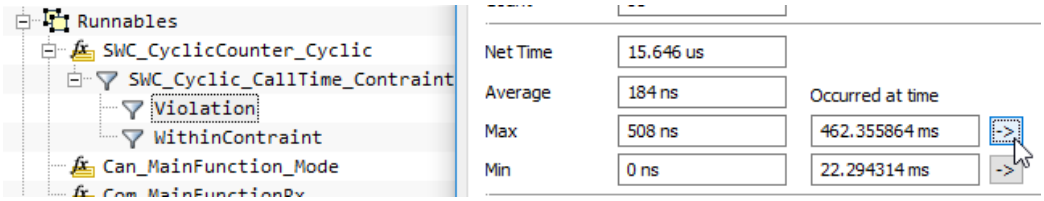


Figure 18: Net Time Properties for the "Violation" Inspector Object. The maximum Timing Violation of 508ns occurred at trace time stamp 462.35ms.

Figure 19 shows the instance of the Runnable "SWC_CycleCounter_Cyclic" which results in the maximum timing violation.

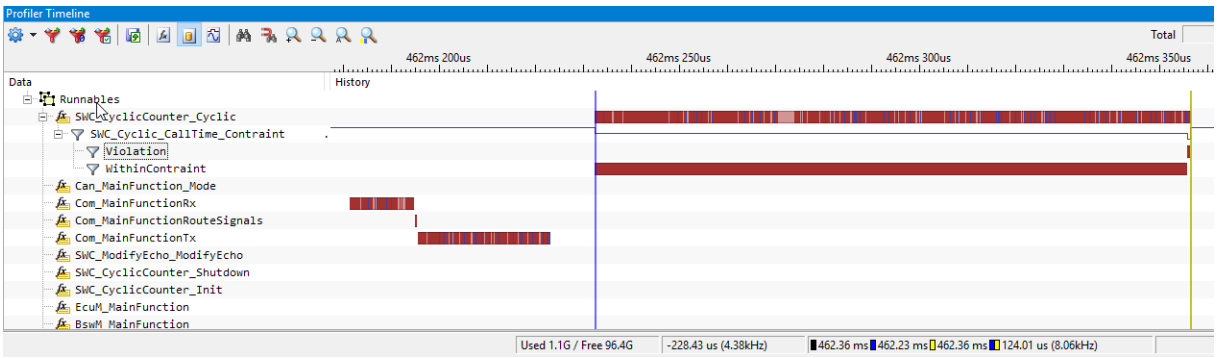


Figure 19: Zoomed View of the maximum Timing Violation occurrence

Inspectors can be created or modified either within the winIDEA GUI or as a textual representation in form of a JSON file.
The Inspector Dialog of the winIDEA GUI can be open within the iSYSTEM Profiler Timeline window, by clicking the "Inspectors" icon as shown in Figure 20.

Figure 20: "Inspectors" Icon within the Profiler Timeline Window

Figure 21 shows the configuration dialog of the sample Inspector "SWC_Cycle_CallTime_Contraint".

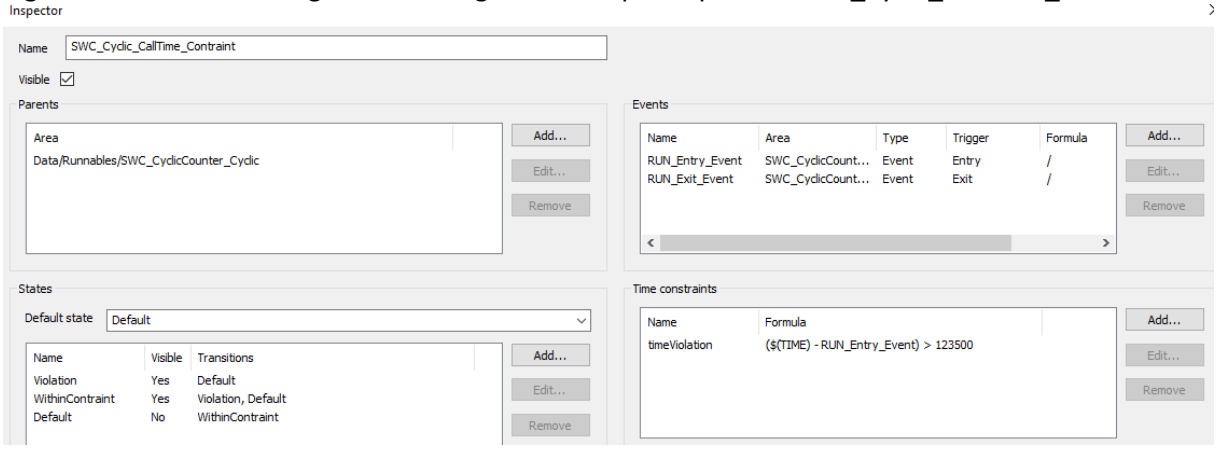


Figure 21: Configuration Dialog for the Inspector "SWC_Cycle_CallTime_Contraint"

6 Technical Support

6.1 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

6.2 Contact

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