LAUTERBACH
DEVELOPMENT TOOLS

Application Note Profiling on AUTOSAR CP with ARTI

MANUAL

Application Note Profiling on AUTOSAR CP with ARTI

Application Note Profiling on AUTOSAR CP with ARTI

Version 06-Jun-2024

History

03-Oct-23	DAP Streaming for miniMCDS in chapter "TriCore Bus Trace" added.
01-Aug-23	PowerDebug E40/PRO/X50 and AUTO26 Debug Cable added as second configuration for TriCore DAP Streaming.
05-Jul-23	Initial version of the manual.

About this manual

This document provides information about using TRACE32 for performance analysis on systems based on the AUTOSAR Classic Platform.

Introduction

Starting with release R20-11, AUTOSAR includes the so-called "**AUTOSAR Run-Time Interface**" (ARTI), which is intended for debugging and tracing applications and the OS of the Classic Platform. It basically includes a description of the implemented OS and a hook interface for instrumented tracing.
For detailed information, please refer to the AUTOSAR specifications (see **"Related Documentation"**, page 6).

The OS description of ARTI is created in ARXML and is meant to be a successor of the "**OSEK Run-Time Interface**" (ORTI) description. As of today (July 2023), no commercial AUTOSAR stack provider supports the creation of the ARTI description yet, however TRACE32 already supports the import of the ARTI ARXML file. The ARTI features described in this document still rely on the available ORTI description, with some side notes on the upcoming ARTI description.

ARTI defined a new interface to trace events based on instrumented hooks. This was not available for ORTI profiling, which relies completely on hardware based data trace capabilities. Using instrumented hooks overcomes several limitations that a data trace has. The hook based interface allows to:

- trace all cores, even if there is a hardware limitation (e.g. TriCore MCDS, which only allows tracing for n cores out of r (n <= r))
- trace complex events that would overload the trace port (e.g. task states)
- trace specific AUTOSAR artifacts (e.g. runnables)
- trace only events of interest, increasing the trace depth drastically and allowing medium speed trace tools such as TRACE32 CombiProbe
- accomplish a pure software trace, if no hardware trace is available at all

This manual will show you how to instrument your software to use ARTI profiling on various trace methods, as well as the evaluation and analysis of the trace information generated by this instrumentation.

Related Documentation

1. AUTOSAR specifications
   - AUTOSAR_CP_SWS_OS.pdf (formerly AUTOSAR_SWS_OS.pdf)
   - AUTOSAR_CP_SWS_ARTI.pdf (formerly AUTOSAR_SWS_ClassicPlatformARTI.pdf)
   - AUTOSAR_CP_EXP_ARTI.pdf (formerly AUTOSAR_EXP_ClassicPlatformARTI.pdf)
   - AUTOSAR_FO_TR_TimingAnalysis.pdf (formerly AUTOSAR_TR_TimingAnalysis.pdf)
   - AUTOSAR_CP_TPS_TimingExtensions.pdf (formerly AUTOSAR_TPS_TimingExtensions.pdf)
2. TRACE32 documentation
   - **"General Commands Reference Guide T"** (general_ref_t.pdf): **Trace.STATistic** / **Trace.Chart**

Using ARTI Hooks

Hook Macros

The AUTOSAR CP ARTI specification includes hook macros called ‘ARTI_TRACE’.
This chapter describes these hooks. The OS or the RTE must include the hooks in the appropriate locations. The user can decide which events are of interest to him by switching on the individual hooks, The generation of the trace messages, appropriate to the selected TRACE32 trace method is then done by the implementation of the hook macro, provided by Lauterbach.

OS Hooks

By specification, the OS contains empty ARTI hooks for the following events of interest:

- task state changes
- ISR2 state changes
- spinlocks
- OS calls

Lauterbach offers ready-to-use OS hook implementations, see chapter **“Instrumentation”**, page 8. A hook is switched on by its implementation.

ARTI also defines ISR1 (interrupt) hooks. Since ISR1s are not part of the OS, the user must manually place the ISR1 hooks in his interrupt routines.

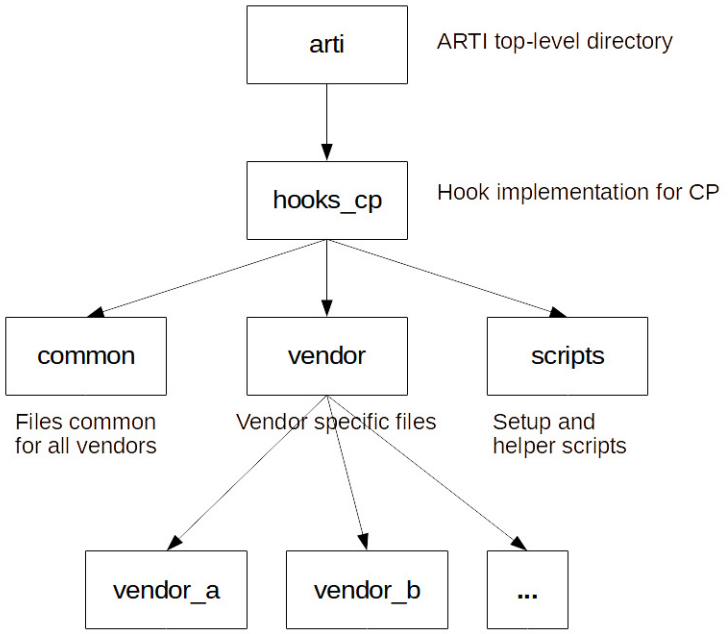
As of today (July 2023), most OSes do not natively contain ARTI hooks. Instead, each OS contains individual, proprietary hooks. The ARTI hook implementation for TRACE32 contains adapters to map these proprietary hook interfaces to ARTI (see chapter **“Instrumentation”**, page 8).

RTE Hooks

The ARTI hooks for the RTE mainly include the start and stop events of runnables. The AUTOSAR VFB (Virtual Function Bus) tracing hooks are used to realize the RTE hooks. This requires two steps to switch on the RTE hooks for the events “Runnable started” and “Runnable stopped”.

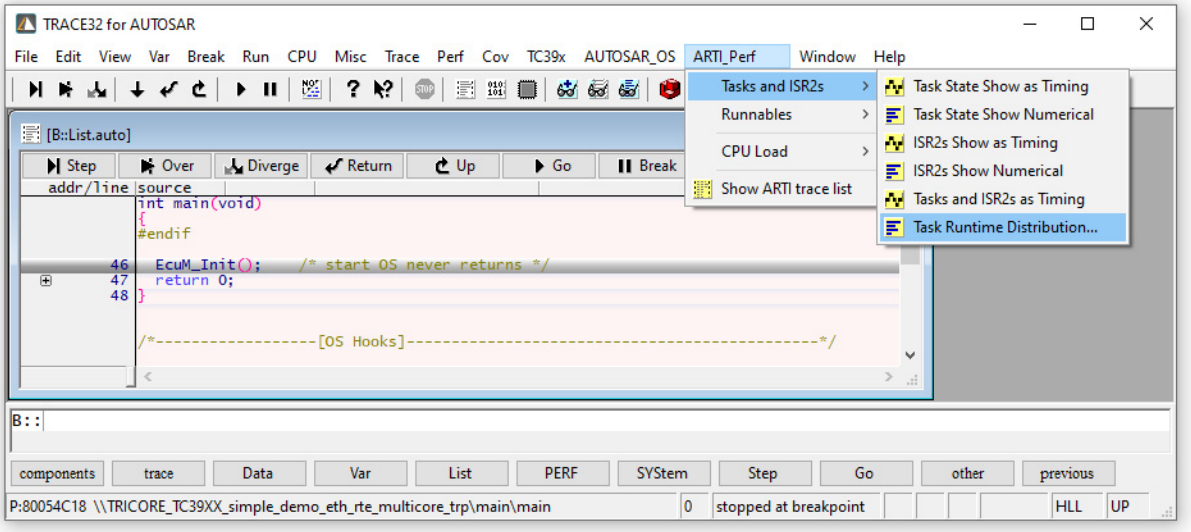
1. Enable the VFB tracing hooks in the configuration of the AUTOSAR system in general and for the individual runnables.
2. Use the RTE vendor specific Python script provided by Lauterbach that populates the VFB tracing hooks with ARTI hooks (see chapter **“Instrumentation”**, page 8).

You can find the ARTI hook implementation and TRACE32 adapters for the proprietary hooks in the TRACE32 installation folder, directory `~~/demo/kernel/arti/hooks_cp`.



To include the TRACE32 support for ARTI in your project, simply copy the `common` directory to your build environment, and add the `arti.c` file to your build artifacts. Within `arti.h`, select the trace method to use. Please also read the `readme.txt` file within the `common` directory.

The directory `scripts` contains scripts for the TRACE32 setup for the selected TRACE32 trace method. Additionally, you may execute the `arti_menu.cmm` script to add a menu item `ARTI_Perf` to easily access some of the features of TRACE32 for ARTI Profiling.



Trace Methods

Various TRACE32 trace methods can be used for ARTI profiling. The methods depend on the actually used core architecture and TRACE32 tool set. The TRACE32 trace methods differ significantly in their impact to the application. Some have almost no impact at all but depend on the availability and capability of the trace protocol of the core-under-test (e.g. data trace). Some have a high impact on memory consumption (e.g. **LOGGER**), and some have a high impact on timing (e.g. **FDX**). Please also read carefully the `readme.txt` files in the appropriate folder of the implementation.

Data Trace

Message generation: The trace message for the event-of-interest is generated by the core trace logic when a write access to the ARTI trace variables occur.

Trace sink: Onchip trace buffer or trace buffer within the TRACE32 trace tool.

Prerequisites:

- The core(s)-under-test must provide the capability to generate trace messages on write accesses to variables.
- A trace sink must be available, either on chip or in form of a TRACE32 trace tool.
- Very low impact in code size.
- Low impact with regards to the timing behavior.
- Simple TRACE32 trace setup, only trace filter on address ranges required. Please refer to your **Processor Architecture Manual**.
- **Onchip trace:** **chip timestamp** required, TRACE32 trace license required.
- **Parallel off-chip trace port:** TRACE32 Trace Tool required, CombiProbe possible if available for the core architecture under test, **TRACE32 Trace Streaming** possible without limitations.
- **Serial off-chip trace port:** TRACE32 Trace Tool required, **chip timestamp** required, **TRACE32 Trace Streaming** possible without limitations.

Caveats:

- Not suitable for TriCore AURIX, data tracing is not always possible for all cores.

Message generation: The trace message for the event-of-interest is generated by the MCDS, MCDSlight or miniMCDS when a write accesses to the ARTI trace variables that are located in the LMU space (TC2xx) or in the OLDA memory space (TC3xx, TC4x) occur. Via the **MCDS.MODULE.NAME()** function you can find out whether your AURIX TriCore microcontroller has one of the necessary MCDS variants.

Trace sink: Onchip trace buffer or trace buffer within the TRACE32 trace tool.

Prerequisites:

- Very low impact in code size, but adjustment to linker script required.
- Low impact with regards to the timing behavior.
- Complex TRACE32 trace setup, **CTL** trigger program required.
- **TRACE32 Trace Streaming** possible without limitations for DAP streaming and AGBT/SGBT serial trace.

Details of the trace sinks:

- **Onchip trace:** always possible if the microcontroller contains one of the necessary MCDS variants.
- **DAP streaming:** almost always possible if the microcontroller contains one of the necessary MCDS variants. Some pre-production microcontrollers include an MCDS variant, but are not capable of DAP streaming. In case of doubt, please contact your **local Lauterbach support**.

  There are two TRACE32 tool variants for DAP streaming:

  1. TRACE32 PowerDebug Module and Debug Cable

  PowerDebug Module with USB interface (PowerDebug E40 available since 06/2022 and all models to come thereafter) or TRACE32 PowerDebug with USB and Ethernet interface (PowerDebug PRO available since 12/2014 and all models to come thereafter)

  plus TRACE32 AUTO26 Debug Cable V3 (available since 04/2023 and all debug cable variants to come thereafter)

  2. TRACE32 medium range debug and trace tool CombiProbe otherwise
- **AGBT/SGBT serial trace:** TRACE32 high-end trace tool required, MCDS (TC2xx, TC3xx, TC4x) or MCDSlight (TC3xx) required

TRACE32 LOGGER Trace

Message generation: The trace message for the event-of-interest is generated by the **LOGGER** trace instrumentation in the target application when a write access to the ARTI trace variables occur.

Trace sink: Target memory, the **LOGGER** trace instrumentation stores the trace message in the target memory.

There are two implementations available: one for SMP and one for single-core. The SMP variant is free of spinlocks, but uses a inter-core buffer management that adds some overhead compared to the single-core variant.

- Suitable for all cores (that do not provide a core trace logic), but timestamp resource such as timer, counter required on the target.
- Low (medium) impact to code size by additional **LOGGER** instrumentation.
- Medium impact with regards to the timing behavior.
- Sufficient free target memory required for trace buffer implementation and some spare computing time for buffer handling.
- Simple TRACE32 trace setup, established **LOGGER** command group.
- No extra TRACE32 tool or TRACE32 trace license required.

The **"Application Note for the LOGGER Trace"** (app_logger.pdf) provides an introduction to the use of the logger trace.

TRACE32 FDX Trace

Message generation: The trace message for the event-of-interest is generated by the **FDX** trace instrumentation in the target application when a write access to the ARTI trace variables occurs.

Trace sink: Target memory, the **FDX** trace instrumentation writes the trace message to a small trace buffer that is located in the target memory. The FDX host application ensures that the data is transferred from the small target trace buffer to a large trace buffer on host computer while the program execution is running.

There are two implementations available: one for SMP and one for single-core. The SMP variant uses spinlocks, thus adding some time overhead compared to the single-core variant.

- Suitable for all cores (that do not provide a core trace logic, but enable runtime-memory access), timestamp resource such as timer, counter required on the target.
- Low (medium) impact to code size by additional **FDX** instrumentation.
- Medium impact with regards to the timing behavior.
- Some free target memory (typically 4K bytes) required for trace buffer implementation and some spare computing time for buffer handling.
- Complex TRACE32 trace setup, FDX host application has to be established.
- (Almost) unlimited trace time, because the host computer, on which the trace information is permanently transferred, allows very large trace buffers.
- No extra TRACE32 tool or TRACE32 trace license required.

The **"Application Note for FDX"** (app_fdx.pdf) provides an introduction to the use of the FDX trace.

- **Elektrobit**

  Elektrobit provides two different OS versions: AutoCore OS and SafetyOS (aka MikroOS).
  Please use the appropriate ARTI binding (eb_autocore or eb_microos).
  With AutoCore OS, you may need to adjust the ORTI file, see the `readme.txt` file.
  With SafetyOS, the ORTI file needs to be converted with a special script. See the `readme.txt` file.

- **ETAS**

  With ETAS RTA-OS, there are two different ARTI bindings, depending whether the application is multicore or single core. Please use the appropriate one.

- **Vector**

  For Vector's DaVinci Configurator and MICROSAR OS, there's a special "**Vector-Lauterbach-Timing-Bundle**" available in `~~/demo/env/vector/rte_profiling`. Please use this bundle for the ARTI profiling.

- **FreeRTOS**

  There is also an ARTI binding for FreeRTOS available. You need to configure FreeRTOS to include the OS tracing hooks. See the `readme.txt` file.
  FreeRTOS doesn't provide an **RTE**, so there is no RTE/runnable tracing available.

- **SafeRTOS**

  There is also an ARTI binding for SafeRTOS available. SafeRTOS doesn't provide OS hooks. A python script patches the SafeRTOS kernel sources to include the ARTI Hooks. See the `readme.txt` file.

In order to decode and analyze the recorded trace, TRACE32 has to map the recorded ARTI trace IDs to the actual AUTOSAR object (task “SchMComTask”, runnable start “Rte_Runnable_200ms”, etc.). By AUTOSAR specification, the ARTI ARXML description contains the mapping between the AUTOSAR objects and their ARTI trace IDs. Unfortunately, as of today (July 2023), the ARTI description is not yet created by the stack vendors. Thus the ARTI traceID mapping needs to come from another source.

- **Tasks**

  The task ID in the ARTI trace is directly mapped to the index of the task within the ORTI file. This means, when using ARTI, you always need to also load the accompanying up-to-date ORTI file. With some OSes, the build system does not directly create an “ARTI matching” ORTI file. In this case, the ORTI file needs to be preprocessed, see **Vendor Specifics** above. Remember that matching the ORTI file with the ARTI hooks is currently just a workaround for the missing ARTI description.

- **ISR2s**

  The ID for category 2 ISRs in the ARTI hooks is related to the index of the ISR2 in the ORTI file. For this, the ISR2s must be listed in the ORTI file in a special, defined way. Sometimes the ORTI file needs to be preprocessed after its creation to meet this need. See **Vendor Specifics** above.

- **Interrupts**

  Depending on the core architecture and the trace protocol, TRACE32 can identify interrupts (or category 1 ISRs) in the instruction trace recording. Some architectures provide extra interrupt notifications, on others the interrupts are detected by the access to the exception vector table. For this, however, the entire program flow would need to be recorded. A “pure” ARTI trace does not include this information.

  If category 1 ISRs are traced with the according ARTI hooks, the ID provided to the hook will serve as ISR1 ARTI trace ID. No further translation (e.g. interrupt name) is possible, as ISR1s names are listed nowhere.

- **Runnables**

  When creating the VFB Tracing functions that contain the ARTI hooks for runnables, additionally a script `rte_runnable.cmm` is generated. This script contains the declarations of the runnables and their ARTI trace IDs. Execute this script in TRACE32 to announce the ID-to-runnable mapping and to be able to identify the runnables in the further analysis.

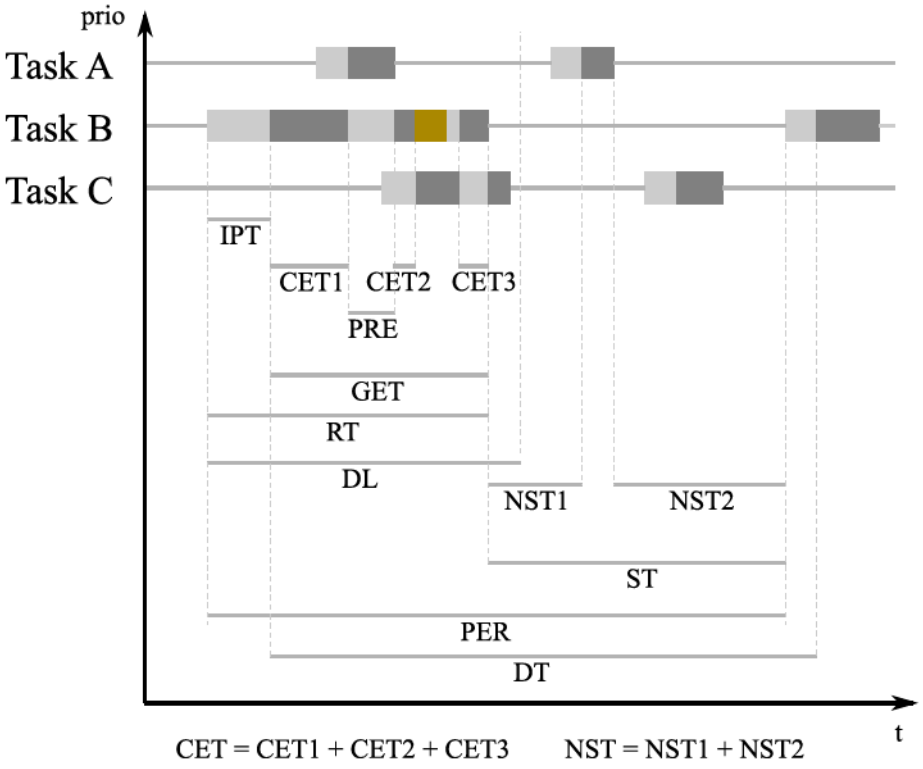
Timing Parameters

Trace recordings for ARTI profiling must always contain a timestamp for each event-of-interest. After recording a trace of runnables, tasks and/or ISRs, a set of timing parameters can be evaluated. These parameters include absolute run-times, minimum/maximum run-times, cyclic event measurement etc. For a list of available parameters, please refer to **Trace.STATistic** -> **"Parameters"** (general_ref_t.pdf) and **Trace.STATistic.TASKState**.

AUTOSAR defines its own set of parameters in the "Timing Analysis and Design" specification (AUTOSAR_FO_TR_TimingAnalysis.pdf). The chapter "AUTOSAR Classic Platform Operating System" therein provides the following parameters and their descriptions:

ID	Abr.	Name	Description
1	IPT	initial pending time	from activation to start
2	CET	core execution time (computation time)	execution time not including any preemptions or "waiting" time
3	GET	gross execution time execution	time including all preemptions and "waiting" time
4	RT	response time	from activation to termination
5	DL	dead line	max. allowed response time
6	DT	delta time	from start to start ("measured period")
7	PER	period	from activation to activation (period not as measured but as configured)
8	ST	slack time	"remaining" run-time: from termination to activation (tasks) or start (interrupts)
9	NST	net slack time	"potential additional" run-time: the ST minus all CET blocks of any task or ISRs with higher priority during the ST
10	JIT	jitter	deviation of delta time from period (not shown in the figure below)
11	PRE	preemption time	time a task is preempted by higher priority task(s) (not shown in the figure below)
12	CPU	CPU load	fraction of CPU time spent nonidle (usually reported in percent) (not shown in the figure below)

An overview of the timing parameters can be found in the following diagram:



Depending on the recorded events and the available trace data, TRACE32 is able to evaluate these events, too. Some of the **Trace.STATistic** commands allow the display of these parameters (by using the /**ARTI** option) or directly accept the AUTOSAR timing parameter as item. See the next chapter for details.

Overview of TRACE32 Command Structure

This chapter gives an overview of useful TRACE32 commands when working with AUTOSAR systems. It may also serve as a quick reference guide. For a detailed description of the individual commands see the general reference guides.

TASK.ARTI

The **TASK.ARTI** command group mainly manages the import of AUTOSAR ARXML files. The **Arti** module of AUTOSAR exports information about used task, runnables, hooks, and also some information about the internals of the OS to be able to perform a sophisticated debugging, tracing and profiling on systems running AUTOSAR Classic Platform. It is intended to replace and extend the outdated ORTI format.

TRACE32 is ready to import the ARTI information in ARXML format. Unfortunately, as of today (July 2023), no OS/RTE vendor creates ARTI ARXML data. This means, the **TASK.ARTI** command is there only for future use.

TASK.ORTI

The **TASK.ORTI** command group handles all information that comes with the ORTI file. ORTI is a debugging and tracing standard for OSEK OS based systems. It allows to view OS related information (e.g. tasks, alarms, etc.) and allows real-time tracing of tasks. See **"OS Awareness Manual OSEK/ORTI"** (rtos_orti.pdf) for detailed information. All known AUTOSAR stacks that provide an AUTOSAR OS are able to create an ORTI file when generating the OS.

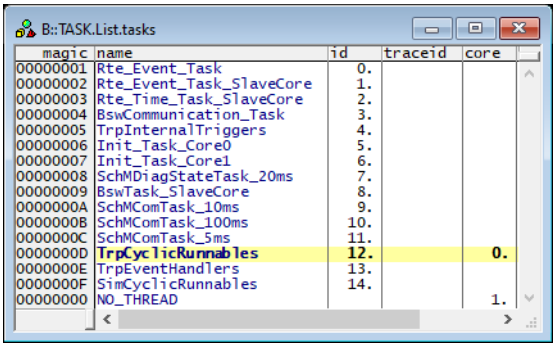
Note, that ORTI does **not** support any RTE features. In particular, ORTI does not know anything about runnables.

To evaluate traced events, the debugger needs to know about the mapping of the ARTI trace ID to the AUTOSAR object (see above chapter **"Object Detection"**, page 13). As long as the ARTI ARXML description is not available, information about tasks and ISR2s is taken out of the ORTI file. So ensure to have the ORTI file loaded any time using AUTOSAR related features.

The **TASK.List** command group lists objects that are relevant for profiling.

- **TASK.List.tasks**

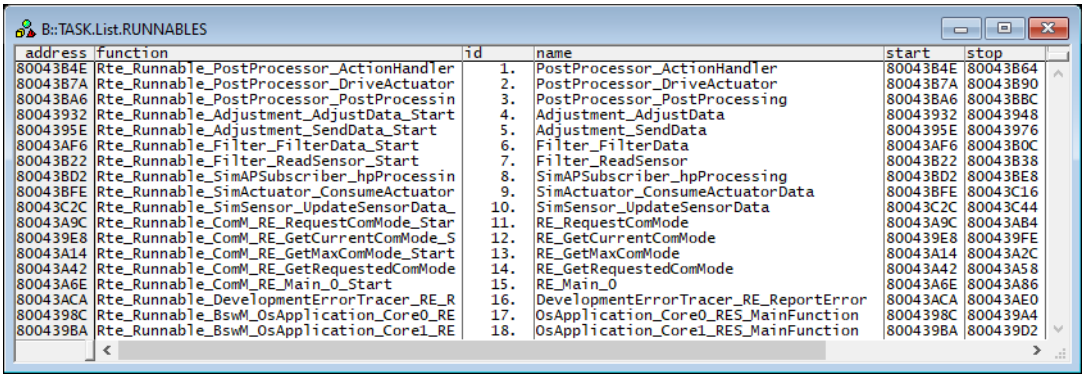
  This command shows all tasks that are known to the TRACE32 debugger. The debugger uses this table when creating reports (charts, statistics, exports) out of the trace. The task list is populated by the information from the ORTI file.



magic	name	id	traceid	core
00000001	Rte_Event_Task	0.		
00000002	Rte_Event_Task_SlaveCore	1.		
00000003	Rte_Time_Task_SlaveCore	2.		
00000004	BswCommunication_Task	3.		
00000005	TrpInternalTriggers	4.		
00000006	Init_Task_Core0	5.		
00000007	Init_Task_Core1	6.		
00000008	SchMDiagStateTask_20ms	7.		
00000009	BswTask_SlaveCore	8.		
0000000A	SchMComTask_10ms	9.		
0000000B	SchMComTask_100ms	10.		
0000000C	SchMComTask_5ms	11.		
0000000D	**TrpCyclicRunnables**	**12.**		**0.**
0000000E	TrpEventHandlers	13.		
0000000F	SimCyclicRunnables	14.		
00000000	NO_THREAD			1.

- **TASK.List.RUNNABLES**

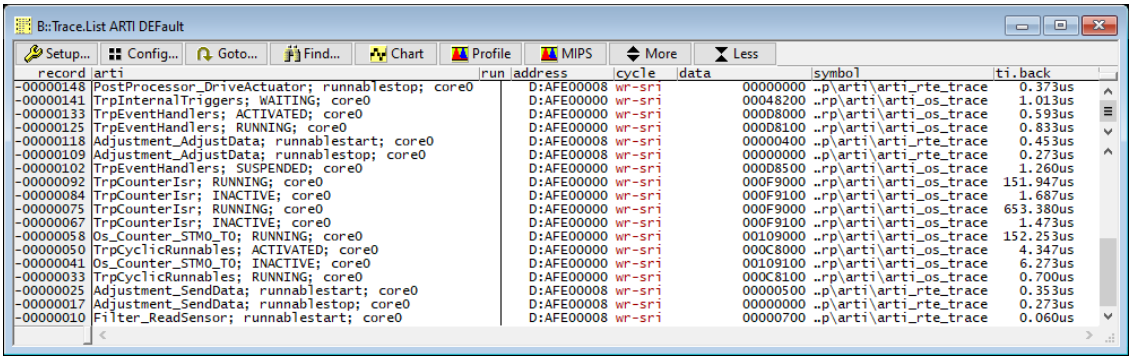
  This command shows all runnables known to the TRACE32 debugger, with their addresses and their IDs. The debugger uses this table when creating reports (charts, statistics, exports) out of the trace. The runnables have to be declared to the debugger with the command **TASK.Create.RUNNABLE**. Usually the vendor binding provided with the hook implementation contain a python script, that creates a PRACTICE script `rte_runnable.cmm` with all necessary runnable declarations during the RTE generation phase.



address	function	id	name	start	stop
80043B4E	Rte_Runnable_PostProcessor_ActionHandler	1.	PostProcessor_ActionHandler	80043B4E	80043B64
80043B7A	Rte_Runnable_PostProcessor_DriveActuator	2.	PostProcessor_DriveActuator	80043B7A	80043B90
80043BA6	Rte_Runnable_PostProcessor_PostProcessin	3.	PostProcessor_PostProcessing	80043BA6	80043BBC
80043932	Rte_Runnable_Adjustment_AdjustData_Start	4.	Adjustment_AdjustData	80043932	80043948
8004395E	Rte_Runnable_Adjustment_SendData_Start	5.	Adjustment_SendData	8004395E	80043976
80043AF6	Rte_Runnable_Filter_FilterData_Start	6.	Filter_FilterData	80043AF6	80043B0C
80043B22	Rte_Runnable_Filter_ReadSensor_Start	7.	Filter_ReadSensor	80043B22	80043B38
80043BD2	Rte_Runnable_SimAPSubscriber_hpProcessin	8.	SimAPSubscriber_hpProcessing	80043BD2	80043BE8
80043BFE	Rte_Runnable_SimActuator_ConsumeActuator	9.	SimActuator_ConsumeActuatorData	80043BFE	80043C16
80043C2C	Rte_Runnable_SimSensor_UpdateSensorData_	10.	SimSensor_UpdateSensorData	80043C2C	80043C44
80043A9C	Rte_Runnable_ComM_RE_RequestComMode_Star	11.	RE_RequestComMode	80043A9C	80043AB4
800439E8	Rte_Runnable_ComM_RE_GetCurrentComMode_S	12.	RE_GetCurrentComMode	800439E8	800439FE
80043A14	Rte_Runnable_ComM_RE_GetMaxComMode_Start	13.	RE_GetMaxComMode	80043A14	80043A2C
80043A42	Rte_Runnable_ComM_RE_GetRequestedComMode	14.	RE_GetRequestedComMode	80043A42	80043A58
80043A6E	Rte_Runnable_ComM_RE_Main_0_Start	15.	RE_Main_0	80043A6E	80043A86
80043ACA	Rte_Runnable_DevelopmentErrorTracer_RE_R	16.	DevelopmentErrorTracer_RE_ReportError	80043ACA	80043AE0
8004398C	Rte_Runnable_BswM_OsApplication_Core0_RE	17.	OsApplication_Core0_RES_MainFunction	8004398C	800439A4
800439BA	Rte_Runnable_BswM_OsApplication_Core1_RE	18.	OsApplication_Core1_RES_MainFunction	800439BA	800439D2

Trace.List

Trace.List shows the contents of the recorded trace. In case of an ARTI trace, by default it shows the raw data emitted to the trace by the instrumentation. The keyword `ARTI` will show an additional column in the **Trace.List** window that shows the decoded ARTI object details. Use **Trace.List ARTI DEFault** to display the decoded trace together with the recorded data. Please note that the decoding will only take place on trace data that contain ARTI data (e.g. ARTI specific variables like `arti_os_trace`). Records that do not belong to the ARTI trace (e.g. additional program flow trace) will be empty in the ARTI column.



record	arti	run	address	cycle	data	symbol	ti.back
-00000148	PostProcessor_DriveActuator; runnablestop; core0		D:AFE00008	wr-sri	00000000	..p\arti\arti_rte_trace	0.373us
-00000141	TrpInternalTriggers; WAITING; core0		D:AFE00000	wr-sri	00048200	..rp\arti\arti_os_trace	1.013us
-00000133	TrpEventHandlers; ACTIVATED; core0		D:AFE00000	wr-sri	000D8000	..rp\arti\arti_os_trace	0.593us
-00000125	TrpEventHandlers; RUNNING; core0		D:AFE00000	wr-sri	000D8100	..rp\arti\arti_os_trace	0.833us
-00000118	Adjustment_AdjustData; runnablestart; core0		D:AFE00008	wr-sri	00000400	..p\arti\arti_rte_trace	0.453us
-00000109	Adjustment_AdjustData; runnablestop; core0		D:AFE00008	wr-sri	00000000	..p\arti\arti_rte_trace	0.273us
-00000102	TrpEventHandlers; SUSPENDED; core0		D:AFE00000	wr-sri	000D8500	..rp\arti\arti_os_trace	1.260us
-00000092	TrpCounterIsr; RUNNING; core0		D:AFE00000	wr-sri	000F9000	..rp\arti\arti_os_trace	151.947us
-00000084	TrpCounterIsr; INACTIVE; core0		D:AFE00000	wr-sri	000F9100	..rp\arti\arti_os_trace	1.687us
-00000075	TrpCounterIsr; RUNNING; core0		D:AFE00000	wr-sri	000F9000	..rp\arti\arti_os_trace	653.380us
-00000067	TrpCounterIsr; INACTIVE; core0		D:AFE00000	wr-sri	000F9100	..rp\arti\arti_os_trace	1.473us
-00000058	Os_Counter_STM0_T0; RUNNING; core0		D:AFE00000	wr-sri	00109000	..rp\arti\arti_os_trace	152.253us
-00000050	TrpCyclicRunnables; ACTIVATED; core0		D:AFE00000	wr-sri	000C8000	..rp\arti\arti_os_trace	4.347us
-00000041	Os_Counter_STM0_T0; INACTIVE; core0		D:AFE00000	wr-sri	00109100	..rp\arti\arti_os_trace	6.273us
-00000033	TrpCyclicRunnables; RUNNING; core0		D:AFE00000	wr-sri	000C8100	..rp\arti\arti_os_trace	0.700us
-00000025	Adjustment_SendData; runnablestart; core0		D:AFE00008	wr-sri	00000500	..p\arti\arti_rte_trace	0.353us
-00000017	Adjustment_SendData; runnablestop; core0		D:AFE00008	wr-sri	00000000	..p\arti\arti_rte_trace	0.273us
-00000010	Filter_ReadSensor; runnablestart; core0		D:AFE00008	wr-sri	00000700	..p\arti\arti_rte_trace	0.060us

The screenshot was recorded for a TriCore AURIX and will look slightly different for other core architectures.

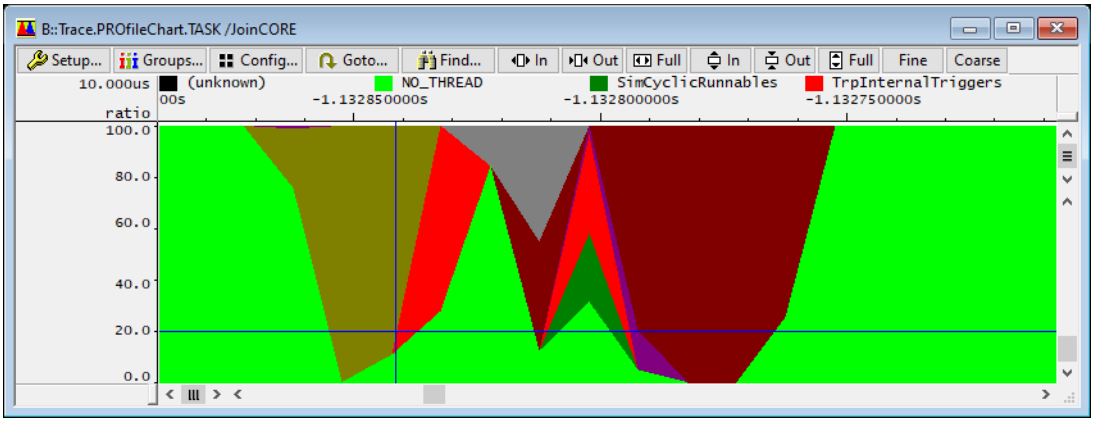
Trace.Chart

The **Trace.Chart** command group opens up timing chart windows for various ARTI objects. You can display the timing of task run times, task states, interrupts, runnables aso. Chart windows give a quick overview on the timing behavior. You can link several chart windows together with the /**Track** option.

The following chapters will give more details on the usage of **Trace.Chart** with tasks, ISR2s and runnables.

Trace.ProfileChart

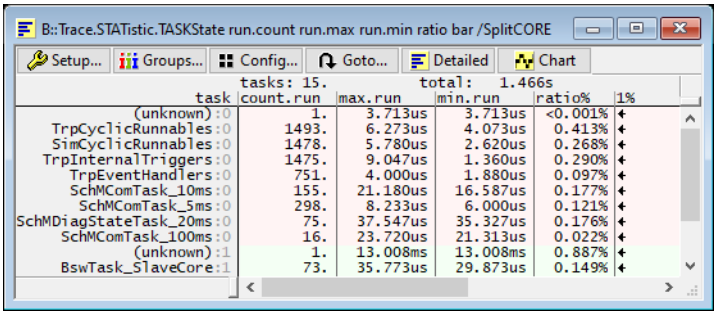
The command group **Trace.PROfileChart** shows in a color chart how much time an ARTI object (e.g. a task) has consumed within fixed time intervals. It is especially useful to give a quick overview of how much CPU load a specific portion of the code or object has consumed.



See the chapter **CPU Load Measurement** below how to use this command in AUTOSAR environments.

Trace.STATistic

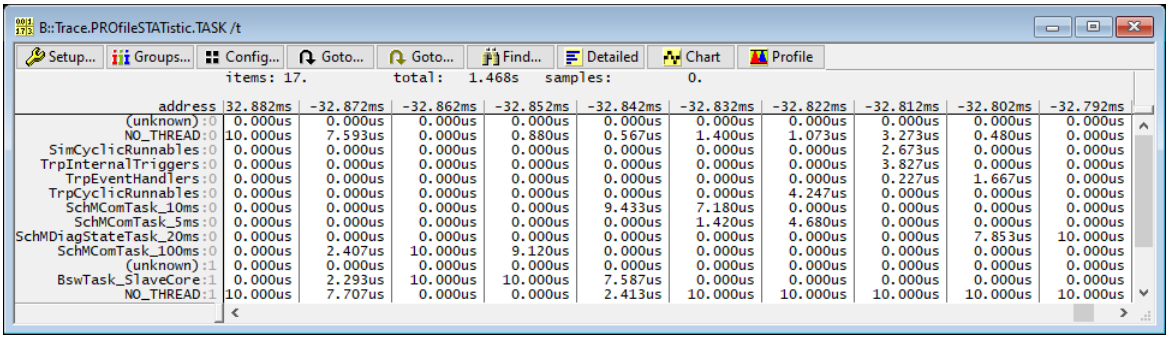
The **Trace.STATistic** command group provides tables with statistic evaluations of the timing of the individual ARTI objects. It is meant to work on various timing parameters, including those mentioned in chapter **"Timing Parameters"**, page 14. You will gain minimum, maximum and average times. When exporting the **Trace.STATistic** windows to a CSV file, it could be used to execute and check timing in CI (Continuous Integration) environments.



The following chapters will give more details on the usage of **Trace.STATistic** with tasks, ISR2s and runnables.

Trace.PROfileSTATistic

The **Trace.PROfileSTATistic** command group shows a table with timing parameters evaluated within fixed time intervals. This allows to detect peaks or trends of parameters over the recording time. It is especially useful to measure the CPU load in fixed time slots.

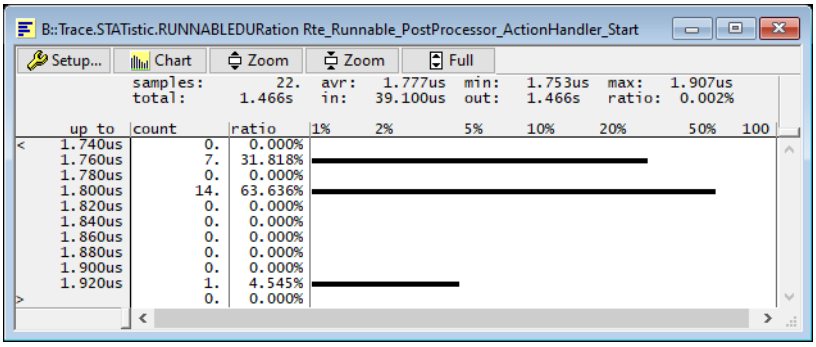


address	32.882ms	-32.872ms	-32.862ms	-32.852ms	-32.842ms	-32.832ms	-32.822ms	-32.812ms	-32.802ms	-32.792ms
(unknown):0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us
NO_THREAD:0	10.000us	7.593us	0.000us	0.880us	0.567us	1.400us	1.073us	3.273us	0.480us	0.000us
SimCyclicRunnables:0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	2.673us	0.000us	0.000us
TrpInternalTriggers:0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	3.827us	0.000us	0.000us
TrpEventHandlers:0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.227us	1.667us	0.000us
TrpCyclicRunnables:0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	4.247us	0.000us	0.000us	0.000us
SchMComTask_10ms:0	0.000us	0.000us	0.000us	0.000us	9.433us	7.180us	0.000us	0.000us	0.000us	0.000us
SchMComTask_5ms:0	0.000us	0.000us	0.000us	0.000us	0.000us	1.420us	4.680us	0.000us	0.000us	0.000us
SchMDiagStateTask_20ms:0	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	7.853us	10.000us
SchMComTask_100ms:0	0.000us	2.407us	10.000us	9.120us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us
(unknown):1	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us	0.000us
BswTask_SlaveCore:1	0.000us	2.293us	10.000us	10.000us	7.587us	0.000us	0.000us	0.000us	0.000us	0.000us
NO_THREAD:1	10.000us	7.707us	0.000us	0.000us	2.413us	10.000us	10.000us	10.000us	10.000us	10.000us

See the chapter **CPU Load Measurement** below how to use this command in AUTOSAR environments.

DURation Analysis

The duration analysis evaluates and shows the time distribution between two events. It is especially useful to display the distribution of execution times, be it tasks or runnables.

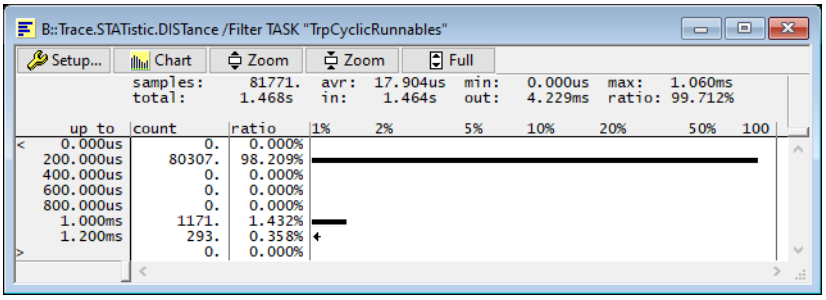


up to	count	ratio	1% 2% 5% 10% 20% 50% 100
< 1.740us	0.	0.000%	
1.760us	7.	31.818%	
1.780us	0.	0.000%	
1.800us	14.	63.636%	
1.820us	0.	0.000%	
1.840us	0.	0.000%	
1.860us	0.	0.000%	
1.880us	0.	0.000%	
1.900us	0.	0.000%	
1.920us	1.	4.545%	
>	0.	0.000%	

The following chapters will give more details on the usage of the duration analysis with tasks and runnables, introducing the commands **Trace.STATistic.RUNNABLEDURation** and **Trace.STATistic.TASKStateDURation**.

DIStance Analysis

The command **Trace.STATistic.DIStance** evaluates and shows the time distribution between the occurrences of a single event.



See the chapter **Jitter Measurement** below how to use this command to measure jitters in AUTOSAR environments.

SMP Options

With multicore systems (SMP) there are several options for time evaluation:

- **SplitCORE**

  With the option /**SplitCORE** the time for each core is calculated and displayed separately. In AUTOSAR Classic Platform the execution of runnables or tasks is strictly bound to a specific core, so this option is useful to explicitly see what happens on each core.

- **MergeCORE**

  Using the /**MergeCORE** option, the time is calculated for each core, but the display summarizes the results of all cores for each item. Executable entities in AUTOSAR classic platform usually do not run on different cores, so this option usually is not needed here. It may be useful for functions that are used cross-core.

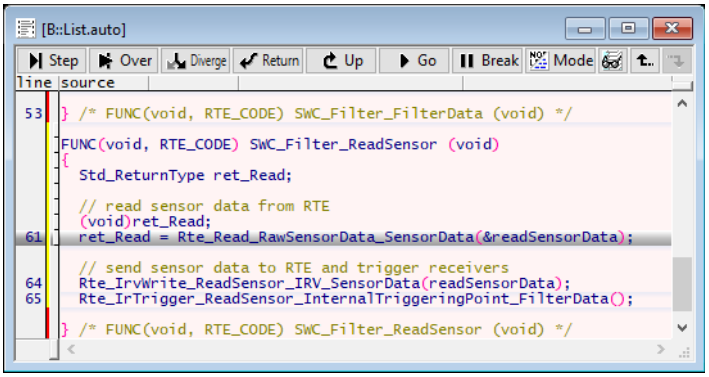
- **JoinCORE**

  The /**JoinCORE** option causes the analysis to ignore the core. This option is useful to measure timings of individual events that may happen on different cores. It is not useful to measure run-times that may overlap on the cores.

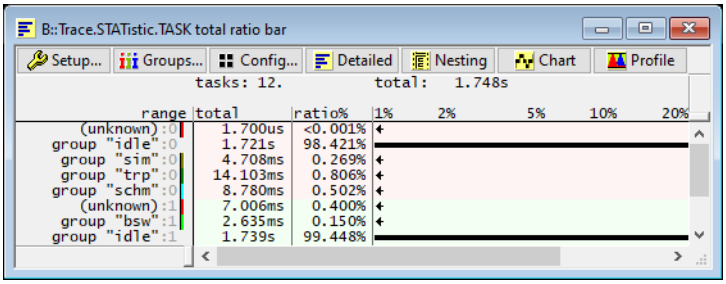
GROUP

The **GROUP** command group is mainly used for two purposes: marking and merging. **GROUP** allows to group address ranges (such as functions or modules) or tasks.

You can assign a color to a specific group. Whenever a window shows portions of the group (e.g. a **List** window showing the code of a function that belongs to a group), the scale area at the left will show this color as a bar. This allows identifying quickly, to which group your currently looking at. E.g. you could group a set of functions and/or runnables together into a group that represents your AUTOSAR SWC.



More important for the use case of profiling is the possibility of merging group members in trace analysis windows. This gives a better overview of your system, if you have hundreds (or thousands) of objects to watch for. You may also hide all groups that are not of interest, and concentrate on the one you're responsible for. It also gives the possibility to assign all background and idle tasks to an “idle group”, calculating the CPU load used by all other tasks. See chapter **CPU Load Measurement** below.



BMC

The **BMC** command group allows access to on-chip performance monitoring capabilities. Sometimes they are called “benchmark counters” or “performance counters”. The capabilities of the **BMC** command group heavily depends on the CPU architecture and the built-in features. In this application note, benchmark counters are only covered by the chapter **“CPU Load Measurement”**, page 34.

Trace.EXPORT

You may want to import the results of the timing analysis done in TRACE32 into external tools that are able to do further analysis or formatting. E.g. you may use the data of a task utilization to create a pie chart in your favorite spreadsheet application.

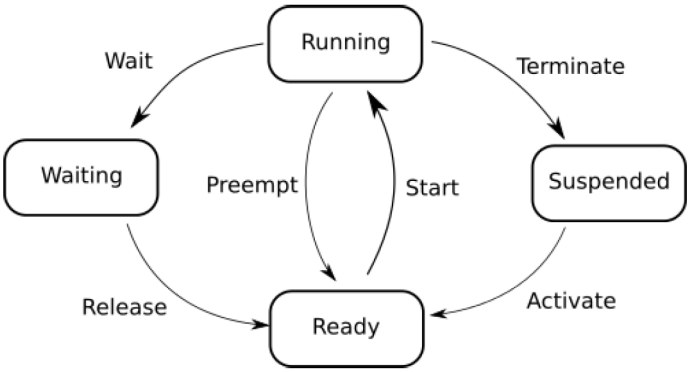
TRACE32 provides several ways of exporting data, the easiest (and most flexible) one being the **PRinTer.FILE** command with the CSV format. Together with the **WinPrint** prefix, this command allows to export **any** command into a CSV table. The exported CSV table then can be imported in any application that understands CSV.

For systems running AUTOSAR Classic Platform, the command group **Trace.EXPORT** is available to interpret and export the recorded ARTI trace to various formats. These exports are dedicated for external timing analysis tools that allow e.g. scheduling analysis and requirement tracing.

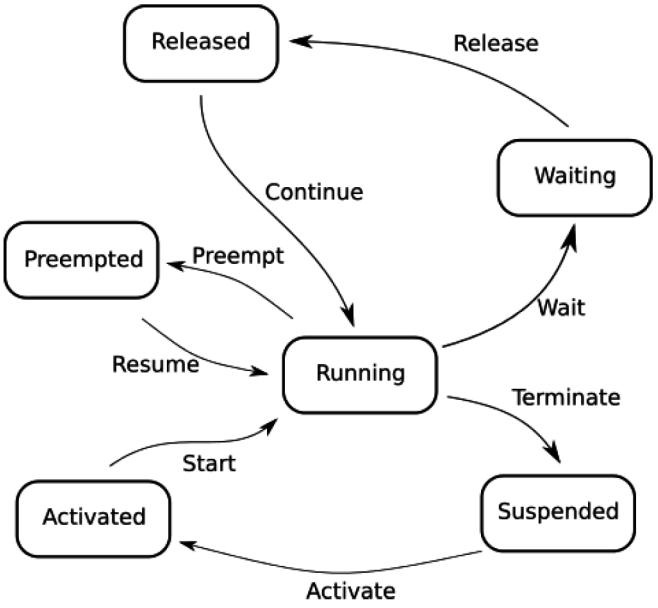
Details about the different exports that are of interest for AUTOSAR are mentioned in the below **Export** chapter.

The ARTI instrumentation together with the appropriate tracing method records all task state changes. The task states in the ARTI recording follow the task state machines defined in the AUTOSAR OS specification. The chapter '**ARTI Hook Macros**' therein defines a `standard` state machine and an `enhanced` state machine.

Standard state machine:



Enhanced state machine:



Which state machine is used depends on the ARTI implementation. The ARTI implementation, in turn, depends on the OS and which hooks the OS provides. TRACE32 is able to decode both implementations, but, of course, only analyzes and provides those task states, that are encoded in the standard or enhanced model.

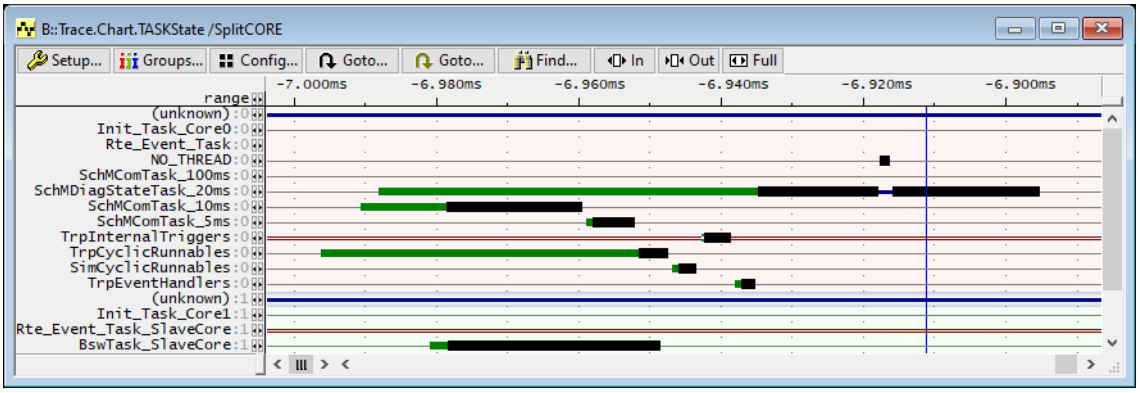
You can check with **Trace.List** **ARTI DEFault**, if the trace contains appropriate ARTI records, and if those are decoded correctly.

Please note: the commands **Trace.Chart.TASK** and **Trace.STATistic.TASK** are **not** suitable for task runtime analysis with ARTI. Those commands **only** evaluate the `Running` state, ignoring all others. E.g. the min/max/avg times state, how long a task was in a running state, not the time from activation to termination. Those commands, however, may be used in calculations where only the `Running` state is of interest, e.g. when calculating the CPU load.

Trace.Chart.TASKState

The command **Trace.Chart.TASKState** draws a timing chart with all tasks and their timeline of the states. The states are encoded as follows:

state	line	graphics
running	solid black bar	
ready, released, preempted	medium blue bar	
activated	green line	
waiting	two thin red lines	
suspended	thin grey line	
unknown	no line	

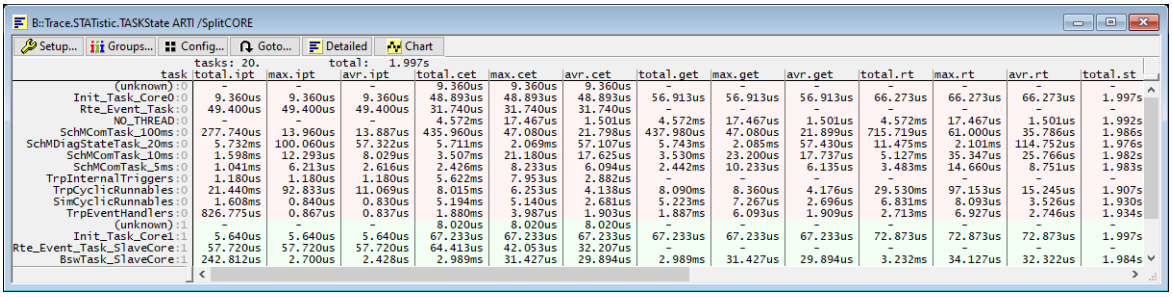


At the beginning of the trace, the states of the tasks are unknown. Only with the first appearance of a state transition of a task the TRACE32 debugger knows about the subsequent state.

Trace.STATistic.TASKState

The command **Trace.STATistic.TASKState** opens a table with all tasks and timing parameters based on the timeline of the task states. The parameters are listed with their maximum, minimum, average run time together with a ratio how much CPU time the task consumed.

Without any parameter, the command will show timing parameters as defined by TRACE32. However, AUTOSAR defines a different set of timing parameters (see chapter **"Timing Parameters"**, page 14). Adding the option /**ARTI** to the command will open a statistic evaluation based on the most interesting AUTOSAR parameters. The option /**AllARTI** lists all available items with ARTI. You can also pick the items of interest to show only those, e.g. **IPT.MAX PER.MIN CET.AVeRage**.



tasks: 20. total: 1.997s

task	total.ipt	max.ipt	avr.ipt	total.cet	max.cet	avr.cet	total.get	max.get	avr.get	total.rt	max.rt	avr.rt	total.st
(unknown):0	-	-	-	9.360us	9.360us	9.360us	-	-	-	-	-	-	-
Init_Task_Core0:0	9.360us	9.360us	9.360us	48.893us	48.893us	48.893us	56.913us	56.913us	56.913us	66.273us	66.273us	66.273us	1.997s
Rte_Event_Task:0	49.400us	49.400us	49.400us	31.740us	31.740us	31.740us	-	-	-	-	-	-	-
NO_THREAD:0	-	-	-	4.572ms	17.467us	1.501us	4.572ms	17.467us	1.501us	4.572ms	17.467us	1.501us	1.992s
SchMComTask_100ms:0	277.740us	13.960us	13.887us	435.960us	47.080us	21.798us	437.980us	47.080us	21.899us	715.719us	61.000us	35.786us	1.986s
SchMDiagStateTask_20ms:0	5.732ms	100.060us	57.322us	5.711ms	2.069ms	57.107us	5.743ms	2.085ms	57.430us	11.475ms	2.101ms	114.752us	1.976s
SchMComTask_10ms:0	1.598ms	12.293us	8.029us	3.507ms	21.180us	17.625us	3.530ms	23.200us	17.737us	5.127ms	35.347us	25.766us	1.982s
SchMComTask_5ms:0	1.041ms	6.213us	2.616us	2.426ms	8.233us	6.094us	2.442ms	10.233us	6.135us	3.483ms	14.660us	8.751us	1.983s
TrpInternalTriggers:0	1.180us	1.180us	1.180us	5.622ms	7.953us	2.882us	-	-	-	-	-	-	-
TrpCyclicRunnables:0	21.440ms	92.833us	11.069us	8.015ms	6.253us	4.138us	8.090ms	8.360us	4.176us	29.530ms	97.153us	15.245us	1.907s
SimCyclicRunnables:0	1.608ms	0.840us	0.830us	5.194ms	5.140us	2.681us	5.223ms	7.267us	2.696us	6.831ms	8.093us	3.526us	1.930s
TrpEventHandlers:0	826.775us	0.867us	0.837us	1.880ms	3.987us	1.903us	1.887ms	6.093us	1.909us	2.713ms	6.927us	2.746us	1.934s
(unknown):1	-	-	-	8.020us	8.020us	8.020us	-	-	-	-	-	-	-
Init_Task_Core1:1	5.640us	5.640us	5.640us	67.233us	67.233us	67.233us	67.233us	67.233us	67.233us	72.873us	72.873us	72.873us	1.997s
Rte_Event_Task_SlaveCore:1	57.720us	57.720us	57.720us	64.413us	42.053us	32.207us	-	-	-	-	-	-	-
BswTask_SlaveCore:1	242.812us	2.700us	2.428us	2.989ms	31.427us	29.894us	2.989ms	31.427us	29.894us	3.232ms	34.127us	32.322us	1.984s

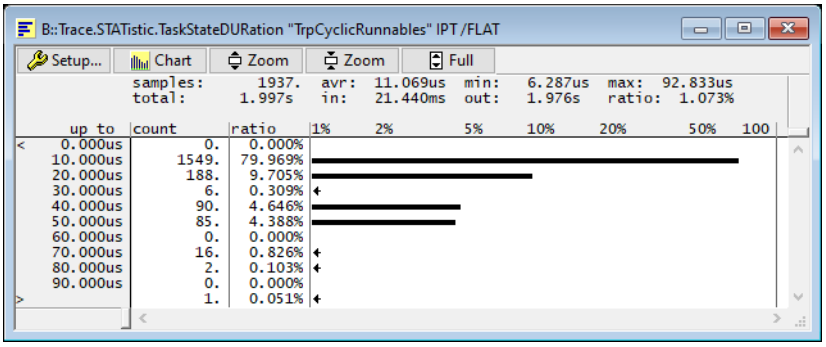
Trace.STATistic.TASKStateDURation

The command **Trace.STATistic.TASKStateDURation** takes a task and a timing parameter and displays the time duration of this parameter for the given task. See chapter **"Timing Parameters"**, page 14, for the available parameters.

```
; list the different time spans of the initial pending time in a flat manner
Trace.STATistic.TASKStateDURation "OsTask_RteTask_10ms" IPT /FLAT
```

Useful parameters are:

Parameter	Measurement
IPT	duration from activation to start
DT	duration from start to start
PER	duration from activation to activation
CET	core execution time duration
GET	gross execution time duration



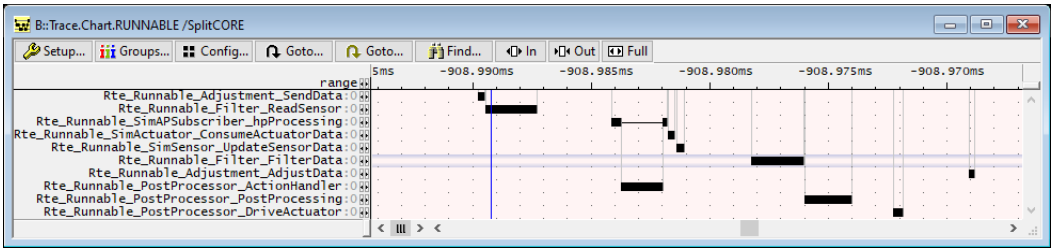
Runnable Runtime Analysis

If the ARTI instrumentation also includes the RTE (see chapter **"RTE Hooks"**, page 7), then the entries and exits of the instrumented runnables will be recorded. This allows an evaluation of the timing of runnables.

You can check with **Trace.List** **ARTI DEFault**, if the trace contains appropriate ARTI records, and if those are decoded correctly.

Trace.Chart.Runnable

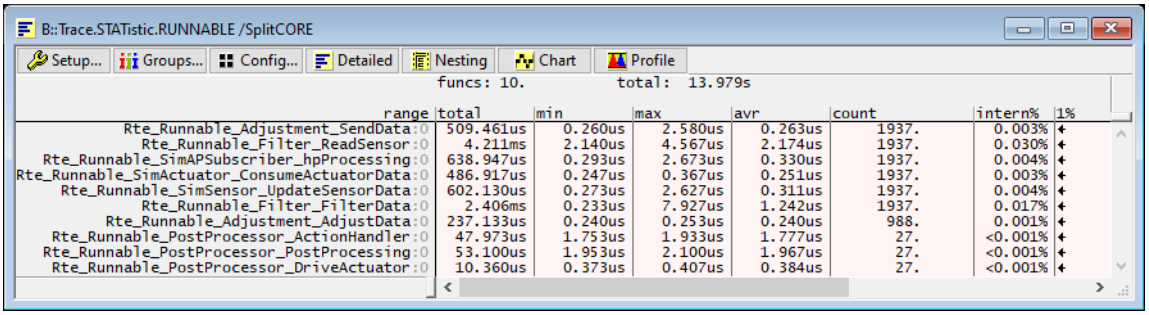
The command **Trace.Chart.RUNNABLE** draws a timing chart with all instrumented runnables and their timeline.



At the beginning of the trace, it is unknown, which (or even if a) runnable is running. Only with the first appearance of an entry or exit of a runnable the debugger knows about the timing.

Trace.STATistic.RUNNABLE

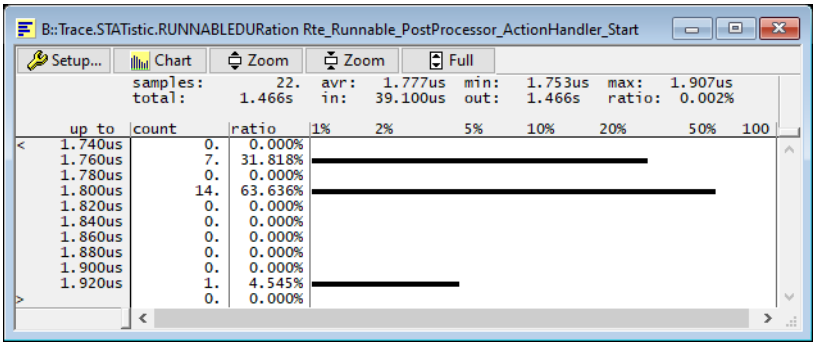
The command **Trace.STATistic.RUNNABLE** opens a table with all instrumented runnables and timing parameters based on the timeline of the runnables. The parameters are listed with their maximum, minimum, average run time together with a ratio how much CPU time the runnable consumed.



Trace.STATistic.RUNNABLEDURation <runnablestart>

The command **Trace.STATistic.RUNNABLEDURation** takes the start address of the runnable VFB tracing hook as parameter and displays the time duration of the given runnable. Check with **TASK.List.RUNNABLES** for the exact naming of your runnable.

```
; list the different time spans of the runtime of runnable MyRunnable
MyRunnable.Trace.STATistic.RUNNABLEDURation MyRunnable
```



ISR2 Runtime Analysis

If the ARTI instrumentation includes tracing of category 2 ISRs (ISR2), all entries and exits of ISR2s are recorded. This allows an evaluation of the timing of ISR2s.

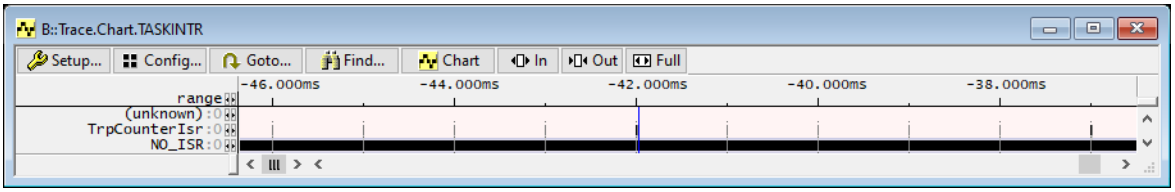
The **Trace** command group uses a different naming for ISR2s. Please note the term “interrupt” therein refers to ISR1s, while the term “INTR” or “TASKINTR” refers to ISR2s.

ISR2s are very implementation specific. While ARTI includes a clear definition and interface for ISR2s, some OSes internally do not make a clear distinction between tasks and ISR2s. In this case usually the ORTI file needs a modification to better separate ISR2s from tasks, in addition a proper ARTI hook implementation must be used. See also chapters **“Vendor Specifics”**, page 12, **“Object Detection”**, page 13 and **“TASK.ORTI”**, page 16.

You can check with **Trace.List** **ARTI DEFault**, if the trace contains appropriate ARTI records, and if those are decoded correctly.

Trace.Chart.TASKINTR

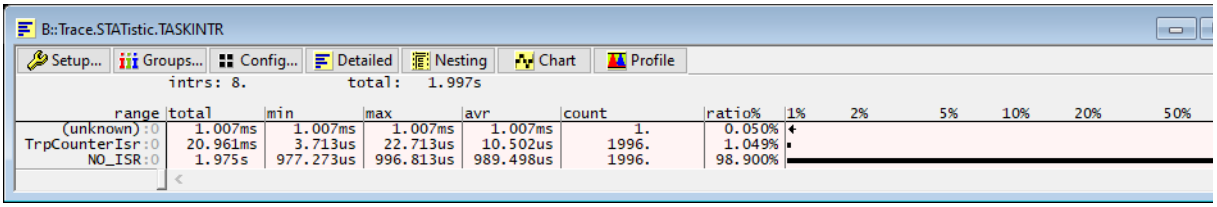
The command **Trace.Chart.TASKINTR** draws a timing chart with all ISR2s and their timeline.



At the beginning of the trace, it is unknown, if an ISR2 is active. Only with the first appearance of an entry or exit of an ISR2, the debugger knows about the timing.

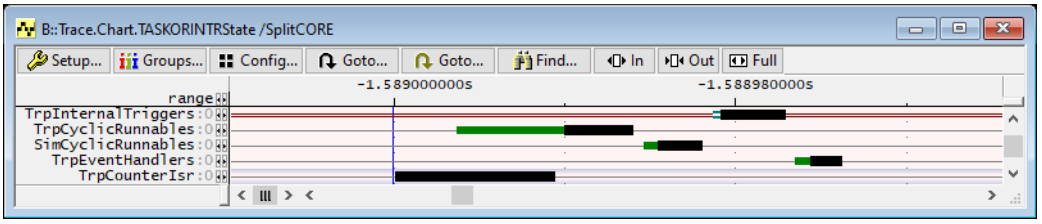
Trace.STATistic.TASKINTR

The command **Trace.STATistic.TASKINTR** opens a table with all ISR2s and timing parameters based on the timeline of the ISR2s. The parameters are listed with their maximum, minimum, average run time together with a ratio how much CPU time the ISR2 consumed.



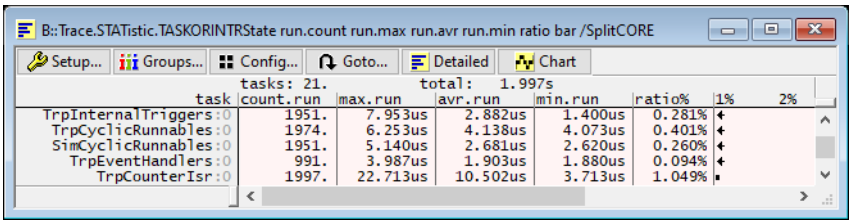
Trace.Chart.TASKORINTRState

The command **Trace.Chart.TASKORINTRState** displays the timeline of both, ISR2s and task states, side-by-side. This allows a quick overview if and when an ISR2 interrupted a task.



Trace.STATistic.TASKORINTRState

The command **Trace.STATistic.TASKORINTRState** opens a table with both, ISR2s and task states, and displays the timing parameters based on the timeline of the ISR2s and tasks. It is basically a merge of **Trace.STATistic.TASKState** and **Trace.STATistic.TASKINTR**. It allows to analyze ISR2s and tasks both together in one single table.



Interrupt Runtime Analysis

Tracing of interrupt routines (aka category 1 ISRs, aka ISR1s) is very architecture specific and not (yet) handled by this application note.

Spinlock Analysis

ARTI allows to trace spinlocks, if the OS supports the appropriate hooks. The Spinlock analysis is currently not (yet) coverd by this application note.

CPU Load Measurement

By measuring the task run times you can also compute the CPU load by calculating, how many time shares the CPU does “nothing”, runs in an idle loop, and/or runs in an idle task.

Please note that the CPU load is always related to a time span, and can change over time. If, for example, the trace covers 10 seconds, the overall CPU load will simply say, how much time within these 10 seconds the CPU was active. But this does not mean, that the CPU never was overloaded. In this scenario, a CPU load of 50% could mean, in the worse case, that it had 100% load in the first 5 seconds and 0% load in the second 5 seconds. On the other hand, if you take the time slot too small, e.g. smaller than the run-time of a runnable, you will **always** see slots with a CPU load of 100% (executing the runnable). But that's, of course, not the CPU load you're interested in. At the end it is very important to check the overall CPU load against loads measured with wisely set time slots.

Grouping the Idle Tasks

To measure the CPU load, or, in turn, the idle time, the debugger needs to know, which tasks count as "idle". If no task is running at all, the ORTI specification defines a dummy task named "NO_TASK" to be set to running. Systems may include an own idle task, or background tasks that should also be counted as "idle". To be able to calculate the idle time, group all idle tasks together into an "idle" group, using the **GROUP.CreateTASK** command, e.g.:

```
GROUP.CreateTASK "idle" "NO_TASK" "myIdleTask" "myBackgroundTask"
```

For a visual effect when analyzing the idle time later, you may colorize the group against all other tasks:

```
GROUP.COLOR “idle” NONE
GROUP.COLOR “other” RED
```

To add up the times when in idle time and when not in idle time, use the **GROUP.Merge** command:

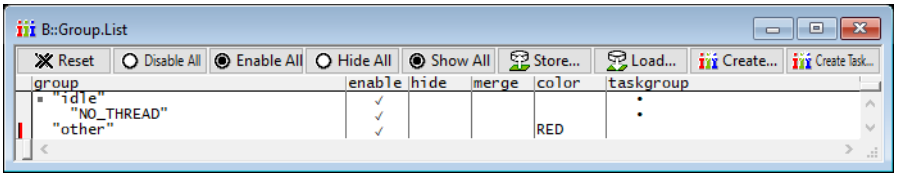
```
GROUP.Merge “idle”
GROUP.Merge “other”
```

The time spent in “other” then relates to the CPU load.

The merging of the group members applies to all **Trace** windows. If you want to switch back to see the timing of the individual members, cancel the merging with:

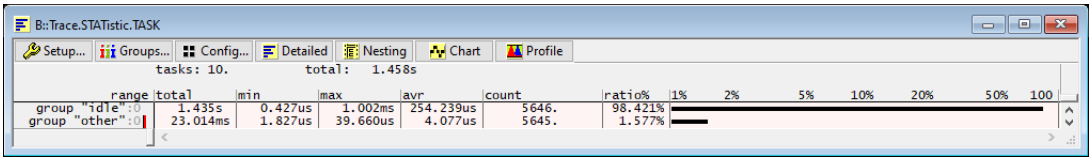
```
GROUP.SEParate “idle”
GROUP.SEParate “other”
```

The **GROUP.List** window gives you a dialog where you can set all these items interactively.

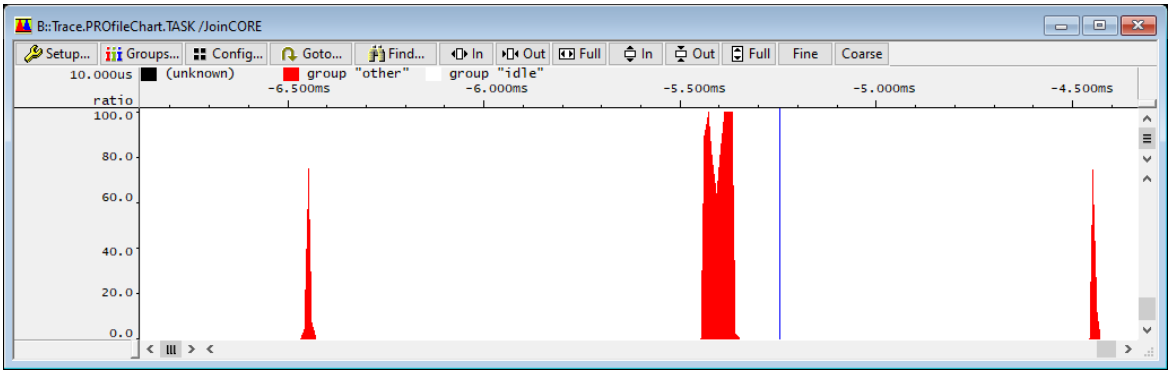


CPU Load Overview

After grouping and merging the idle tasks, you can get a quick overview of the CPU load within the recorded trace. **Trace.STATistic.TASK** now shows the absolute time spent within the “idle” and “other” tasks, as well as their ratio in CPU run time. The overall CPU load is hereby simply the ratio of the “other” group.



To get a quick overview of the CPU load over time, use the command **Trace.PROfileChart.TASK** with the option /**JoinCORE**. This command will provide a colorized chart showing how much percentage the CPU load was within fixed time intervals. The /**JoinCORE** option sums up the CPU load of all cores. If you set the colors of the group as mentioned in chapter **“Grouping the Idle Tasks”**, page 34, you'll see the CPU load in red, while the idle time is white. To further elaborate the timing within the time intervals, see the next chapter **CPU Load in Time Slots**.

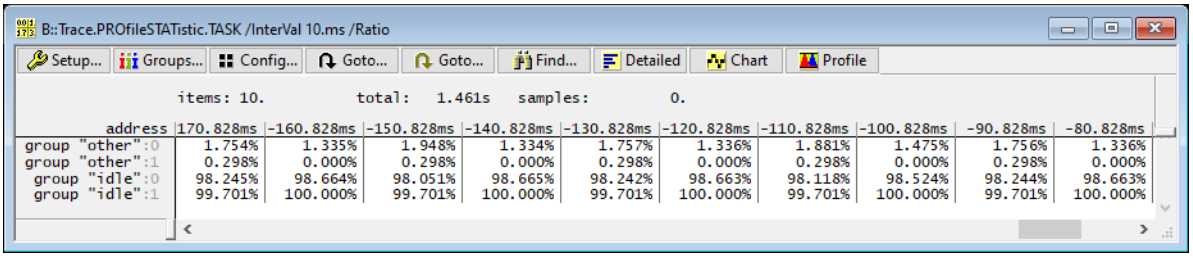


CPU Load in Time Slots

If the traced system has fixed time slots, it is useful to measure the CPU load within these slots. You can quickly identify time slots that do not fulfil the requirements on CPU load. To measure the CPU load in fixed time slots, merge the groups as mentioned in chapter **“Grouping the Idle Tasks”**, page 34. Open statistic evaluation of task with the interval of the time slot to list all slots with the ratio of the CPU load. E.g:

```
Trace.PROfileSTATistic.TASK /Interval 10.ms /Ratio
```

If you identified a time slot that exceeds your limits, open a **Trace.List ARTI DEFault /Track** window and click on the item in the table. The trace listing will scroll to the record in the trace that caused this calculation, and you can further analyze why the CPU load was so high within this time interval.



address	-170.828ms	-160.828ms	-150.828ms	-140.828ms	-130.828ms	-120.828ms	-110.828ms	-100.828ms	-90.828ms	-80.828ms
group "other":0	1.754%	1.335%	1.948%	1.334%	1.757%	1.336%	1.881%	1.475%	1.756%	1.336%
group "other":1	0.298%	0.000%	0.298%	0.000%	0.298%	0.000%	0.298%	0.000%	0.298%	0.000%
group "idle":0	98.245%	98.664%	98.051%	98.665%	98.242%	98.663%	98.118%	98.524%	98.244%	98.663%
group "idle":1	99.701%	100.000%	99.701%	100.000%	99.701%	100.000%	99.701%	100.000%	99.701%	100.000%

CPU Load by Benchmark Counters

On some architectures, depending on the built-in chip capabilities, it is possible to calculate the CPU load by benchmark counters (command group **BMC**). This is useful if no trace hardware is available.

To use the benchmark counters on TriCore TC3xx, you need a **full** MCDS implementation (MCDSlight or miniMCDS is not sufficient). The counters will work on executed instructions, **not** on time ticks. This means, you can calculate the CPU load on the number of executed instructions, but not on “real” timing. As the execution time of a instruction is not fixed but depends on the action, the measured value is not accurate in timing. Due to limitations in MCDS, you can measure only one core, and you can only specify one task as “idle” task. The demo directory for the ARTI instrumentation also contains a script `cpu_load_tc3xx_bmc.cmm` that calculates the CPU load based on benchmark counters.

Jitter Measurement

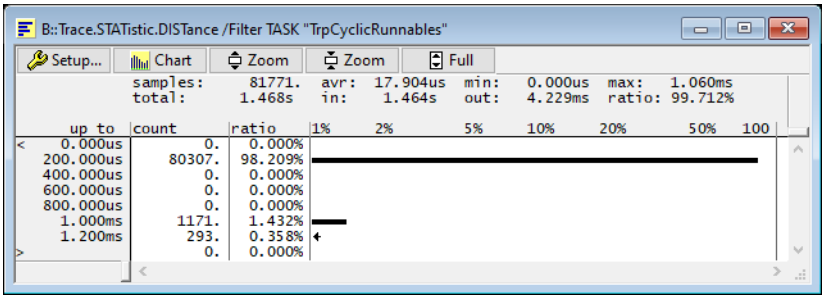
Jitter is the deviation of the actual measured event to the true periodicity. It is the time between the theoretical periodic event time and the actual time the event happened. The deviation can happen due to higher priority events (e.g. interrupts) or even deviating clock signals.

In AUTOSAR, the timing parameter “JIT” is defined as the “deviation of delta time from period”, i.e. the difference of the start-to-start time to the activation-to-activation time.

Jitter is best measured with the **Trace.STATistic.DIStance** command. Use the /**Filter** option to select a specific task, event, and/or core.

Jitter on Tasks

If you just want to measure the jitter of task run times (delta time, DT) use the command **Trace.STATistic.DIStance** /**Filter Task** <*task*>



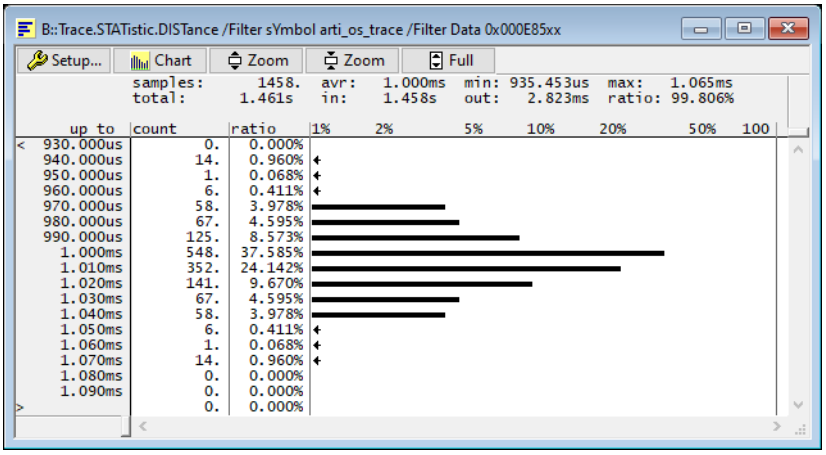
For measuring other events, you have to filter directly on the encoded ARTI data. The encoding of the traced data is:

```
arti_os_trace = (task_id << 16) | 0x8000 | (event_id << 8) | core_id
```

E.g if you want to measure the jitter of the event “Terminate” (ID 5) of task with ID “2” on core 1, use:

```
Trace.STATistic.DIStance /Filter sYmbol arti_os_trace /Filter Data /0x00028501
```

You can ignore fields, e.g. to ignore the core in this example, use `0x000285xx`.



It is intended to provide a script and dialog for the encoding of tasks and events, however, this is currently not yet available.

Jitter on Runnables

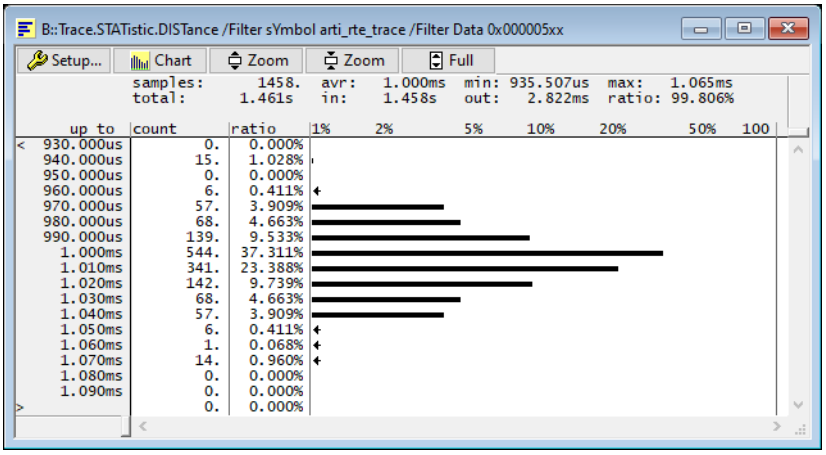
The Jitter on Runnables measures the deviation of the actual periodicity of the runnables (start-to-start). Use **Trace.STATistic.DIStance** with a filter encoding the ARTI data of the runnable event. The encoding of the traced data is:

```
arti_rte_trace = (runnable_id << 8) | core_id
```

E.g if you want to measure the jitter of the runnable with ID “4” on core 1, use:

```
Trace.STATistic.DIStance /Filter sYmbol arti_rte_trace /Filter Data /0x00000401
```

You can ignore fields, e.g. to ignore the core in this example, use `0x000004xx`.



It is intended to provide a script and dialog for the encoding of runnables, however, this is currently not yet available.

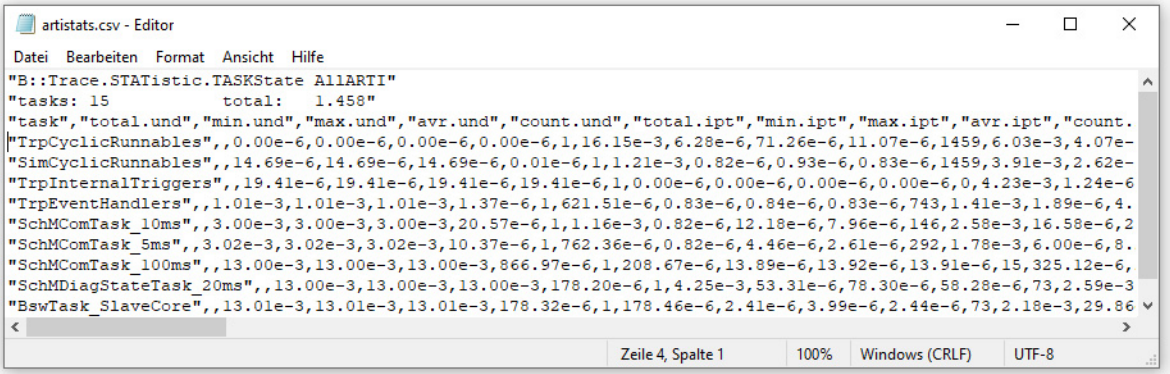
Export

If you want to post-process the traced data, there are various ways to export both, the trace and the statistic evaluations.

CSV Export

Remember, that **every** window in TRACE32 can be exported to a image or a textual representation. Use the **PRinTer.FILE** command to specify the file name and the file format of the destination. The following command with the **WinPrint** prefix will then save its contents into this file using the specified format. E.g. if you want to export all ARTI timing parameters into a CSV file named `artistats.csv`, use the commands:

```
PRinTer.FILe artistats.csv CSV
WinPrint.Trace.STATistic.TASKState AllARTI
```



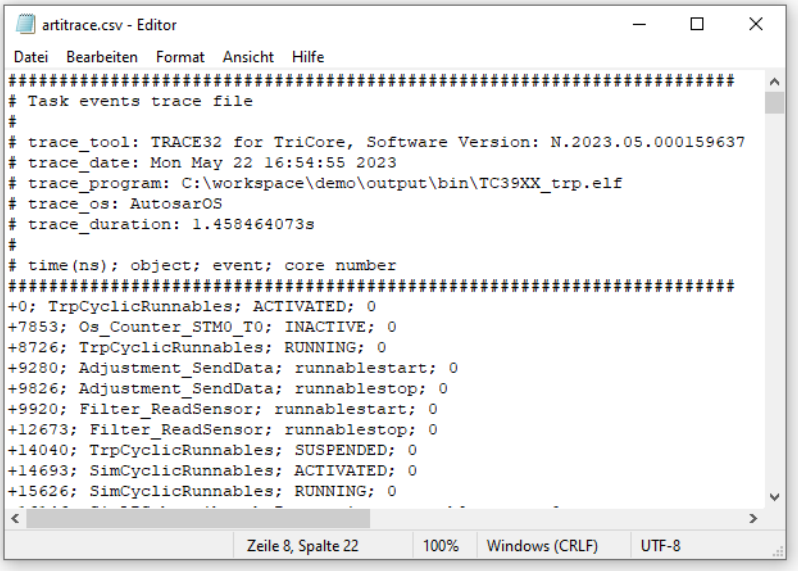
Trace.EXPORT.TASKEVENTS (deprecated)

TRACE32 includes a proprietary export format for instrumented task tracing.
Trace.EXPORT.TASKEVENTS is able to detect, decode and export this format. The “TASKEVENTS” format is still available only for backward-compatibility, for new projects please use the ARTI format.

Trace.EXPORT.ARTI

The recorded ARTI trace can be exported into a well-defined CSV format using the **Trace.EXPORT.ARTI** command. The exported file is a trace file in a textual CSV format, containing all decoded ARTI information, especially the task state changes and runnable start/stop events. Several timing analysis tools are able to import this format, please contact the tool vendor or Lauterbach for more information about tool compatibility.

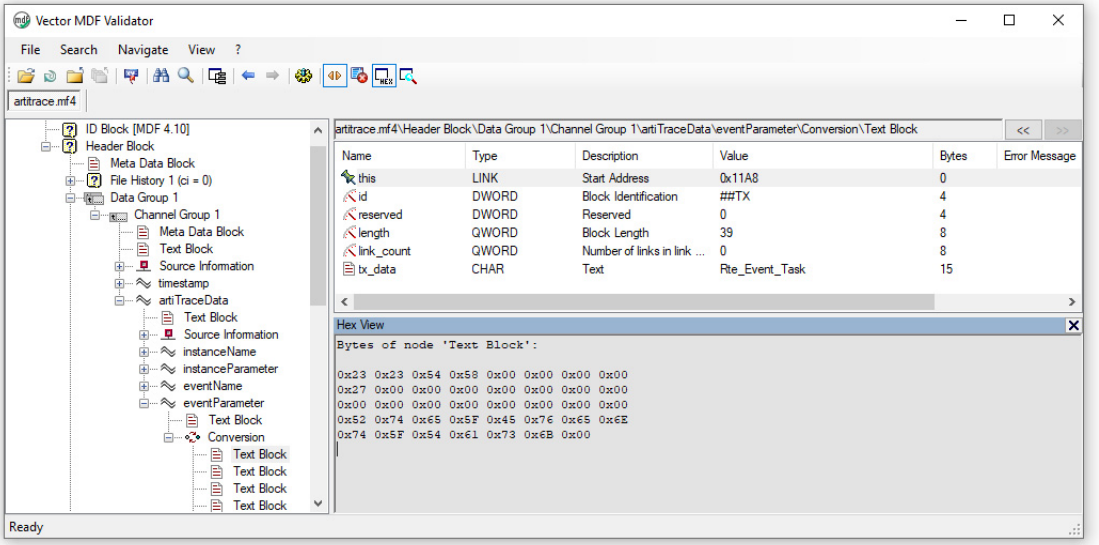
Please note that ARTI specifies two different task state machines: a “standard” and an “enhanced” state machine (see chapter **“Task Runtime Analysis”**, page 24). Depending on which state machine the ARTI implementation uses, specify the appropriate option /**STanDard** or /**ENHanced** to the export command.



Trace.EXPORT.MDF

The recorded ARTI trace can be exported into an MDF file as specified by the “ASAM Run-Time Interface Base Standard” (ASAM ARTI BS). The exported file is a binary file containing all decoded ARTI information, especially the task state changes and runnable start/stop events. Timing analysis tools that conform to this standard are able to import this file. Exported traces can become rather big, use the /**ZIP** option to reduce the file size.

Please note that ARTI specifies two different task state machines: a “standard” and an “enhanced” state machine (see chapter **“Task Runtime Analysis”**, page 24). Depending on which state machine the ARTI implementation uses, specify the appropriate option /**STanDard** or /**ENHanced** to the export command.



TIMEX

TIMEX is an AUTOSAR specification that allows to define timing events, event chains and especially timing constraints in an AUTSAR XML format. This allows tools to evaluate the measured times against requirements. While TRACE32 is able to measure some of the TIMEX artifacts (like “execution time constraints”), it does not import TIMEX descriptions. Especially TRACE32 does not do any requirement analysis.

To use TRACE32 for requirement analysis on TIMEX, perform the ARTI tracing as mentioned in this document. Export the trace either with **Trace.EXPORT.ARTI** or **Trace.EXPORT.MDF** (see chapters above). Use your favorite timing requirement analysis tool to import the TIMEX file and the exported trace.

TRACE32 can be easily automated and scripted, including all the functionality described in this document. So you can even include such a requirement analysis in your CI/CT environment.