Introduction to AUTOSAR Classic Profiling

Updated: 07/24/2020

Table of Contents

1 Overview

In this document, we give an overview of the AUTOSAR Classic Profiling use-case. Profiling is the process of recording, analyzing, and evaluating the timing behavior of an embedded system. If we talk about AUTOSAR Classic profiling, we mean profiling the runtime behavior of an AUTOSAR Classic based application, specifically the application itself, the Run-Time Environment (RTE), the Operating System (OS), and other parts of the Basic Software (BSW). With the resulting data, users can cover use-cases such as CPU load analysis, event-chain analysis, timing requirements validation, timing constraint validation, OS metric calculation, and more. If you are entirely new to this topic, we recommend watching the Introduction to AUTOSAR Classic Profiling webinar.

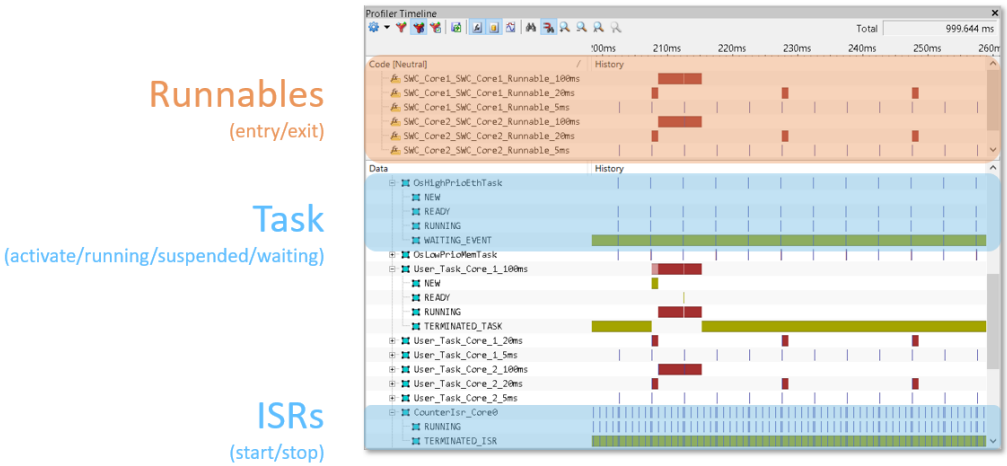

Figure 1: The result of AUTOSAR Profiling in iSYSTEM winIDEA. The Gantt-Chart shows the state of different OS (Tasks/ISRs) and RTE (Runnables) objects over time. The dark red color indicates that an object is running.

The general process for AUTOSAR profiling always follows the same pattern, as shown in Figure 2. First, the user assesses the available trace capabilities, the OS and RTE in use, and the trace objects to analyze. Second, the user configures the OS, winIDEA, and the hardware trace logic. Finally, winIDEA traces the application, profiles the recorded data, and makes it available for analysis by the user. The result is a detailed view of the OS and RTE, as shown in *Figure 1*. The user then analyzes the data in winIDEA itself, generates reports, or exports the data for analysis in other tools. To summarize, AUTOSAR Profiling requires the three steps assessment, configuration, and profiling. The section *Workflow* discusses these steps in more detail.

Before we discuss the workflow, it is essential to understand the variables that influence the steps. First, the user defines the objects of interest in the analysis. For example, CPU load analysis requires profiling only Tasks and ISRs, while event-chain analysis may require Runnables and RTE-Port communication data. Second, depending on the microcontroller, certain trace features may or may not be available. For example, data-trace is not available on all microcontrollers that have trace capabilities, in which case instrumentation trace may be required. Finally, not all AUTOSAR Operating Systems and RTEs are equal. Some differences between vendors need consideration to end up with a working setting. To summarize, the user must consider the three factors desired objects, available trace capabilities, and operating system and RTE vendor, for the configuration. The section *Background* discusses each element in more detail.

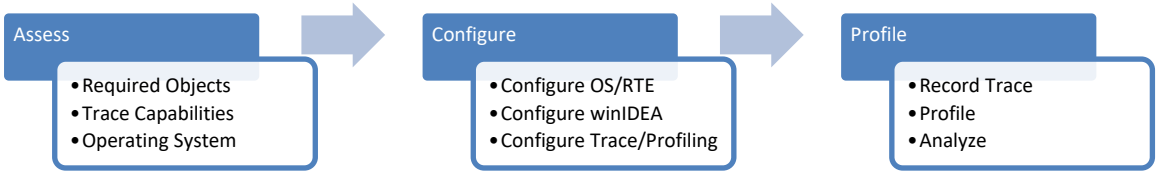

Figure 2: Process for AUTOSAR Classic Profiling in winIDEA.

2 Background

To configure AUTOSAR profiling correctly, working knowledge of trace objects, trace techniques, and operating systems is necessary. The following sections discuss these topics.

2.1 Trace Objects

An AUTOSAR application consists of different objects such as Tasks, ISRs, Runnables, Functions, RTE Ports, and more. The user must decide which types of objects are relevant for their use-case. Profiling the Running Task and ISR is a good start if the use-case is not clear.

2.1.1 Running Task/ISR

The most common objects for timing analysis are Tasks and ISRs. Tasks are operating system containers for Runnables. ISRs are interrupt triggered service routines for high priority computations. A trace containing only the information about the currently running Task and ISR is called a Running Task/ISR Trace.
Some operating systems use Thread as an umbrella term for Tasks and ISRs.

2.1.2 Task State/Running ISR

A Running Task/ISR Trace is not always sufficient for timing analysis. For example, while the information is adequate to calculate the CPU utilization of a system, it is not enough to calculate the response time of a given task.
Response time is the time from activation to termination of a Task, including the times during which the task is in the ready state (meaning a higher priority task is running). With a running Task/ISR, trace information about activations and preemptions is not available. In this case, a Task State/Running ISR trace is necessary because it includes to task state model of each task.
With the knowledge about the states, the Profiler knows when a task is terminated or just preempted. Task states make calculating a couple of advanced metrics, including the response time, feasible.

2.1.3 Runnables

AUTOSAR Runnables are special functions defined in the RTE. They are mapped to tasks and executed in the context of those tasks. RTE events, such as timing events and data-received events, trigger the execution of Runnables. Runnable tracing becomes essential when a more detailed view of the application is required.
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 Runnables. By tracing the Task State and Running ISR information, the Profiler can reconstruct the Runnable preempt and resume events.

2.1.4 Other Objects

AUTOSAR Profiling currently focuses on the three discussed objects, Tasks, ISRs, and Runnables. In the future, additional classes are going to become relevant, and we will extend this section accordingly.
AUTOSAR Runnables map to software components. Communication between different software components happens via communication interfaces, such as sender/receiver and client/server interfaces. Profiling communication via these interfaces is one possible future use-case.
Additional objects are Spinlocks, Resources, OS Critical Sections, and other synchronization mechanisms. Finally, a user may want to record custom events such as write and read accesses to a specific signal. Tracing signals allows for verification of so-called data age constraints. A data age constraint is a time limit on the period between data production and consumption.

2.2 Trace Techniques

There are three main trace measurement techniques: software, hybrid, and hardware-based tracing. A trace is a list of events with timestamps in ascending order enabling the user to examine the runtime behavior of a system.
In this section, we focus on hardware-based tracing. Hardware-based tracing relies on dedicated on-chip trace logic, which captures events of interest and sends them off-chip. *Figure 3* shows the steps involved in recording trace data from the target. Depending on the microcontroller, zero or more of the following techniques are available. The available trace techniques influence which Trace Objects are recordable on for how long.

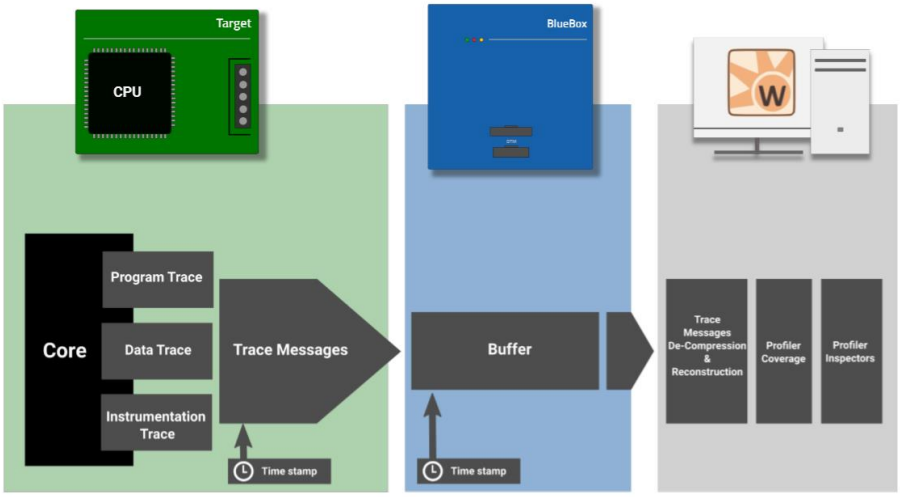

Figure 3: Recording a hardware trace with winIDEA. Dedicated on-chip trace logic generates trace messages that are captured by the debugger and send to the host computer for profiling.

2.2.1 Program Flow Trace

A program flow trace (also called instruction trace) records the instructions the CPU executes. This technique 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 Runnables and functions.
Note that the on-chip trace logic does not create a dedicated trace message for every instruction. Instead, it only creates messages for instructions that change the flow of the program, such as calls and conditional jumps. winIDEA then reconstructs the full program flow trace from the compressed data.
Even with compressed function trace, program flow trace still generates high amounts of data. Therefore, depending on the trace interface (the interface between the microcontroller and the debugger), recording program flow traces may only be possible for short periods (less than 100ms).

2.2.2 Data Trace

Data tracing records read and write accesses to memory, and thus allows recording of write accesses to global variables. Comparators enable the user to configure the memory areas and variables of interest. An example use-case for data tracing is monitoring the OS running task variable to generate a task trace.

2.2.3 Instrumented Data Trace

For some use-cases, pure data tracing is not sufficient to record all necessary information. For example, the RTE does not provide a variable indicating the currently running Runnable. In such cases, adding instrumentation to write the desired information into a dedicated variable is a compromise between pure hardware-based tracing and software trace. This compromise is also commonly used for recording task state traces for specific operating systems.

2.2.4 Instrumentation Trace

Instrumentation trace is similar to Instrumented Data Trace. Instrumentation trace utilizes microcontroller specific memory areas or instructions. When the application writes into one of these memory areas, or when it executes one of the instructions, the on-chip trace logic automatically generates a trace message. Examples for Instrumentation Trace are Software Trace (SFT) for RH850 based microcontrollers and Ownership Trace Messages (OTM) for PowerPC based microcontrollers.
For example, the RH850 software trace provides three trace instructions: DBCP messages include the current PC value, DBTAG messages include a constant value (specified at compile-time), and DBPUSH takes a value from a specific general-purpose register. The software trace module sends the trace messages off-chip via the serial port LPD-4 interface. The LPD-4 interface contains an eight-line FIFO where each line has 48 bits capacity. While the FIFO helps to overcome short term bottlenecks generating too many software trace messages may still result in overflows.
On microcontrollers where data-trace is not available, instrumentation trace is often a good alternative.

2.3 Operating System

Even though all AUTOSAR operating systems adhere to the same specification, there are still differences that impact AUTOSAR OS and RTE profiling.
For example, while all AUTOSAR based operating systems have a so-called task-state attribute for each task, the method for each OS differs significantly. For some operating systems, the formula points to a simple global variable; for others, it is a complex expression containing multiple variables, as shown in *Listing 1.*
The first task signals the state via a simple global variable. The operating system keeps track of the task states via a global array and each field maps to a specific task. By recording write accesses to this array, the Profiler can reconstruct the states for all task objects.
The second task uses a complex formula consisting of multiple global variables to signal the state. To reconstruct the state, we must record the write accesses to each of these variables. We then use a unique winIDEA feature, the so-called Profiler Inspectors, to reconstruct the state from the individual variables. We provide a tool that generates the Profiler Inspectors for all tasks automatically.

```
TASK Init_Task
{
  STATE = "OS_taskState[1]";
  /* lines removed for readability */
};

TASK QM_Task {
  STATE = "(((Os_RunningTask == Os_const_tasks[0]) * 1) & 1) +
((Os_ReadyTasks.p0 & 0x1) << 1)";
  /* lines removed for readability */
};
```

Listing 1: ORTI task state attributes for two tasks. We can record the state of the first task via a single global variable. However, the second task encodes the state information into multiple global variables.

The most common AUTOSAR classic operating systems are Elektrobit AutoCore OS, Elektrobit Safety OS, ETAS RTA-OS, and Vector MICROSAR OS. We provide specific resources for each of these operating systems.

3 Workflow

With a basic understanding of the technologies involved in AUTOSAR classic profiling, we can now discuss the workflow in more detail. This section also links to additional resources such as specific application notes and webinars.

3.1 Assess

The first step is to decide which types of objects to record based on the trace techniques you have available on your microcontroller. Currently supported classes are running tasks and running ISR, task states and running ISR, and Runnables. Depending on the microcontroller, there may or may not be a way to record the desired information. The row headers in *Table 1* list the different hardware trace techniques, and the column headers contain the different types of objects. Note that there is an additional column for signals. Signals are global variables, and winIDEA can profile them with regular data areas.

Table 1: Trace techniques allow profiling of specific types of objects.

	Running Task ISR	**Task State Running ISR**	**Runnables**	**Signals**
Program Flow Trace			**iSYSTEM XML**	
Data Trace	**ORTI** **iSYSTEM XML**	**iSYSTEM XML**		**iSYSTEM XML**
Instrumented Data Trace		**iSYSTEM XML** OS Trace Hooks	**iSYSTEM XML** VFB Trace Hooks	
Instrumentation Trace	**iSYSTEM XML** OS Trace Hooks	**iSYSTEM XML** OS Trace Hooks	**iSYSTEM XML** VFB Trace Hooks	

The table shows which trace techniques can record a specific class of objects. Not all combinations are feasible. In such cases, the respective cell in the table is empty. For supported combinations, the field shows the winIDEA configuration file format in bold (either ORTI or iSYSTEM Profiler XML) and the underlying instrumentation technique (for instrumentation-based use-cases). Task and ISR profiling relies on OS Trace Hooks, and Runnable profiling uses the so-called VFB Trace Hooks.
All use-cases require a so-called ORTI (OSEK Runtime-Interface) file. The OS generates this file (usually it has an ORT or ORTI file ending), which includes information about OS objects such as Tasks, ISRs, and alarms. Additionally, for all use-cases except the basic Running Task/ISR profiling (and even there it may be necessary), a winIDEA specific configuration file called iSYSTEM Profiler XML is mandatory. This file informs the winIDEA Profiler about the profiling objects, the entity mapping (for example, the task ID to task name mapping), and further options such as BTF trace export configuration. iSYSTEM provides a tool called *iTCHi* (iSYSTEM trace configuration helper) to generate the XML file automatically.
With the trace objects and the underlying trace technology to record them available, the last step is to find out the operating system used by the application. iSYSTEM provides application notes for the most common AUTOSAR operating system vendors mentioned in the previous section. To summarize, go through the following three points.

- Specify the object classes you want to record.
- Verify the trace capabilities of your microcontroller.
- Find out which operating system your application uses.

Once you have this information available, you are ready to configure your profiling use-case.

3.2 Configure

With the assessment complete, you are now able to configure the OS and RTE, winIDEA, and the winIDEA Analyzer to profile the data based on your specific use-case. The first step is to enable ORTI support. Additionally, for instrumentation-based use-cases, the respective OS timing-hooks and RTE virtual function bus trace hooks must be equipped. Please, refer to the operating system specific application note for this step:

- Vector Microsar Profiling
- Elektrobit EB tresos AutoCore Profiling
- EB tresos Safety OS 2.x Thread Profiling

If your operating system is not on this list, feel free to reach out to iSYSTEM, and we can assist you with the setup. As long as you are using an AUTOSAR based operating system, winIDEA supports OS and RTE profiling.
The next step is to implement the instrumentation hooks if used and to make winIDEA aware of the profiling use-case. To create OS and RTE awareness, you have to generate an iSYSTEM Profiler XML file via iTCHi. You can find iTCHi in the scripts directory inside your winIDEA installation directory. The folder should include the iTCHi executable as well as a readme that explains the different use-cases, as shown in Table 1Table 2. The table is similar to the previous one, except it shows the respective iTCHi commands for the tracing technique/object class pairings. Refer to the iTCHi documentation for how to configure each of these commands. Note, that you can run multiple commands at the same time, for example, if you want to profile task states via instrumentation and Runnables via program flow trace the respective iTCHi call would be:

```
itchi-bin.exe --task_state_instrumentation --runnable_program_flow
```

Table 2: iTCHi commands for the different trace technique to trace object mappings.

	Running Task/ ISR	**Task State/ Running ISR**	**Runnables**
Program Flow Trace			runnable_program_flow
Data Trace	running_taskisr	task_state_simple_variable task_state_complex_expression	
Instrumented Data Trace		task_state_instrumentation	runnable_instrumentation
Instrumentation Trace		task_state_instrumentation	

After generating an iSYSTEM Profiler XML file, a Profiler Inspectors JSON file, and instrumentation code (do not forget to rebuild if you use instrumentation), you can move on to configure winIDEA. First, add the Profiler XML to winIDEA and then start with the Analyzer configuration. Depending on the use-case, you may have to configure the trace manually. Please consult the application notes mentioned above and watch the webinar for your operating system for detailed information:

- ETAS RTA-OS Profiling
- Instrumented Vector MICROSAR Profiling
- Non-Instrumented MICROSAR Profiling

Once you have completed these steps, you are ready to profile the OS and RTE of your application.

3.3 Profile

Congratulations, you can now run the microcontroller and start profiling your application. In case you have any questions, contact iSYSTEM via the contact information below.

4 Technical support

4.1 Online resources

Online Help ▸ winIDEA and testIDEA online help	Knowledge Base ▸ Tips & tricks categorized by issue type and architecture	Tutorials ▸ From a beginner to an 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

4.2 Contact

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