MICROSAR Runtime Measurement

Technical Reference

Version 11.00.00

Authors	visazb, visore, viszda
Status	Released

Document Information

History

Author	Date	Version	Remarks
visazb	2013-05-02	1.00.00	Initial version
visore	2013-08-06	1.00.01	ESCAN00068380, ESCAN00069596
visore	2013-11-29	1.01.00	ESCAN00069152, ESCAN00072340, ESCAN00068381
visore	2014-05-21	1.01.01	ESCAN00075161
viszda	2014-09-22	1.02.00	ESCAN00070189, ESCAN00076463
viszda	2014-12-04	1.02.01	ESCAN00079842, ESCAN00079844, ESCAN00079535
viszda	2015-12-03	2.00.00	ESCAN00085574, ESCAN00083646
viszda	2016-03-10	2.00.01	ESCAN00088550
viszda	2016-05-17	2.01.00	
viszda	2016-09-21	2.02.00	ESCAN00091859, ESCAN00092231
viszda	2017-02-03	2.02.01	ESCAN00093857
viszda	2017-04-07	3.00.00	STORYC-724
viszda	2017-04-21	3.01.00	STORYC-671, STORYC-672
viszda	2017-06-13	3.01.01	ESCAN00095484, ESCAN00091207
vishr	2017-11-09	3.01.02	Implemented Review findings
viszda	2020-12-01	4.00.00	SWAT-82: Removed section ‘Component History’. Added example for OS_VTH_SCHEDULE.
viszda	2021-03-17	5.00.00	SWAT-1285, SWAT-1289, SWAT-1291: Added multiple sections regarding CPU load histograms, task response time histograms and task stack usage.
viszda	2021-05-24	6.00.00	SWAT-1468: Reworked section ‘Measurement Result Overflow’. Added section ‘Memory mapping’.
viszda	2021-06-16	6.01.00	SWAT-1545: Reworked section ‘Memory mapping’.
viszda	2021-08-11	6.02.00	SWAT-1611: Renamed measurement metrics.
viszda	2021-09-03	7.00.00	SWAT-1591: Remove description of special memory mapping handling ESCAN00110264: Add limitation for CPU load measurement
viszda	2021-11-17	7.01.00	SWAT-1737: Migrate Asr4Rtm design to text-based format
viszda	2021-12-17	7.02.00	SWAT-1683: Refactor the CPU load measurement mechanism

viszda	2022-01-27	7.03.00	SWAT-1599: Calculate Task Response Time between Activation and Termination
viszda	2022-03-24	8.00.00	SWAT-727: Explain how C_API works SWAT-2012: Support C-API access to all MPs
viszda	2022-07-18	8.00.01	SWAT-2047: Tidy up (explain how to get MP id). Removed memory mapping sections due to refactoring.
viszda	2022-10-07	9.00.00	SWAT-2235: Tidy up (complete sentence for ItemValuePtr and improve description of chapter Measurement on multi core system) SWAT-2298: Persist all measurement results via NvM ESCAN00112984
viszda	2022-11-25	9.01.00	SWAT-2378: Support of Logical Execution Time (LET) ESCAN00113220: Removed memory mapping section
viszda	2023-02-13	9.01.01	ESCAN00113723 (correct description of Rtm_Shutdown) ESCAN00114139 (rework chapter 3.5 A2L due to new A2L workflow) ESCAN00113925 (added limitation for Vector OS) SWAT-2371 (fix of typos, LET MPs are optionally created, persisting the RTM results explained, core specific Rtm_Init mentioned, mention RTM comfort editor) ESCAN00114587: MaxRunTimeInUs reported is unexpectedly lower than the percentiles calculated in the task response time histogram. ESCAN00114649 : Missing limitation that each OS core must be represented by one RtmCoreDefinition. ESCAN00115072: MPs used to measure Task response time have certain limitations.
mgourabathi	2023-06-29	10.00.00	SWAT-2751: Add description about the parameters need to set in OsDebug. SWAT-2749: Rtm trunk: Improving the handling of multiple NET and FET MPs in a Task. SWAT-2486: Add missing values in Table 4-20 of MpSetting. SWAT-2857: Generator: Add validator that /RtmGeneral/Rtm32BitTimer is true if /RtmMeasurementPoint/RtmTargetRuntime is too high.

mgourabathi	2023-09-05	11.00.00	SWAT- 2851:Rtm: Support Flat and Net MPs for Multi-Core. SWAT-2887: Deprecated Rtm_Enter/ Leave, TaskFct(), IsrFct() functionality. ESCAN00115299: Unexpected high MaxRunTimeInUs reported in Task Response time histogram for extended Tasks during ECU Startup.

Reference Documents

No.	Source	Title	Version
[1]	Vector	User Manual AUTOSAR Calibration	1.0
[2]	Vector	UserManual AMD – MICROSAR 4	1.0.0 or later
[3]	Vector	Technical Reference MemMap	See delivery
[4]	Vector	Technical Reference Rte	See delivery
[5]	Vector	Technical Reference OS	See delivery

This technical reference describes the general use of the Monitoring RTM basis software module.

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module RTM.

Supported AUTOSAR Release*:	4	
Supported Configuration Variants:	pre-compile	
Vendor ID:	RTM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	RTM_MODULE_ID	255 decimal

* For the precise AUTOSAR Release 4.x please see the release specific documentation.

Runtime Measurement (RTM) allows the user to determine runtimes and CPU load of BSW modules and user code sections.

RTM provides a set of macros, which are used to instrument the source code to be measured. Such an instrumented code section is called measurement point (MP).

Measurement is controlled- and evaluated in CANoe by RTM’s frontend or a self-written RTM application. Data exchange between CANoe and the ECU is done by the XCP protocol (e.g. using CAN, FlexRay or Ethernet network communication).

1.1 Architecture Overview

The Figure 1-2 shows the interfaces to adjacent modules of RTM. These interfaces are described in chapter 4.

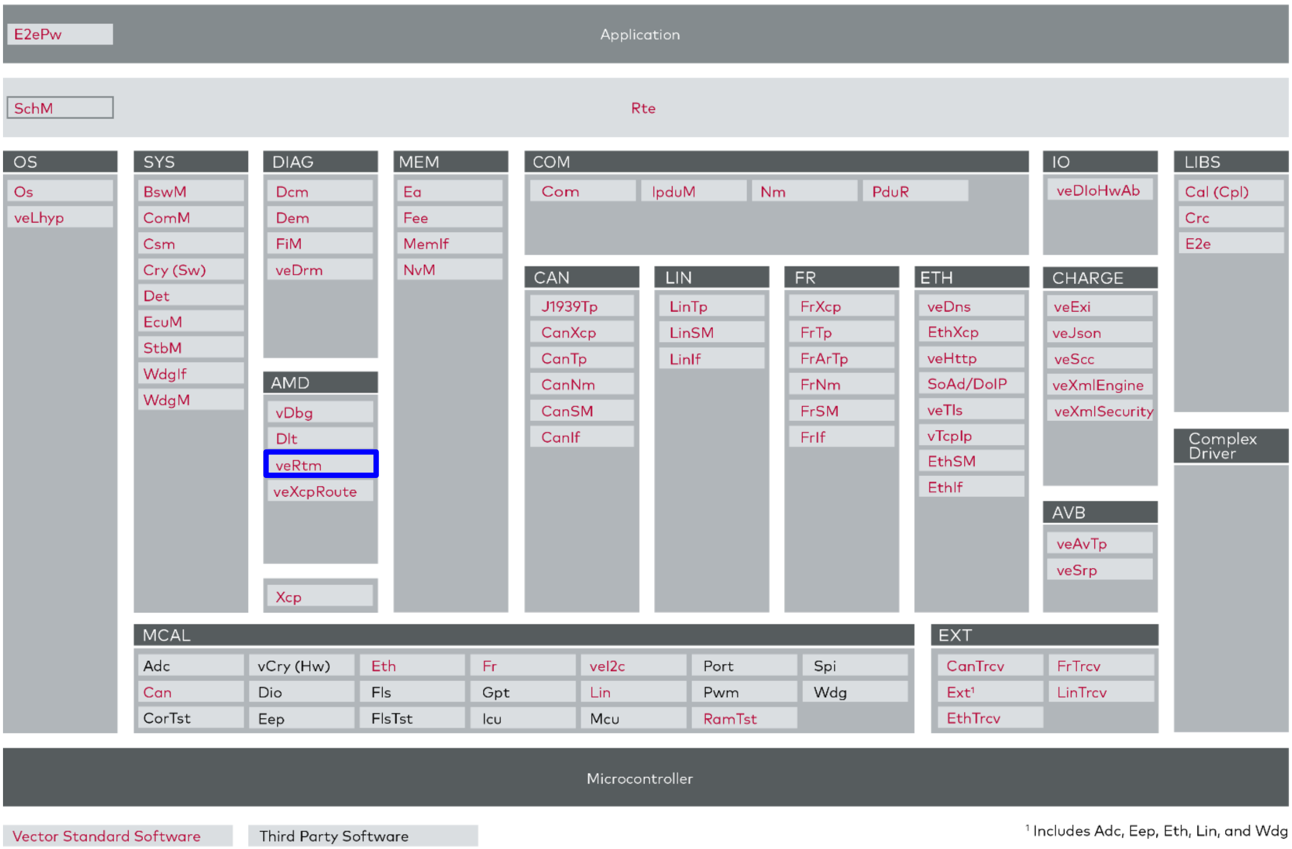


Figure 1-1 AUTOSAR 4.x Architecture Overview

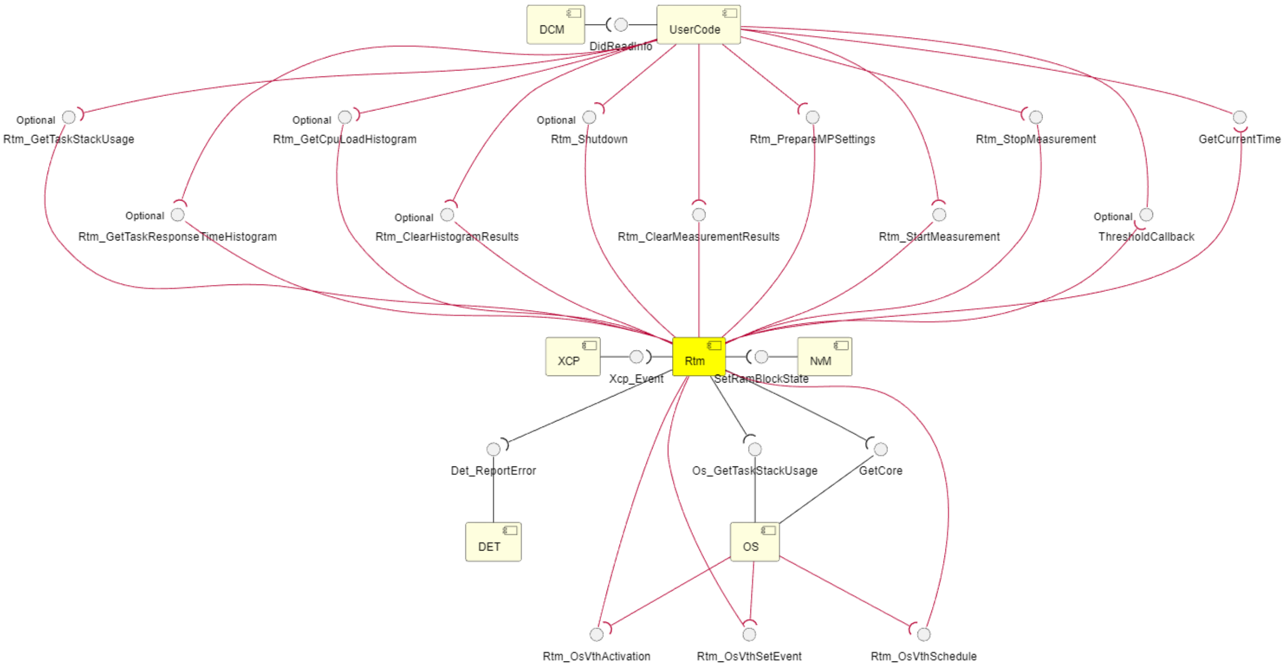


Figure 1-2 Interfaces to adjacent modules of the RTM

2 Functional Description

2.1 Features

RTM allows several MPs within the ECU's embedded code. Each MP is identified uniquely, allowing measurement of several MPs simultaneously without interference. MPs can be deactivated at pre-compile time and thus introduce no overhead in this case.

To minimize the impact of RTM's own code to the runtime behavior of the ECU, all MPs are "inactive" by default. In this state, RTM does require very little CPU-load but also does not record measurement data. An exception is auto start measurement which is active by default, see Chapter: 2.1.5 for details.

Performing measurement with RTM means activating one or more MPs. The MPs can be activated within CANoe with the help of RTM's frontend or a self-written Rtm application. Once activated, data is collected if the corresponding code section is executed.

To reduce RTM's code and data size as well as runtime impact, MPs store only the raw values of a measurement. Statistical analysis of the raw data is performed by RTM's CANoe frontend. Analysis includes:

- Absolute maximum runtime
- Absolute minimum runtime
- Average runtime
- Average CPU-load caused by the result of the specific MP

RTM allows individual activation of available MPs at runtime. The MPs can be selected for measurement in the RTM frontend in CANoe. Three different measurement variants are supported:

- Parallel Measurement
- Serial Measurement
- Live Measurement

2.1.1 Measurement modes

2.1.1.1 Parallel Measurement

With parallel measurement, all MPs enabled at pre-compile time and activated in CANoe are measured simultaneously. The measurement duration is specified at measurement start and the results are recorded for the provided duration.

After measurement, a HTML report containing the results is generated.

Measurement results are visualized by CANoe system variables:

- Rtm_Result_X_time
- Rtm_Result_X_count

> Rtm_Result_X_min
> Rtm_Result_X_max

'X' has to be replaced with the internal measurement Id. These variables are updated after measurement has finished.

2.1.1.2 Serial Measurement

With serial measurement, all MPs enabled at pre-compile time and activated in CANoe, are measured consecutively. Serial measurement can be used when the runtime impact on the ECU shall be as small as possible. The measurement duration is specified at measurement start and applies for each MP individually, leading to a longer measurement duration than in parallel measurement.

After measurement, a HTML report containing the results is generated.

Measurement results are visualized by CANoe system variables:

> Rtm_Result_X_time
> Rtm_Result_X_count
> Rtm_Result_X_min
> Rtm_Result_X_max

'X' has to be replaced with the internal measurement Id. These variables are updated after measurement has finished.

Note
Since all measurement sections are measured for the specified duration consecutively, the overall measurement duration is [# selected MPs] x Measurement duration.

To avoid excessively long measurement durations, variant Parallel can be used.

2.1.1.3 Live Measurement

In contrast to Parallel and Serial Measurement, Live Measurement sends data on a cyclic basis. There is no pre-defined measurement duration. Instead the measurement has to be stopped by the user.

Live Measurement is especially interesting for the observation of dynamic changes like the impact of certain ECU states to the CPU-loads and runtimes.

Measurement results are visualized by CANoe system variables:

> <MP Name>_cpuload
> <MP Name>_runtime

RTM updates these variables continuously during measurement.

Note
In live measurement mode, no report is generated. The results have to be visualized in CANoe, e.g. in State Tracker or in the Graphics window.

2.1.2 RTM CANoe Control

To use the RTM via CANoe there are two ways:

1. Use the provided Rtm Test Module for GUI based measurement control (the delivered RTM frontend).
2. Use the provided services of the RTM CAPL file (RtmCan.cin) to control the runtime and CPU load measurements more individually.

The two ways are shown in the following figure.

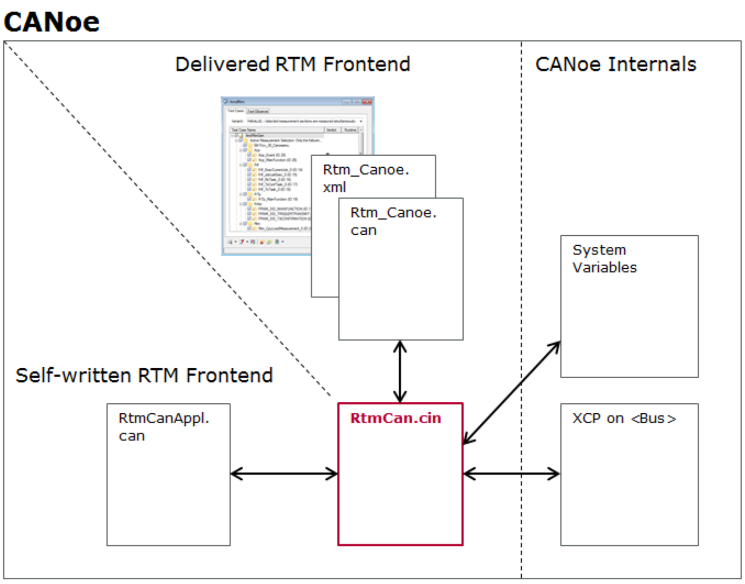


Figure 2-1 RTM control within CANoe

The delivered RTM frontend provides static control of the runtime and overall CPU load measurement. The active MPs, the measurement variant and measurement duration are selectable. A html test report can be generated after measurement end. More details follow in chapter 2.1.3.

With the self-written RTM application the same MPs, the same measurement variants and the same measurement durations are selectable. But additionally it is possible to start or stop the measurement on a specific action within the CANoe simulation, like an occuring event.

```
…
on timer MyTimer {
  RtmCan_StartMeasurement(RTMCAN_MODE_PARALLEL, 1000 /* Measurement Duration =
1s */);
}
…
```

Another advantage is, the measurement results can be requested at runtime.

```
…
  RtmCan_GetMPResultByName(„MyMeasurementPoint“, result /* Reference to result
structure */);
  If (result.RtmCan_Result_NumberOfExecution > 0) {
    /* This MP was executed. */
  }
  else {
    /* This MP was not executed. */
  }
…
```

A short overview of RtmCan.cin is given in chapter 3.7.3. The detailed description follows in chapter 7.

2.1.3 CANoe Frontend

The CANoe frontend is an easy to use user interface that controls the runtime measurement on the ECU. The frontend displays all MPs that have been pre-compile time enabled in DaVinci Configurator. MPs that are assigned to user defined groups in DaVinci Configurator (Parameter: `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementGroup`) are sorted by these groups in the CANoe interface.

MPs can be selected and deselected individually, or group wise for measurement.

The measurement variant (serial, parallel, live) can be selected by a drop-down box. The measurement duration is requested in form of a user dialog after measurement start.

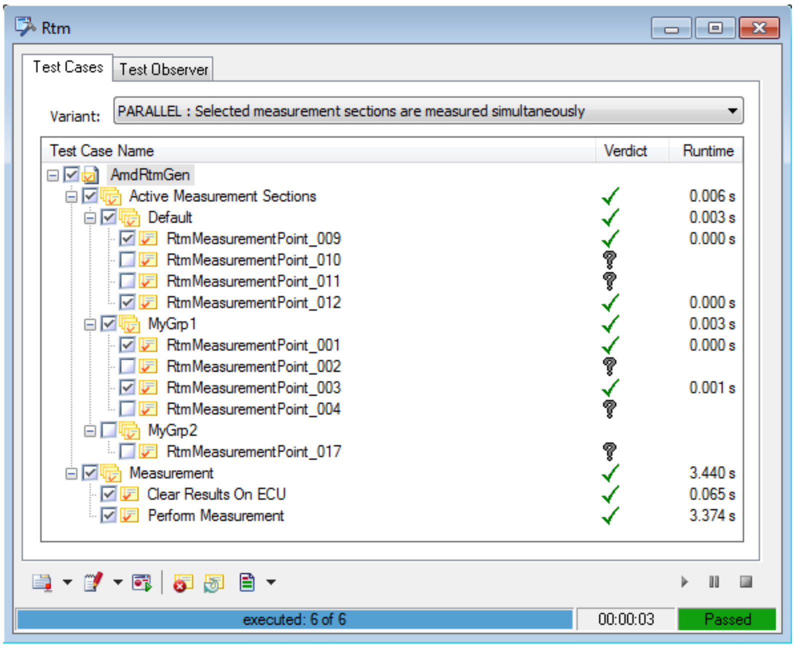


Figure 2-2 RTM CANoe Frontend

2.1.4 Report Generation

For serial and parallel measurement variants from chapter 2.1.1.1 and 2.1.1.2 a report is generated. The test report contains the following information for each MP:

- Counter value of measurement section execution
- Absolute minimum runtime [µs]
- Absolute maximum runtime [µs]
- Average runtime [µs]
- Average CPU load caused by this measurement section [%]
- Assigned core ID

The assigned core ID is only relevant in multi core systems and is set “-“ if no core is specified. Please refer to chapter 2.1.12.2.

Furthermore the frontend stores the results of a measurement session in a CSV file (report.csv) for analysis purposes.

2.1.4.1 Report for Timing-Architects™

A second report is generated, containing only MPs with measurement group “Runnable”.

The report is called “runnableReport.csv” and contains the measurement group, the name, the min-, max- and average runtimes of the runnable MPs.

Note
RTM does not check MPs with measurement group “Runnable” to measure runnable runtime.
This has to be verified by the RTM user.

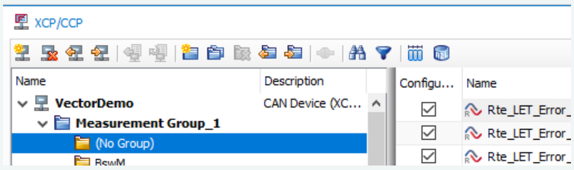
To measure runtime of runnables it is possible to use the feature VFB tracing (`/MICROSAR/Rte/RteGeneration/RteVfbTraceEnabled` and `RteVfbTraceFunction`) of the RTE. The RTE generates hook functions for all selected runnables.

Within these hook functions, the RTM macros (Rtm_Start(…) and Rtm_Stop(…)) can be called.

2.1.4.2 LET report

A third report is generated, containing only LET MPs (starting with “Rte_LET_”). Those MPs are automatically generated by the RTE if the feature LET is enabled (please refer to [4] for more information about the RTE).

Caution
If LET feature is used, in the XCP/CCP window of CANoe, the Rte_LET_Error_... variables must be set to configured. They can be found in “(No Group)”:



The generated report is called “Report_LET.csv” and contains the MP name, the maximum runtime, the LET interval (equal to configured ./RtmTargetRuntime), the relative remaining time, and the LET error.

The LET errors are described in the following Table 2-1.

LET error	Description
`RTMCAN_LET_ERROR_OK`	No error occurred.
`RTMCAN_LET_ERROR_INTERVAL_EXCEEDED`	LET interval violated: Too long execution time.
`RTMCAN_LET_ERROR_UNEXPECTED_RELEASE`	RTE’s LET state-machine violated: Unexpected release of LET.

RTMCAN_LET_ERROR_UNEXPECTED_TERMINATE	RTE's LET state-machine violated: Unexpected termination of LET.
RTMCAN_LET_ERROR_UNEXPECTED_RUNNABLE_EXECUTION	RTE's LET state-machine violated: Unexpected runnable started in LET interval.

Table 2-1 LET errors

If the API Rtm_TriggerReading() is called (usually by RTE), the measurement is stopped and intermediate reports are generated (before the test panel is stopped):

> report_<DateAndTime>.csv

> runnableReport_<DateAndTime>.csv

> Report_LET_<DateAndTime>.csv

The <DateAndTime> is usually formatted as follows:

> <NameOfDay>_<Month>_<Day>_<CurrentTime>_<Year>

  > Example: Fri_Nov_25_091458_2022

2.1.5 Auto start Measurement Points

MPs can be configured to start measurement right at ECU start up before the XCP connection to CANoe is established. Auto start MPs perform measurement until any other measurement is started by RTM's frontend. To enable auto start for a MP, the following DaVinci Configurator parameter has to be set to true: `/MICROSAR/Rtm/RtmMeasurementPoint/RtmAutostartEnabled`

Auto start measurement can be used i.e. for init-functions which are called only once while ECUs start up.

The result of the auto start measurement is written to the according system variable as soon as a regular measurement is started. Therefore, the MP of the auto start measurement has to be activated. The CANoe parameter **Clear ResultsOn ECU** has to be cleared, else the result of auto start measurement is overwritten.

Caution
Like all MPs during ongoing measurement, auto start points generate additional CPU load. This influences the ECU's runtime behavior. Hence this feature should only be used rarely and/or for code sections which are executed seldom.

Caution
The runtime measurement only delivers correct results if the used timer is already started.

It is possible that the used timer in ‘RTM_GET_TIME_MEASUREMENT_FCT()’ reports an error if accessed before it is started. Therefore, only access the timer if it is already started. If started, return the timer’s value. Otherwise, return 0. This leads potentially to incorrect measurement results.

For task response time measurement this is considered as acceptable, because the longer the measurement runs, the smaller is the effect of this one incorrect measurement result.

For other MPs this must be evaluated individually. If one incorrect measurement result is not acceptable, it may be better to start the MP manually short time after ECU start up by calling Rtm_PrepareMPSettings() and Rtm_StartMeasurement() (of course this is not possible for initialization functions).

2.1.6 Threshold Callbacks

Each MP can be configured to have a threshold callback, selectable via DaVinci Configurator parameter:

`/MICROSAR/Rtm/RtmMeasurementPoint/RtmThresholdCallback`

For this, the additional parameter must be set:

`/MICROSAR/Rtm/RtmMeasurementPoint/RtmTargetRuntime`

Each time the measured runtime exceeds the specified threshold, a user implemented callback function is called. Within this function user code can be implemented which reacts to the runtime violation.

The threshold callback follows the name scheme:

> void Rtm_<MP_Name>_ThresholdCbk(Rtm_MeasurementTimestampType executionTime)

Note
If the LET feature (refer to 2.1.22) is enabled by RTE, the thresholds of LET specific MPs (starting with ”Rte_LET_”) are implemented by RTE. Therefore, those callbacks cannot be implemented by application.

2.1.7 Calibration

RTM automatically corrects measurement results by the overhead introduced during measurement.

2.1.8 Measurement Types

There are three different types of measurements:

1. GrossExecutionTime
2. NetExecutionTime
3. FlatExecutionTime

These types can be set for each MP individually (`/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType`); they define the measurement behavior of each MP.

The meaning of these types is described by the following examples:

2.1.8.1 GrossExecutionTime

MPs measuring the gross execution time, measure the absolute time from measurement start to stop. These MPs do not consider interruptions in their measurements.

These MPs can be started on one core and stopped on another core. The same applies to tasks and ISRs.

The following is an example of measurement type GrossExecutionTime:

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	GrossExecutionTime	0	0
MP1	GrossExecutionTime	0	0
MP2	GrossExecutionTime	0	1
MP3	GrossExecutionTime	0	1
MP4	GrossExecutionTime	1	2
MP5	GrossExecutionTime	1	2

Table 2-2 Measurement Config – all MPs GrossExecutionTime

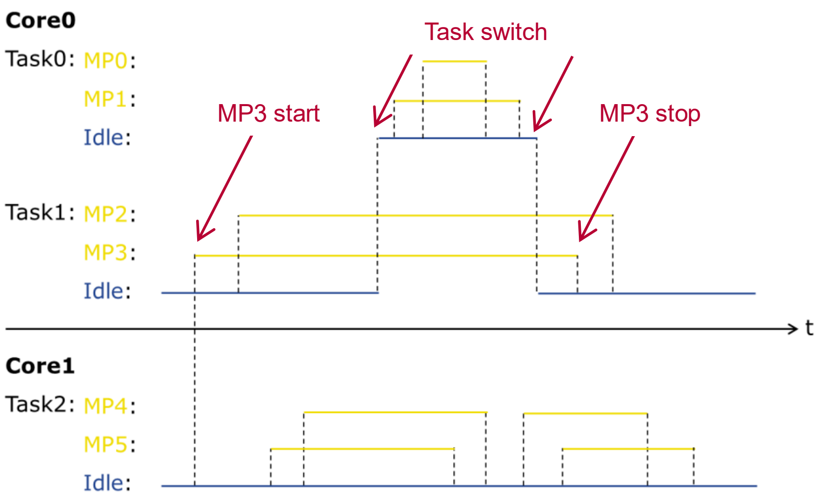


Figure 2-3 All measurement points of type GrossExecutionTime

Each measurement point measures the time from start until stop, this is the gross execution time. Measurement point `MP3` is started on `Core1` and stopped on `Core0`. This is only possible for measurement points set to GrossExecutionTime. All other measurement points are started and stopped on the same core.

It is also possible to start a measurement point on one task and stop it on another, where both tasks run on the same core.

2.1.8.2 Net- and FlatExecutionTime

MPs measuring net or flat execution time remove the execution times of interrupting tasks and ISRs from their measurement results.

To enable these measurements, RTM provides two mechanisms OS timing hooks and OS Pre-/Post Task and ISRs Hooks for RTM module. For more details, please refer to 3.4.1 and 3.4.2 respectively. (The recommended mechanism is OS timing hooks).

For more details about OS configuration for RTM module, please refer to 3.4.1.1.

2.1.8.2.1 NetExecutionTime

MPs of type NetExecutionTime consider the execution time of interrupting tasks and ISRs. They do not consider the execution time of nested MPs.

It follows that a MP started on core 0 must also be stopped on core 0. The same applies to tasks and ISRs.

Note
The following example explains the potential behavior of NetExecutionTime MPs in a multicore system.

The following is an example of measurement type NetExecutionTime:

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	NetExecutionTime	0	0
MP1	NetExecutionTime	0	0
MP2	NetExecutionTime	0	1
MP3	NetExecutionTime	0	1
MP4	NetExecutionTime	1	2
MP5	NetExecutionTime	1	2

Table 2-3 Measurement Config – all MPs NetExecutionTime

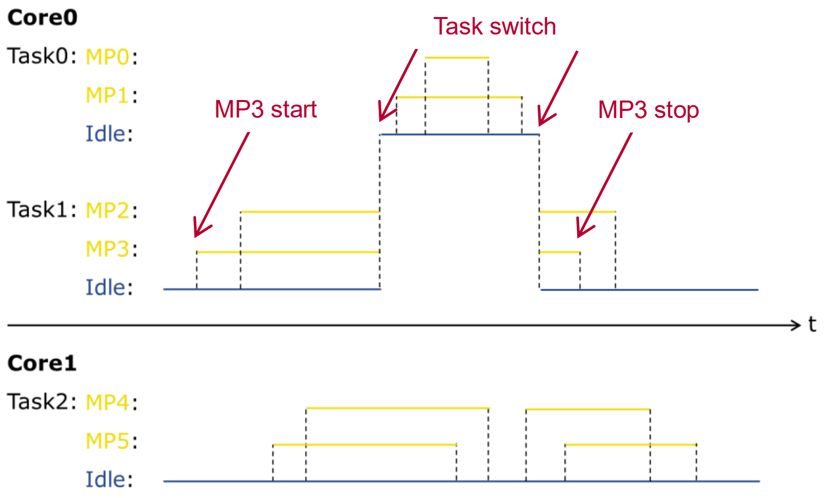


Figure 2-4 All MPs of type NetExecutionTime

Each MP can be interrupted by task switches and interrupts. They only measure the runtime of their own task.

The measurement result of `MP2` and `MP3` is from start until stop minus the execution time of `Task0`.

2.1.8.2.2 FlatExecutionTime

MPs of type FlatExecutionTime consider the execution time of nested MPs additionally to the execution time of interrupting tasks and ISRs.

It follows that a MP started on core 0 must also be stopped on core 0. The same applies to tasks and ISRs. Additionally, all nested FET MPs must be started and stopped in correct order. This means the last started MP must be the first stopped.

Caution
The execution time of a nested MP is only subtracted from the outer MP, if the nested MP is of type FlatExecutionTime.

Note
The following example explains the potential behavior of FlatExecutionTime MPs in a multicore system.

The following is an example of measurement type FlatExecutionTime:

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	FlatExecutionTime	0	0
MP1	FlatExecutionTime	0	0
MP2	FlatExecutionTime	0	1
MP3	FlatExecutionTime	0	1
MP4	FlatExecutionTime	1	2
MP5	FlatExecutionTime	1	2

Table 2-4 Measurement Config – all MPs ExecutionTime_Nested

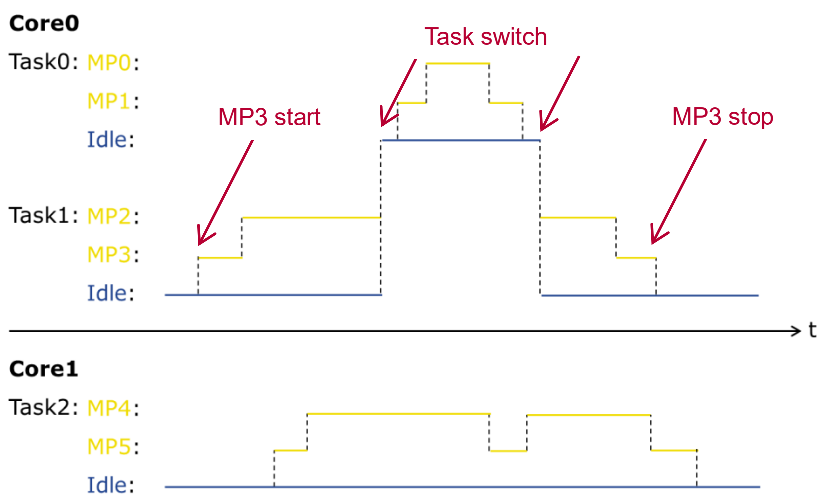


Figure 2-5 All MP s of type FlatExecutionTime

Each MP can be interrupted by task switches, interrupts and other MPs on the same core.

In this example, `MP3` is interrupted by `MP2` and `MP2` is interrupted by `Task0`. After switch back to `Task1` `MP2` is executed again. After finish of `MP2`, `MP3` is executed again.

The runtime of `MP3` is calculated from start to stop minus execution time of `Task0` and minus execution time of `MP2`.

2.1.8.3 Mixed measurement point types

Note
The following example explains the potential behavior of flat and net execution time MPs in a multicore system.

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	FlatExecutionTime	0	0
MP1	NetExecutionTime	0	0
MP2	GrossExecutionTime	0	1
MP3	FlatExecutionTime	0	1
MP4	FlatExecutionTime	1	2
MP5	FlatExecutionTime	1	2

Table 2-5 Measurement Config – mixed MP types

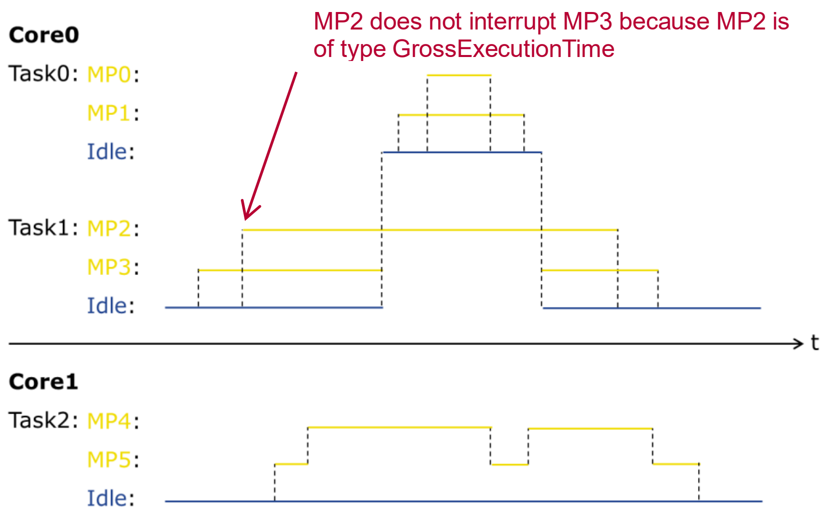


Figure 2-6 Mixed measurement types

In this example `MP4` and `MP5` are set to `FlatExecutionTime`. Therefore, the execution of `MP5` is interrupted by `MP4`.

Even though `MP3` is set `FlatExecutionTime` only the execution time of `Task0` is subtracted, but not the execution time of `MP2`. This is because the type of `MP2` is set to `GrossExecutionTime`. MPs of type `GrossExecutionTime` do not affect the runtime of other MPs. In this example the runtime of `MP2` is greater than the runtime of `MP3`, thus the resulting runtime of `MP3` would be negative.

The runtime of `MP0` is not subtracted from runtime of `MP1` because `MP1` is of type `NetExecutionTime`.

2.1.8.4 Functionality of nested MPs

If the MP's type measuring `Func_1` is set to `FlatExecutionTime` and the called function `Func_2` is set to `FlatExecutionTime`, the execution time of `Func_2` is subtracted from the execution time of `Func_1`. This is shown in Figure 2-7 on the right.

If the called function is not measured by a separate MP, or one of both MPs is of type `NetExecutionTime` or `GrossExecutionTime`, the execution time of `Func_1` does not consider the execution time of the inner function `SubFunc_1`. This is shown in Figure 2-7 on the left.

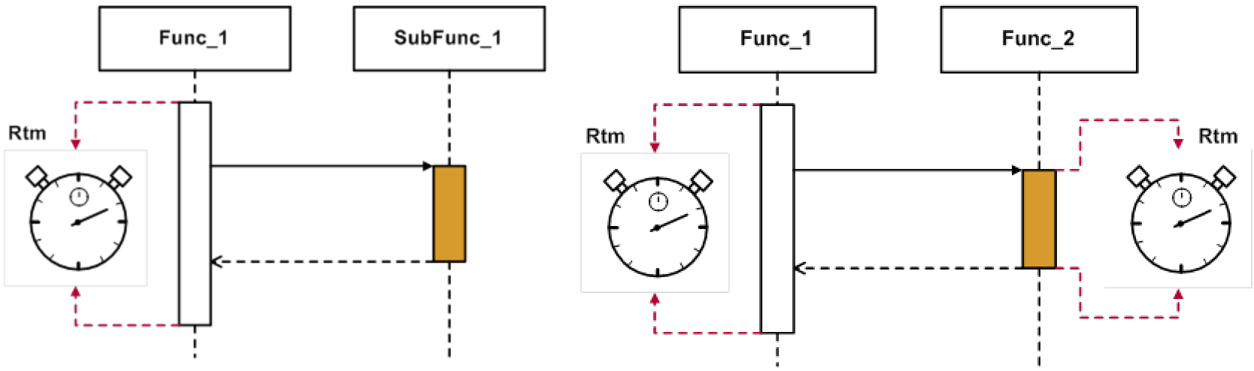


Figure 2-7 On the left: complete runtime of Func_1 is measured, on the right: the runtime of Func_2 is subtracted from Func_1

Caution
Because serial measurement does only measure one MP after each other, there is no information about the runtime of the nested measurement. Hence, the net runtime can only be measured with parallel or live measurement.

2.1.9 MP Type

There are 3 MP types:

1. Runtime
2. Task
3. CPU_Load

The type can be set for each MP individually (/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementPointType); it defines the MP behavior.

2.1.9.1 Runtime

This is a regular MP using one of the measurements described in 2.1.8.

2.1.9.2 Task

This MP type is used to measure the task response time and can be used in combination with the task response time percentile histogram. The start and stop of such a MP are generated in the OS timing hooks.

For more details about OS timing hooks and OS configuration for RTM module, please refer to 3.4.1 and 3.4.1.1 respectively.

For more details about task response time histogram, please refer to 2.1.17.2.

> **Caution**
> The MP type (`/RtmMeasurementPointType`) 'Task' is reserved for the task response time histogram feature.
> Do not set manually created MPs to this type!
> Instead, always use 'Runtime' as type.

2.1.9.3 CPU_Load

This MP type is used to measure the CPU load. It can be used in combination with the CPU load percentile histogram.

The start and stop of such a CPU load MP is generated in the OS timing hook OS_VTH_SCHEDULE (refer to 4.2.24).

For more details about OS timing hooks and OS configuration for RTM module, please refer to 3.4.1 and 3.4.1.1 respectively.

For more details about CPU load histogram, please refer to 2.1.17.1.

> **Caution**
> The MP type (`/RtmMeasurementPointType`) 'CPU_Load' is reserved for the automatically created CPU load MPs (Rtm_CpuLoadMeasurement<_CoreId>).
> Do not set manually created MPs to this type!
> Instead, always use 'Runtime' as type.

2.1.9.3.1 CPU Load Measurement

RTM allows to measure the current overall CPU load of all cores.

There are two ways to control the CPU load measurement. The CPU load control mode can be selected in pre-compile time with the DaVinci Configurator parameter:

- `/MICROSAR/Rtm/RtmCpuLoadMeasurement/RtmCpuLoadControlMode`

C_API	The CPU load measurement requires a special MP for each core with short name `Rtm_CpuLoadMeasurement<_CoreId>`. This MP has the same configuration parameter as any other MP. The measurement is started and stopped by API that can be called within the BSW and SWCs (`Rtm_Start_CpuLoadMeasurement<_CoreId>` and `Rtm_Stop_CpuLoadMeasurement<_CoreId>`). Those APIs are just a more comfortable way to en-/disable the CPU load MPs compared to the generic APIs described in 2.1.20.2. This means, one call of `Rtm_Start_CpuLoadMeasurement(_CoreId)` is comparable to the sequence: > `Rtm_PrepareMPSettings(RTM_MP_SETTING_ENABLE_ONE_MP, RtmConf_RtmMeasurementPoint_Rtm_CpuLoadMeasurement);` > `Rtm_StartMeasurement(0u);` The result can be requested via `Rtm_GetMeasurementItem()`.
Xcp	The CPU load measurement requires a special MP with the short name `Rtm_CpuLoadMeasurement<_CoreId>`. This MP has the same configuration parameter as any other MP. The measurement is enabled and disabled within the Rtm_MainFunction. The result can be requested within the ECU via `Rtm_GetMeasurementItem` and is automatically send to CANoe via XCP.

Table 2-6 CPU load control modes

The CPU load measurement is available for any measurement mode (Serial, parallel and live) if `Xcp` is chosen for `RtmCpuLoadControlMode`.

Note
The MP `Rtm_CpuLoadMeasurement(_CoreId)` cannot be used as usual MP. It is reserved for the CPU load measurement, independent of the used control mode.

Note
The CPU Load measurement can result in a fault measurement when the nesting counter [2.1.11] and DET checks (`RTM_DEV_ERROR_DETECT`) are disabled.

2.1.10 Target Runtime

The parameter /MICROSAR/Rtm/RtmMeasurementPoint/RtmTargetRuntime has three use cases:

1. It is used to define the 100% runtime for Task MPs. The 100% defines the center of the percentile histogram. The percentiles range from 0% to 200%. The number of percentiles can be configured at

/MICROSAR/Rtm/RtmApplicationCoreDefinition/RtmNumberOfTaskResponseTimePercentiles.

2. It is used to calculate RTM_ITEM_RELATIVE_MAX of Rtm_ItemType and therefore it is relevant for Rtm_GetMeasurementItem.

3. It is used for Threshold Callbacks to define the threshold when the callback is actually called.

Note
If the target run time set is more than the 16-bit timer, it will either suggest the preferred value within limit or set the timer to 32-bit in configurator.

2.1.11 Nested counter

The parameter `RtmUseNestedCounter` avoids starting an already started MP. This parameter can be cleared if it is verified that all MPs are stopped before the next start occurs. Otherwise, measurement results can be corrupted.

2.1.12 Measurement on multi core system

It is possible to measure runtime and overall CPU load on several cores simultaneously. The feature can be activated by adding one `RtmCoreDefinition` per core.

To use this feature there are some pre-conditions:

- The used micro controller has to have a 32bit CPU
- Used timer for RTM has to have a 32bit base
- RTM requires the use of same timer source for all cores to make measurement results comparable and cross-core measurements possible, therefore the timer has to fulfill one of the following requirements:
  - The timer request must be reentrant OR
  - The access to the timer value must be atomic OR
  - OS Spinlocks have to be used
    - All three exclusive areas have to implement Spinlocks and exclusive areas!
    - Note that this causes heavy runtime!
- The RTE and the OS (SC1 or SC2) have to support multi core

Caution
To support OS SC 3 and SC 4 map the following memory sections to a memory area where all cores have permission for write access (global shared memory):

- RTM_START_SEC_VAR_INIT_UNSPECIFIED
- RTM_START_SEC_VAR_NOINIT_UNSPECIFIED
- RTM_START_SEC_VAR_ZERO_INIT_8BIT

2.1.12.1 Core Definition

The `RtmCoreDefinition` considers cores from either EcuC or OS. For each core defined, one `RtmCoreDefinition` shall be provided.

From OS:

`RtmCoreDefinition` considers the AUTOSAR cores defined in OS.

The following parameter specifies whether the OS-Core defined is an AUTOSAR core.

`/MICROSAR/Os/OsCore/OsCoreIsAutosar`

From EcuC:

`RtmCoreDefinition` considers the cores defined in EcuC.

`/MICROSAR/EcuC/EcucHardware/EcucCoreDefinition`

Note
For the older versions of OS, `RtmCoreDefinition` considers EcuC. It is recommended to verify the existence of container `/MICROSAR/Os/OsCore` to determine the latest version of OS.

Note
The requirement is to enable measurements for all cores that have been defined in OS/EcuC. For example, there are four cores defined in OS, it is not possible to enable measurement for only one or some cores.

2.1.12.2 Assignment of MPs to a core

Each MP can be assigned to one core by setting the reference `RtmAssignedToCore` to a `RtmCoreDefinition` with set `./RtmCore`.

If the MP references a `RtmCoreDefinition` without `./RtmCore`, the MP is not assigned to any core. The MP can then be started on any core and stopped on any core. It is allowed to start and stop the MP on the same core. But it is recommended to always use the same core for starting and always the same core for stopping a specific MP.

If the same MP can be started and stopped on several cores simultaneously it is hard to interpret the measurement results.

Note
It is recommended to activate the parameter `RtmUseNestedCounter` if at least one MP is not assigned to any core. It has to be verified that a MP is never started and stopped simultaneously, e.g. by starting/stopping the MP from different cores. In the multi core case the RTM variables have to be mapped to a shared memory area.

If the MP references a `RtmCoreDefinition` with `./RtmCore`, this MP has to be executed only on the specified core. It must not be executed on any other core. This check is not executed by RTM; therefore, the RTM user has to verify it.

MPs where this parameter does not exist can be used to measure cross core.

2.1.12.3 Measurement examples

The following figure shows part of measurement sequence of six example MPs which are measured in parallel mode.

The first two MPs (`Mp01_C0` and `Mp02_C0`) are exclusively executed on core 0. The following two MPs (`Mp03_C1` and `Mp04_C1`) are exclusively executed on core 1. The last two MPs (`Mp05_AllC` and `Mp06_AllC`) are cross core MPs and may be executed on any core. `Mp05_AllC` is always started and stopped on core 0 and `Mp06_AllC` is always started on core 0 and always stopped on core 1.

Figure 2-8 MP execution on multi core systems

Now the RTM support flat and net execution time measurement on multi cores.

2.1.13 Safe RTM

Note
This chapter is only relevant for the use of ASIL software.

The RTM provides two variants to be used within ASIL software:

1. Safe disable
2. Functional safety

Only one variant shall be used at a time.

2.1.13.1 Safe disable

The RTM provides the possiblity to disable the RTM's functionality completely.

To disable the RTM the following steps are required:

1. Set the parameter `RtmControl` to enabled.
2. Disable the RTM at runtime by setting the variable `Rtm_ControlState` to `RTM_CONTROL_STATE_DISABLED` (= 0). This lock must be done after the call to Rtm_InitMemory but before the call to Rtm_Init.

The variable `Rtm_ControlState` must not be written by RTM or other software with lower ASIL level than the highest available. Therefore make the following memory section read-only for RTM and other low safety level software:

> `RTM_START_SEC_VAR_INIT_UNSPECIFIED_SAFE`

> `RTM_STOP_SEC_VAR_INIT_UNSPECIFIED_SAFE`

Please refer to [3] for details about MemMap.

Caution
The RTM may only be used in an operating mode that does not impose risk for health of persons.

Caution
To support OS SC 3 and SC 4 map the following memory sections to a memory area where all cores have permission for write access (global shared memory):

> RTM_START_SEC_VAR_INIT_UNSPECIFIED

> RTM_START_SEC_VAR_NO_INIT_UNSPECIFIED

> RTM_START_SEC_VAR_ZERO_INIT_8

2.1.13.2 Functional safety

The RTM provides the possibility to use most of its functionality in ASIL software.

Unsupported features are Net- and FlatExecutionTime MPs. Therefore, set the measurement type (`/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType`) of all MP to GrossExecutionTime.

2.1.14 Runtime Measurement of Runnables

A common task is to measure the runtime of runnables in the system. The recommended way to do this is to configure a MP for each runnable. These MPs can be started and stopped by using VFB trace hook function generated by the RTE.

The configuration of MPs, VFB trace functions and their implementation is assisted through the ‘Runtime Measurement’ comfort editor and a script provided by RTM. The necessary configuration steps are explained in the following.

1. Generate the RTE.
2. In the ‘Runtime Measurement’ comfort editor open ‘Measurement Points’ and click on ‘Runnables and MainFunctions’. Click on the plus symbol to open the 'Import Assistant' and select the generated file Rte_Hook.h.
3. Select all Start/Return pairs for those runnable (or schedulables) that shall be measured.
4. Switch to the ‘Script Tasks’ window, expand the script ‘CreateRunnableMeasurementPoints’ and right click on the script project. Select 'Execute'.

The script generates the source file Rte_Hook_Rtm.c in the folder .\Appl\Source. It contains the configured runnable hooks that call the Rtm macros `Rtm_Start` or `Rtm_Stop`.

Note
The script only works for standard paths, i.e. .\Appl\GenData and .\Appl\Source. If the project has different paths configured the script CreateRunnableMeasurementPoints.dvgroovy must be adapted.

2.1.15 Measurement Result Overflow

The measurement result of any MP, including the overall CPU load MP, is cached in a corresponding 32bit variable. Therefore, the result may overflow after a specific measurement duration (how to calculate this duration is described in the section 2.1.16).

A handling of measurement result overflows is only recommended if time-limited execution time/CPU load measurements are planned to be executed. The reason for this is the reset of measurement results to the latest measured values whereas the measurement duration is not reset. Therefore, the relation between result and measurement duration gets wrong after the first reset.

If endless measurements are planned or only gross execution time is measured, the overflow handling is not required.

The Rtm provides the possibility to report the DET error `RTM_E_MEASUREMENT_POINT_RESULT_OVERFLOWN` if any measurement result is overflown. This DET error is only triggered if DET error reporting is enabled and the `RTM_REPORT_DET_FOR_MEASUREMENT_POINT_RESULT_OVERFLOW` is defined as `STD_ON` within a user config file (`/Rtm/RtmGeneral/RtmUserConfigFile`). Therefore, this DET is initially deactivated as it is not an actual error but just a notification for the user.

Caution
If the DET is active and the measurement result of a MP is overflown, the results cannot be used to calculate the MP specific CPU load anymore, because the execution time was reset during measurement!

The reported DET error just indicates that one MP was corrupted, it cannot tell which MP was corrupted. Therefore, repeat the measurement and set a breakpoint where the error code `RTM_E_MEASUREMENT_POINT_RESULT_OVERFLOWN` is set in the file Rtm.c.

If `RTM_REPORT_DET_FOR_MEASUREMENT_POINT_RESULT_OVERFLOW` is defined as `STD_ON` and the DET error `RTM_E_MEASUREMENT_POINT_RESULT_OVERFLOWN` is reported, the measurement duration is too long. To find out how long the measurement may be without any MP to overflow, please refer to the following section.

2.1.16 Maximum Measurement Duration

The MPs of RTM are limited in their maximum measurement duration. The RTM accumulates the execution time of each enabled MP. The time value [in ticks] is cached in a uint32 variable. Therefore, its maximum value is 0xFFFFFFFF.

The maximum measurement duration (`MaxMeasDuration`) of a MP depends on the following values:

1. The counter frequency (`Frequency [Hz]`)
2. The maximum value of the accumulated time (`MaxValue`)
3. Ratio between execution time of a MP and the measurement duration (`RatioExecTimeToMeasDuration`)

The formula:

➔ MaxMeasDuration = 1/RatioExecTimeToMeasDuration * MaxValue * 1/Frequency

Example:

1. `Frequency [Hz]` = 10MHz
2. `MaxValue` = 4294967295
3. `RatioExecTimeToMeasDuration` = 40/100 = 2/5

➔ **MaxMeasDuration** =
1/(2/5) * 4294967295 * 1/10,000,000[Hz]
= 5/2 * 4294967295 * 0.0000001s
= **859s** (= 14,3min)

Note
The percental execution time (`PercentalExecTime`) of a MP can be measured with a short measurement duration (e.g. 1s). Execute the measurement, open the report and calculate:

➔ `RatioExecTimeToMeasDuration` = $\mathtt{MP}_{\mathtt{AverageTime}}$ * $\mathtt{MP}_{\mathtt{Count}}$ / `MeasurementDuration`

Example: The MP was measured 200 times with an average execution time of 1000µs. The measurement duration was 1s:

➔ 1,000µs * 200 / 1,000,000µs = 200,000µs / 1,000,000µs = **20/100 = 1/5**

2.1.17 Histograms

The Rtm provides the possibility to gather its measurement results as histogram. Each histogram consists of a configurable number of percentiles. Those percentiles give a percental distribution of the results. E.g. how often the CPU load was in range of 40-49% compared to all other percentiles. Therefore, the sum over all percentiles is always 100(%).

Note
Due to not avoidable rounding errors, the sum over all percentiles is mostly only near 100(%). Nevertheless, the relation between those values is correct.

The following results are available as histogram:

> CPU load

> Task response time

The available histogram gathers the current measurement results, calculates the corresponding percentile, and increments this percentile.

For the calculation, the current measurement result is taken into relation to a target value. The target value depends on the histogram type.

2.1.17.1 CPU Load Histogram

For the CPU load histogram feature a hyperperiod is defined which elapses after one or multiple calls of the Rtm_MainFunction. The hyperperiod's runtime is the basis for calculation of the percentile (the target value).

Example:

If the Rtm_MainFunction cycle time is 10ms and the hyperperiod is set to 100ms, each 100ms a new result for the histogram is available.

CPU load results for the 10 Rtm_MainFunction cycles:

> 40%, 45%, 62%, 50%, 47%, 80%, 12%, 51%, 49%, 50%

Calculation of active runtime:

> (40+45+62+50+47+80+12+51+49+50) / 10 = **48%**

With 10 (11 if the special percentile 100% is considered) percentiles, there are the following percentiles:

> 0-9%, 10-19%, 20-29%, 30-39%, 40-49%, 50-59%, 60-69%, 70-79%, 80-89%, 90-99%, 100%

Therefore, the percentile **40-49%** is incremented.

With the service Rtm_GetCpuLoadHistogram the results can be requested.

Example:

In the following diagram is shown what the service Rtm_GetCpuLoadHistogram returns after the first, second, 10th, and the 100th measurement result is available.

After the first measurement (elapse of first hyperperiod) only one result is available, therefore this percentile indicates that 100% of all measurement results were measured in this percentile. The percentile in this example is 40-49%.

The second percentile in this example is 60-69%, therefore after the second measurement both percentiles 40-49% and 60-69% indicate 50%.

After 10 and 100 available measurement results, the curve gets sharper. It is most likely that the curve does not change anymore after a specific number of measurement results are available.

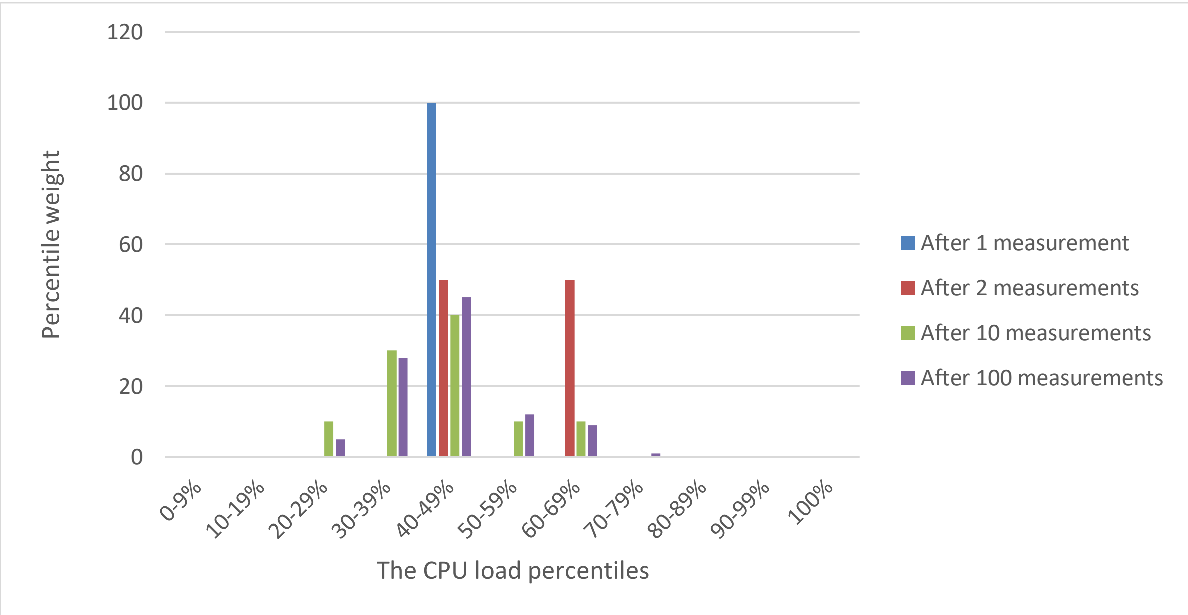


Figure 2-9 Example of CPU load histogram results after different number of available measurement results

For configuration details, please refer to 3.5.3.1.

2.1.17.2 Task Response Time Histogram

For the task response time histogram each task MP has a target runtime. With this target runtime the percentile is calculated.

Example:

> The target runtime is 10µs

> The timer frequency is 10,000 ticks/ms = 10 ticks/µs

With 10 (11 if the special percentile 100% is considered) percentiles, there are the following percentiles:

> 0-19%, 20-39%, 40-59%, 60-79%, 80-99%, 100-119%, 120-139%, 140-159%, 160-179%, 180-199%, >=200%

If the task runtime 5µs is measured, the percentile 40-59% is incremented.

With the service Rtm_GetTaskResponseTimeHistogram the results can be requested.

Example:

In the following diagram is shown what the service Rtm_GetCpuLoadHistogram returns after the first, second, 10th, and the 100th measurement result is available.

After the first measurement the percentile 160-179% indicates that 100% of all available task response time results are in this percentile.

The second measurement result is between 40-59%, therefore both percentiles now indicate the weight of 50%.

After 10 measurements, the results more distribute over a set of percentiles.

After 100 measurements, it becomes clear that the lowest percentile 40-59% and the highest percentile 180-199% are just exceptions. It is possible that the significance of these percentiles become 0 within the next measurement cycles.

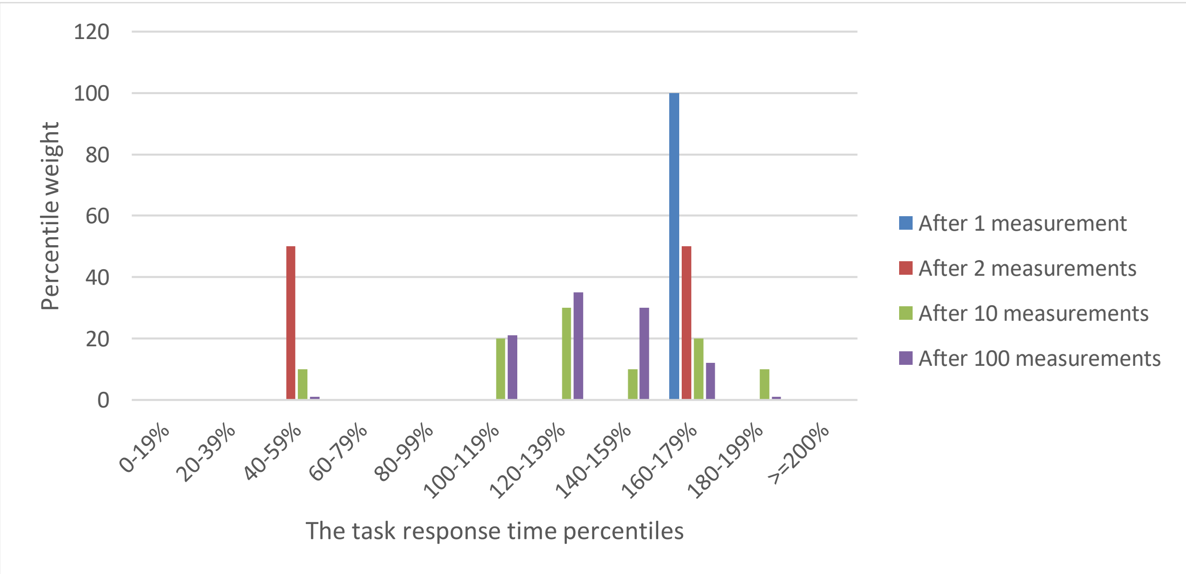


Figure 2-10 Example of task response time histogram results after different number of available measurement results

For configuration details, please refer to 3.5.4.

2.1.17.2.1 Oversampling

To be able to use the task response time histograms, the timer resolution must be at least such that the percentiles can be sampled with a resolution good enough for the purpose.

The timer resolution varies with the number of configured percentiles and the configured target runtime. If the target runtime is too small or there are too many percentiles, it is possible that some percentiles cannot be updated.

Example:

> The `/MICROSAR/Rtm/RtmMeasurementPoint/RtmTargetRuntime` is set to 1µs

> The number of percentiles `/MICROSAR/Rtm/RtmCoreDefinition/RtmNumberOfTaskResponseTimePercentiles` is set to 10

> The timer frequency `/MICROSAR/Rtm/RtmGeneral/RtmMeasurementCtrFrequency` is set to 1,000 ticks/ms

> 1,000 ticks/ms = 1 tick/µs

Table 2-7 shows the required runtime for each percentile if the task's target runtime is 1µs. Because the timer frequency is 1 tick/µs the only percentiles that can be reached are 0,0-0,19%, 100-119%, and >=200%.

Therefore, the timer frequency has to be increased to at least 10 ticks/µs (10,000 ticks/ms) or the number of percentiles has to be reduced to 2 (0-99%, 100-19%, >=200%).

Target runtime: 1µ											
Percentiles	0-19%	20-39%	40-59%	60-79%	80-99%	100-119%	120-139%	140-159%	160-179%	180-199%	>= 200%
Required runtime [µs]	0,0-0,19	0,2-0,39	0,4-0,59	0,6-0,79	0,8-0,99	1,0-1,19	1,2-1,39	1,4-1,59	1,6-1,79	1,8-1,99	>=2

Table 2-7 Example for task response time histogram: required runtime for each percentile with target runtime 1µs

2.1.17.2.2 Calculation of Maximum runtime in µs

The Maximum runtime in µsec represents the longest duration of task response MP among all possible measurements. However, it is important to note that it does not support overrun functionality, i.e. for MPs which are once started with Rtm_Start() and stopped with Rtm_Stop() after the maximum time (0xFFFFFFFF ticks) is elapsed, the runtime is set to maximum value. If there is even longer measurement run time, it can lead to implausible results, which can impact the reliability and accuracy of the measurement data.

It is therefore important to ensure that the measurements are stopped within the specified maximum time to prevent such issues.

Example:

The counter frequency [Hz] = 100MHz

Max Value in ticks = 4,294,967,295

Max run time in µSec = ticks * (1/counter frequency) = 42,949,672 µsec.

2.1.17.3 Measurement handling for histograms

When the task response time feature (`/MICROSAR/Rtm/RtmCoreDefinition/RtmNumberOfTaskResponseTimePercentiles`) is enabled, one MP for each task is generated. For all task MPs the parameter `/MICROSAR/Rtm/RtmMeasurementPoint/RtmAutostartEnabled` is enabled. This means, that the measurement of all task MPs is active when the RTM is initialized.

For the CPU load MPs the `./RtmAutostartEnabled` has to be set manually, if required.

If the measurement is started and stopped via C-API or XCP (refer to section 2.1.20), all MPs are disabled. This includes the task and CPU load MPs. This means, that the histograms do not get new values anymore until the measurement is started again.

Additionally, the API Rtm_ClearHistogramResults() resets the cached raw values of all task response time MPs, which are used by C-API or XCP. Therefore, the measurement results could be implausible in the reports.

Therefore, it is recommended to measure the histograms or to perform the common measurement via C-API or XCP. If both is required, be aware of the side-affects.

2.1.18 Task Stack Usage

The task stack usage can be request by calling the service Rtm_GetTaskStackUsage.

The result contains the maximum used task stack since the last explicit clear of the results (see Rtm_ClearHistogramResults) and the configured maximum available task stack size. With both data the relative stack usage can be calculated:

> RelativeTaskStackUsage = (MaxTaskStack * 100) / TaskStackSize

For configuration details, please refer to 3.5.5.

2.1.19 Clear histogram results

Rtm provides the possibility to clear the results of CPU load histogram, the task response time histogram, and task stack usage.

This feature is for example required if the results are persisted and a software update is executed. After software update, the persisted results are no longer relevant because the CPU load, the task runtimes, and the task stack usage may change significantly after the update.

The persisted results can be cleared by calling the service Rtm_ClearHistogramResults.

2.1.20 Measurement handling

Measurements can be handled via XCP (respectively CANoe) and/or C-API.

> **Caution**
> It is possible to use the XCP and C-API mechanisms concurrently, but this can cause the CAPL script to fail (e.g. if a running endless measurement started by CANoe/XCP is stopped by calling Rtm_StopMeasurement() the stop may be accepted, but it is not reported via XCP. A later on received stop request from XCP/CANoe is then ignored and no response is transmitted).
>
> Therefore, it is highly recommended to use only one mechanism concurrently.

2.1.20.1 XCP/CANoe

How the measurement is handled via XCP/CANoe is described in multiple chapters. For a generic overview, please refer to sections 2.1.2 and 2.1.3. For integration hints, please refer to sections 3.6 and 3.7.

2.1.20.2 C-API

The Rtm provides functions to start and stop the runtime/CPU load measurement, and to read and clear the measurement results.

> **Note**
> The handling of histograms and task stack usage is already explained in previous sections (2.1.17, 2.1.18, and 2.1.19).

2.1.20.2.1 Prepare required MPs

Before a measurement is started, it is recommended to check if all required MPs are enabled. For this the API `Rtm_PrepareMPSettings()` is available.

The following MP setting options are available:

> RTM_MP_SETTING_DISABLE_ALL
> RTM_MP_SETTING_ENABLE_ALL
> RTM_MP_SETTING_DEFAULT
> RTM_MP_SETTING_DISABLE_ONE_MP
> RTM_MP_SETTING_ENABLE_ONE_MP

For more details, please refer to 4.1.7.

To prepare all required MPs this API can be called multiple times before the measurement is started.

All types of MPs can be prepared (runtime, CPU load, task).

For the MP setting options RTM_MP_SETTING_DISABLE_ONE_MP and RTM_MP_SETTING_ENABLE_ONE_MP an explicit MP must be given for parameter ConfiguredMPId.

The MP id can be selected by the macros also used for Rtm_Start() and Rtm_Stop():

- RtmConf_RtmMeasurementPoint_<MP_Name>
  - Where MP_Name is the configured short name of the MP

2.1.20.2.2 Start the measurement

When the required MPs are prepared, call `Rtm_StartMeasurement()` to trigger the measurement start. The actual start is executed in the next call of `Rtm_MainFunction()`. There are two measurement modes available:

- Endless measurement (pass 0x00 as MeasurementDuration)
- Time-limited measurement (pass 0x0001 – 0xFFFF; defines number of main function cycles)

2.1.20.2.3 Stop the measurement

If an endless measurement is active, the measurement may be explicitly stopped by calling `Rtm_StopMeasurement()`. The actual stop is executed in the next call of `Rtm_MainFunction()`.

This service disables all MPs at once, but all previously prepared MPs remain prepared.

2.1.20.2.4 Read the measurement results

To read the measurement results, there is the API `Rtm_GetMeasurementItem()`. It supports all types of MPs (runtime, CPU load, tasks).

Per call only one result type can be read of exactly one MP. The MP id can be selected by the macros also used for Rtm_Start() and Rtm_Stop():

- RtmConf_RtmMeasurementPoint_<MP_Name>
  - Where MP_Name is the configured short name of the MP

Note
The CPU load MP id can also be determined by calling Rtm_GetConfiguredMPIdOfCPULoadOfCommonConst(<CoreId>).

But note that this causes an exception if called before initialization of Rtm in Post-build configurations.

For CPU load MPs, the following result types (in %) are available:

> RTM_ITEM_CPU_LOAD_AVERAGE
> RTM_ITEM_CPU_LOAD_CURRENT
> RTM_ITEM_MIN
> RTM_ITEM_MAX
> RTM_ITEM_RELATIVE_MAX

Note
The RTM_ITEM_CPU_LOAD_CURRENT reports the CPU load between the last two calls of Rtm_MainFunction. If the Idle Task was not executed between two calls of Rtm_MainFunction, this will report 100% CPU load. This is most likely no indication for an overloaded system. Only if the CPU load remains 100% for multiple main function cycles this could be an indication for an overloaded system.

For runtime/task MPs, the following result types (in microseconds) are available:

> RTM_ITEM_RUNTIME_AVERAGE
> RTM_ITEM_RUNTIME_OVERALL
> RTM_ITEM_MIN
> RTM_ITEM_MAX
> RTM_ITEM_RELATIVE_MAX

2.1.20.2.5 Clear the measurement results

To clear all measurement results, the API `Rtm_ClearMeasurementResults()` can be called. The actual clear is executed in the next call of Rtm_MainFunction.

This service disables all MPs at once, but all previously prepared MPs remain prepared.

2.1.20.2.6 CPU load measurement

There are special APIs specified in the section 2.1.9.3.1 to start and stop the CPU load MPs with more comfort. This is only available if ./RtmCpuLoadControlMode is set to C_API.

Note that all other APIs mentioned in this section are always available.

2.1.21 Trigger reading via XCP

RTM provides the API Rtm_TriggerReading() to trigger the reading of current measurement results via XCP. To guarantee data consistency, this API stops the measurement before reading is actually performed. Only endless parallel measurement is supported.

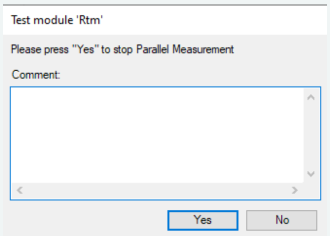
The API and its limitations are specified in 4.2.21.

Note
It is not required to call the function Rtm_TriggerReading() if measurement results are gathered via Rtm_GetMeasurementItem() (C-API), as this function can always be called during runtime.

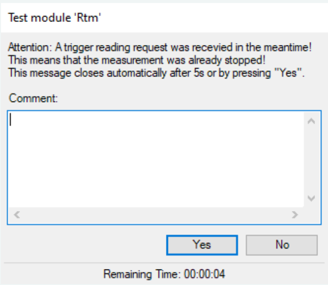
This triggers the generation of all csv reports (report.csv, runnable_report.csv, Report_LET.csv) if the RTM CAPL test node is used. Please refer to section 2.1.3 for more information about the CAPL test node and refer to section 2.1.4 for the information about the reports.

Caution

When an endless parallel measurement is started via CANoe test panel, a pop-up window occurs to stop the measurement:



This window cannot be closed if measurement is stopped by calling Rtm_TriggerReading(). Therefore, if this API is called during active measurement the pop-up still has to be answered manually, but a second pop-up window occurs afterwards to notify user that the measurement was stopped before:



This second pop-up closes automatically after 5s. Additionally, it can be closed manually.

When the measurement is actually stopped, the csv reports with special infix are created:

- report_<NameOfDay>_<Month>_<Day>_<CurrentTime>_<Year>.csv
- Report_LET_<NameOfDay>_<Month>_<Day>_<CurrentTime>_<Year>.csv
- runnableReport_<NameOfDay>_<Month>_<Day>_<CurrentTime>_<Year>.csv

The existence of those reports indicate that the measurement was stopped due to Rtm_TriggerReading(). The infix is the time and date, so it is possible to see when the measurement was actually stopped.

The common reports without infixes are generated when the second pop-up window closes. The content of the reports with and without infixes are equal, as the measurement was already stopped before the reports without infix were generated.

2.1.22 LET

LET is a feature implemented by RTE and RTM. The feature can be activated in RTE, which offers a solving action to automatic create LET specific MPs. The APIs Rtm_Start() and Rtm_Stop() for those MPs are generated by RTE and therefore called by RTE.

RTM monitors the LET interval with the corresponding MP. If the configured `./RtmTargetRuntime` elapses, the `./RtmThresholdCallback` callback is invoked. The callback is generated and provided by RTE.

If any LET violation is detected, the RTE calls the macro Rte_LET_ReportError(). By default, this macro is defined to Rtm_TriggerReading(). Please refer to [4] for more information about the RTE and LET.

Caution
If CANoe/XCP is used, the LET specific system variables starting with "Rte_LET_Error_" have to be activated in the XCP/CCP window. Refer to section 3.7.1.

If RTM shall gather the results via Rtm_GetMeasurementItem(), the macro Rte_LET_ReportError() shall be re-defined. This is required if application shall handle the LET results. E.g., this is the case if another XCP master than CANoe shall be used.

It is recommended to implement the handling differently for single and multi core.

2.1.22.1 Single core C-API handling of LET

The application function can read the required measurement results (all, only LET results, or only the currently affected LET results) directly with Rtm_GetMeasurementItem().

Here an example how the LET relevant results can be read by C-API.

Example

Single core: If maximum runtime, the LET interval, the relative remaining runtime, and the LET error shall be cached for the violated LET MP, add the following struct type + the actual struct:

```
typedef struct
{
  uint32 Max;
  uint32 Inverval;
  uint8  RelativeRemaining;
  uint8  LetError;
} Appl_LetResultsType;


Appl_LetResultsType Appl_LetResults[RTM_NUMBER_OF_LET_MPS];
```

Declare any application function and re-define Rte_LetReportError() to call the new application function:

```
#define Rte_LetReportError(measurement, error) MyAppl_LetReportError(measurement, error)

void MyAppl_LetReportError(uint32 measurement, uint8 error);
```

Implement the function in application (this only updates the affected MP):

```
void MyAppl_LetReportError(uint32 measurement, uint8 error)
{
  uint32 activeMpId;
  uint32 remaining;
  uint32 max;
  Std_ReturnType retVal;

  activeMpId = Rtm_ConfiguredToActivatedMPIds[measurement];
  if (activeMpId < RTM_NUMBER_OF_ACTIVATED_MPS)
  {
    retVal  = Rtm_GetMeasurementItem(
                measurement,
                RTM_ITEM_RELATIVE_REMAINING,
                &remaining);
    retVal |= Rtm_GetMeasurementItem(
                measurement,
                RTM_ITEM_MAX,
                &max);
    if (retVal == E_OK)
    {
      uint32 index = measurement - RTM_FIRST_LET_MP;
      Appl_LetResults[index].Max = max;
      Appl_LetResults[index].Interval =
        (uint32)(Rtm_ThresholdTimes[activeMpId] / RTM_TICKS_PER_MICROSECOND);
      Appl_LetResults[index].RelativeRemaining = (uint8)remaining;
      Appl_LetResults[index].LetError = error;
    }
  }
}
```

If all MPs shall be updated, iterate over all LET MPs with `RTM_NUMBER_OF_LET_MPS` and `RTM_FIRST_LET_MP`.

2.1.22.2 Multi core C-API handling of LET

For multi core, a special handling is required.

The application function, called by RTE, should stop the RTM measurement with Rtm_StopMeasurement() and set a flag that a LET violation occurred.

Additionally, a cyclically called application function is required. Check if the flag is set, and that the Rtm_MainFunction was called at least twice since the flag was set. Afterwards it is guaranteed that the measurement is stopped. Read the required measurement results (all, only LET results, or only the currently affected LET results) with Rtm_GetMeasurementItem().

Example

Multi core: If maximum runtime, the LET interval, the relative remaining runtime, and the LET error shall be cached for the violated LET MP, add the following struct type + the actual struct:

```
typedef struct
{
  uint32 Max;
  uint32 Inverval;
  uint8  RelativeRemaining;
  uint8  LetError;
} Appl_LetResultsType;


Appl_LetResultsType Appl_LetResults[RTM_NUMBER_OF_LET_MPS];
```

Declare any application function and re-define Rte_LetReportError() to call the new application function:

```
#define Rte_LetReportError(measurement, error) MyAppl_LetReportError(measurement, error)

void MyAppl_LetReportError(uint32 measurement, uint8 error);
boolean MyAppl_LetErrorOccurred = FALSE;
uint8 MyAppl_DelayCounter = 0u;
```

Implement the function in application (this only updates the affected MP):

```
void MyAppl_LetReportError(uint32 measurement, uint8 error)
{
  Rtm_StopMeasurement();
  MyAppl_LetErrorOccurred = TRUE;
}

void MyAppl_CyclicLetErrorHandler()
{
  if (MyAppl_LetErrorOccurred == TRUE)
  {
    if (MyAppl_DelayCounter == 2)
    {
      uint32 activeMpId;
      uint32 remaining;
      uint32 max;
```

```
      Std_ReturnType retVal;

      activeMpId = Rtm_ConfiguredToActivatedMPIds[measurement];
      if (activeMpId < RTM_NUMBER_OF_ACTIVATED_MPS)
      {
        retVal  = Rtm_GetMeasurementItem(
                    measurement,
                    RTM_ITEM_RELATIVE_REMAINING,
                    &remaining);
        retVal |= Rtm_GetMeasurementItem(
                    measurement,
                    RTM_ITEM_MAX,
                    &max);
        if (retVal == E_OK)
        {
          uint32 index = measurement - RTM_FIRST_LET_MP;
          Appl_LetResults[index].Max = max;
          Appl_LetResults[index].Interval =
            (uint32)(Rtm_ThresholdTimes[activeMpId] / RTM_TICKS_PER_MICROSECOND);
          Appl_LetResults[index].RelativeRemaining = (uint8)remaining;
          Appl_LetResults[index].LetError = error;
        }
      }

      /* Reset the counter and flag. */
      MyAppl_DelayCounter = 0u;
      MyAppl_LetErrorOccurred = FALSE;

      /* Re-start the endless parallel measurement (if required). */
      Rtm_StartMeasurement(0u);
    }
    else
    {
      MyAppl_DelayCounter++;
    }
  }
}
```

If all MPs shall be updated, iterate over all LET MPs with `RTM_NUMBER_OF_LET_MPS` and `RTM_FIRST_LET_MP`.

2.1.23 Result persistency

All raw measurements of RTM are persisted if the feature is enabled by referring a NvM block:

> `/MICROSAR/Rtm/RtmGeneral/RtmNvMResults`

For this, the API Rtm_Shutdown() must be called during ECU shutdown. This triggers the persisting during NvM_WriteAll() if any MP has a measurement result.

Therefore, if no measurement was executed since ECU start up, the results are not persisted.

The results are restored during ECU start up during NvM_ReadAll(). If the NvM block is in invalid state, the measurement results are cleared during RTM initialization and their persistence is requested.

Note that the API Rtm_ClearMeasurementResults() does not trigger the persistence of cleared measurement results. If this is required, it is possible to mark the NvM block dirty from application (including Rtm.h and NvM.h is required):

```
> (void)NvM_SetRamBlockStatus((NvM_BlockIdType)Rtm_GetResultsNvMSnv(), TRUE);
```

Note
The NvM must include the Rtm.h if any persistency feature of RTM is used. If MSR NvM is used, add 'Rtm.h' to the `/MICROSAR/NvM/NvMCommonVendorParams` list.

2.2 Initialization

Before calling any other functionality (Except auto start MPs) of the RTM module, the initialization function `Rtm_Init()` has to be called. In multi core systems there is one `Rtm_Init_<CoreId>()` per core. It is recommended to call the core specific init functions on the corresponding cores.

The RTM module assumes that some variables are initialized with certain values at start-up. As not all embedded targets support the initialization of RAM within the start-up code the RTM module provides the function `Rtm_InitMemory()`. This function has to be called during start-up and before `Rtm_Init()` is called. For API details refer to chapter 4.2.4 'Rtm_InitMemory'.

2.3 Main Function

RTM provides one main function (`Rtm_MainFunction()`) in case of single core and two or more main functions (`Rtm_MainFunction_<CoreID>()`) in case of multi core systems. These functions have to be called cyclically by the Basic Software Scheduler or a similar component.

The main function(s) handle(s) measurement requests from RTM's CANoe frontend and activates/deactivates measurement sections on the ECU during runtime. Additionally, it controls the CPU load measurement and calculates the current result.

In single and multi core system only one main function triggers sending of measurement results. This is because it cannot be expected that lower layer XCP does support multi core itself.

2.4 Error Handling

2.4.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()`. If development error reporting is enabled (i.e. pre-compile parameter `RTM_DEV_ERROR_DETECT==STD_ON`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

The reported RTM ID is 255.

The reported service IDs identify the services which are described in chapter 0. The following table presents the service IDs and the related services:

Service ID	Service
2	Rtm_GetVersionInfo
3	Rtm_Init
4	Rtm_InitMemory
5	Rtm_MainFunction(_<CoreID>)
6	Rtm_StartMP
7	Rtm_StopMP
8	Rtm_GetMeasurementItem
13	Rtm_Schedule
14	Rtm_StartNetMP
15	Rtm_StopNetMP
16	Rtm_GetCpuLoadHistogram
17	Rtm_GetTaskResponseTimeHistogram
18	Rtm_GetTaskStackUsage
19	Rtm_ClearHistogramResults
20	Rtm_StartMeasurement
21	Rtm_StopMeasurement
22	Rtm_PrepareMPSettings
23	Rtm_ClearMeasurementResults
24	Rtm_StartFlatMP
25	Rtm_StopFlatMP

Table 2-8 Service IDs

The errors reported to DET are described in the following table:

Error Code	Description
RTM_E_UNINIT	API service used without module initialization
RTM_E_WRONG_PARAMETERS	API service used with wrong parameters
RTM_E_INVALID_CONFIGURATION	RTM configuration not consistent
RTM_E_UNBALANCED	Unbalanced called measurement macros. If measurement is started but not stopped or the other way round. (i.e. If a measured function has more than one return path and measurement is not stopped at all paths)

Error Code	Description
RTM_E_MEASUREMENT_POINT_RESULT_OVERFLOWN	Rtm_Stop(Net)MP detects overflow of measurement result. The result is reset. Only reported if explicitly activated with the following code implemented in the user config file: `#define RTM_REPORT_DET_FOR_MEASUREMENT_POINT_RESULT_OVERFLOW STD_ON`

Table 2-9 Errors reported to DET

2.5 Production Code Error Reporting

No production error codes are currently defined for RTM.

3 Integration

This chapter gives necessary information for the integration of the MICROSAR RTM into an application environment of an ECU.

3.1 Scope of Delivery

The delivery of the RTM contains the files which are described in the chapters 3.1.1 and 3.1.2:

3.1.1 Static Files

File Name	Description
Rtm.c	This is the source file of RTM. It contains the implementation of the main functionality.
Rtm.h	This is the header file of RTM, which is the interface for the modules which use the services provided by RTM.
Rtm_Types.h	Header File which includes RTM specific data types.
CreateRunnableMeasurementPoints.dvgroovy	This is the RTM script to generate the file ‘Rte_Hook_Rtm.c’ and to create the corresponding runnable MPs. It is located in ‘./external/DaVinciConfigurator/AutomationScripts’.

Table 3-1 Static files

3.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator.

File Name	Description
Rtm_Cfg.c	This is the pre-compile time configuration source file.
Rtm_Cfg.h	This is the RTM configuration header file.
Rtm_Cbk.h	Header File which contains the function prototypes of the threshold callback functions.
Rtm_Canoe.xml	XML file which describes the measurement section configuration for RTM’s CANoe frontend.
Rtm_Canoe.can	CAPL file which contains RTM’s CANoe frontend logic for Rtm’s test panel.
Rtm_Style.xslt	Transformations file for HTML-report generation.
Rtm_TestSetup.tse	Configuration file for RTM’s CANoe frontend (Requires CANoe in Version 8.1 or higher).
RtmCan.cin	CAPL file which contains RTM’s CANoe frontend logic.
Rte_Hook_Rtm.c	Source file implementing VFB hook functions for selected runnables. This file is generated by the provided AI script 'CreateRunnableMeasurementPoints'.
Rtm_MemMap.h	MemMap header.

Table 3-2 Generated files

3.2 Include Structure

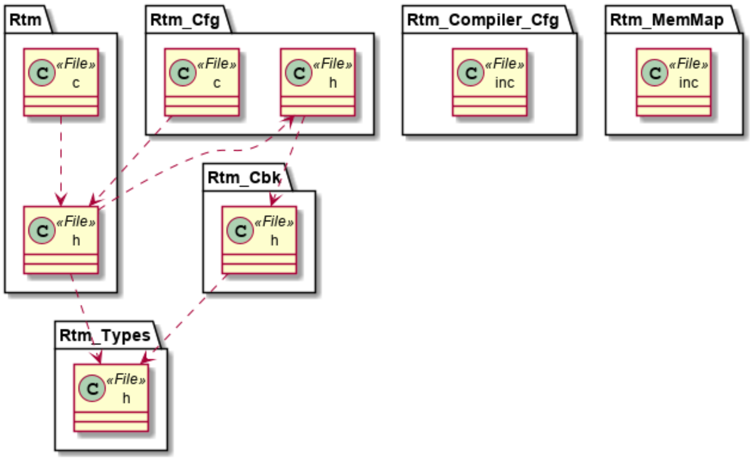


Figure 3-1 Include Structure

3.3 Critical Sections

RTM has three critical sections:

> RTM_EXCLUSIVE_AREA_0

> RTM_EXCLUSIVE_AREA_1

> RTM_EXCLUSIVE_AREA_2

AREA_0 is used by RTM to protect internal data of MPs. This section is used whenever a measurement is started or stopped. The lock duration is expected to be short since it only contains a small section of code.

AREA_1 alternatively used for MPs. It protects not only the start and stop functions of RTM, but also the complete runtime between a start and stop. This is useful to protect the measurement results from errors caused by interrupts and preemption. This option can be activated by parameter `/MICROSAR/Rtm/RtmMeasurementPoint/RtmDisableInterrupts`.

Depending on the measurement section, the length of this area can reach from a single line of code (<1 µs) to several complete functions (>1 ms). Therefore it is recommended to use this option with caution.

AREA_2 is used by RTM to protect internal data errors caused by preemption and interrupts. It is used for the main function(s) and the CPU load measurement.

It is recommended to implement AREA_1 and AREA_2 as global interrupt lock. But to reduce runtime introduced by RTM AREA_0 can be implemented empty without interrupt lock. Depending on the number of simultaneously active MPs the reduction of runtime can be great.

Example configuration:

> 100 MPs

- All MPs are enabled
- Parallel measurement is chosen
- Each MP is executed 100 times per second.
  - Rtm_Start(…) is called 100 times and
  - Rtm_Stop(…) is called 100 times
- All MPs enter and leave AREA_0 in Rtm_Start(…) and Rtm_Stop(…)
  - `RtmDisableInterrupts` == OFF
- The execution time of entering critical section is 1µs.
- The execution time of leaving critical section is 1µs.

This configuration result in:

```
ExeTime[s] = #MeasPts * #APIsRtmStartStop * #Calls *
    #APIsExclusiveArea * ExeTimeExclusiveAreaAPIs
        100 * 2 * 100 * 2 * 1µs = 0.04s
```

execution time for AREA_0.

This means the CPU load introduced by calling AREA_0 is:

$$CPUloadAREA_0 = (0.04s * 100) / 1s = 4\%$$

Note
This additional CPU load is only introduced if the measurement is active. If no RTM measurement is active its introduced CPU load is nearly null.

Caution
Not implementing AREA_0 as interrupt lock increases the probability to get erroneous measurement results.

3.4 OS

This chapter describes the configruation of OS and timinig hooks needed for RTM.

3.4.1 Use of OS Timing Hooks:

The RTM module requires OS timing hooks for the following features:

> Net and Flat Execution time
> Task Response Time
> CPU Load measurement
> Histograms

The OS timing hooks used are:

> OS_VTH_ACTIVATION: It is called on the caller core when that core has successfully performed the activation of TaskId on the destination core. On single core systems both core IDs are identical.
> OS_VTH_SCHEDULE: It is called on a core when it performs a thread switch (from one task or ISR to another task or ISR).
> OS_VTH_SETEVENT: It is called on the caller core when that core has successfully performed the event setting on the destination core.

MPs of basic tasks are started within OS_VTH_ACTIVATION, whereas MPs of extended tasks are started within OS_VTH_SETEVENT. All task response time MPs are stopped in the hook OS_VTH_SCHEDULE.

The following Vector OS hook function must be implemented to call the following RTM function:

Hook Function	RTM Function
OS_VTH_SCHEDULE	Rtm_Schedule

Table 3-3 OS Hook Function mapping (OS_VTH_SCHEDULE)

Note
If the hook function OS_VTH_SCHEDULE is already implemented within another component, this implementation could be extended by the call of Rtm_Schedule in order to implement the execution time measurement. The Rtm.h must be included.

Note
Rtm_Schedule is only required for net and/or flat execution time.

For more detail information about the OS Timing Hooks, please refer [5].

3.4.1.1 Configuration of OS

The OS_VTH_SCHEDULE is automatically implemented if the Rtm.h is included by the Os using the OS configuration (/MICROSAR/Os/OsOS/OsDebug/OsTimingHooksIncludeHeader).

Also choose the debug support parameter (/MICROSAR/Os/OsOS/OsDebug/OsORTIDebugSupport) to the preferred value. Please refer [5] for more information.

3.4.1.2 Alternative tracing tool

If there is requirement, to use another tracing tool for VTHs implementation, then the Rtm_OsVthSchedule, Rtm_OsVthActivation and Rtm_OsVthSetEvent functions shall be called within the VTHs as in example described. The corresponding tool must include Rtm.h. In this case, (/MICROSAR/Os/OsOS/OsDebug/OsTimingHooksIncludeHeader) does not include Rtm.h.

Example

```
#ifndef MYAPPL_H_
#define MYAPPL_H_

#include "Os.h"
# define OS_VTH_ACTIVATION(TaskId, DestCoreId, CallerCoreId) { \
  Rtm_OsVthActivation((uint32)(TaskId), (uint16)(DestCoreId),
(uint16)(CallerCoreId)); \
}

# define OS_VTH_SETEVENT(TaskId, EventMask, StateChanged, DestCoreId,
CallerCoreId) { \
  Rtm_OsVthSetEvent((uint32)(TaskId), (boolean)(StateChanged),
(uint16)(DestCoreId), (uint16)(CallerCoreId)); \
  (void)(EventMask); \
}

# define OS_VTH_SCHEDULE(FromThreadId, FromThreadReason, ToThreadId,
ToThreadReason, CallerCoreId) { \
  Rtm_OsVthSchedule((uint32)(FromThreadId),
(uint32)(FromThreadReason),(uint32)(ToThreadId),(uint32)(ToThreadReason),(uint16)
(CallerCoreId)); \
}

#include "Rtm.h"
#endif
```

3.4.2 Use of OS Pre-/PostTaskHooks and Pre-/PostISRHooks:

An alternative approach used to measure net and flat execution time is using the service function Rtm_Schedule(). Please refer [5] for more information.

> **Note**
> The Pre- and PostISRHooks are only available in the MSR4 Gen6 OS. Because of this and the increased reliability of the measurement results, the first mechanism (OS TiminigHooks) is recommended.

> **Caution**
> Use one mechanism exclusively to measure net and/or flat execution time.

3.5 Embedded Code

This chapter describes the steps to integrate RTM in the embedded code of an existing project.

3.5.1 Timestamp Acquisition

RTM requires timestamps from a real-time source (counter/timer). It is possible to use an already configured or a dedicated counter. It has to be configured as free running counter. After reaching its end value (0 or MAXVAL) the counter has to continue counting beginning with its start value. An interrupt after reaching the end value is not required.

RTM expects a callback function or function like macro which returns a 16-Bit respectively a 32-Bit timestamp from the counter. This callback has to be implemented by the application. The name of this callback can be defined in DaVinci Configurator by parameter:

- `/MICROSAR/Rtm/RtmGeneral/RtmGetMeasurementTimestampFct`

The prototype is:

```
Rtm_TimestampType <Function Name>(void)
```

This user implemented function shall return the current counter value, therefore it is not required to detect or handle counter overflows.

RTM must be configured corresponding to the counter's frequency. Therefore, the RTM provides the following parameter (in ticks per ms):

- `/MICROSAR/Rtm/RtmGeneral/RtmMeasurementCtrFrequency`

Example
If the used counter has a frequency of 10MHz, the parameter `RtmMeasurementCtrFrequency` must be set to 10,000.

The default settings expect a counter with maximum value of 0xFFFF. With a frequency of 10MHz this results in a maximum measurement of 2^16 / f_{ctr} = 6,5ms. This means that the Rtm_Start(..) / Rtm_Stop(..) must be called within this time.

The result is that a 10ms task cannot be measured with this setting. To enable such extended measurement durations, RTM provides two mechanisms:

> The timer overrun support (`/MICROSAR/Rtm/RtmGeneral/RtmTimerOverrunSupport`) or
> The 32bit timer support (`/MICROSAR/Rtm/RtmGeneral/Rtm32BitTimer`)

Note
The features “timer overrun support” and “32-Bit timer” are only exclusively available. Therefore, it is not possible to enable both features together.

3.5.1.1 Timer overrun support

The timer overrun support extends a 16-Bit timer by overrun handling. If between two calls of Rtm_Start(..) and Rtm_Stop(..) the timer elapses (once or multiple times) then an overrun counter is incremented. In the execution of Rtm_Stop(..) this counter is taken into account to calculate the real time since start.

The counter overruns are implicitely detected by the following RTM APIs:

> Rtm_Start(..)
> Rtm_Stop(..)
> Rtm_MainFunction
> Rtm_Schedule

If those functions are not called often enough to detect each overrun (at least two calls per counter overrun), RTM provides the additional function Rtm_CheckTimerOverrun, which must be called cyclically.

The timer overrun support enables the RTM to measure about 429.5s if the counter frequency is 10MHz (2^32 / f_{ctr} = 2^32 / 10MHz = 429.5s).

The drawback of this feature is the increased runtime and memory consumption introduced by the overrun counters.

3.5.1.2 32-Bit Timer

It is recommended to use a 32-Bit counter if required instead of using the timer overrun support because of reduced runtime consumption.

The 32-Bit timer overrun support enables the RTM to measure about 429.5s if the counter frequency is 10MHz (2^32 / f_{ctr} = 2^32 / 10MHz = 429.5s).

> **Caution**
> If the 32-Bit timer is used, the RTM expects a maximum counter value of 0xFF FF FF FF. But some counters have a smaller maximum value. In this case, a user configuration file is required:
>
> - `/MICROSAR/Rtm/RtmGeneral/RtmUserConfigFile`
>
> Add the following example code to the user configuration file (if 24-Bit counter is used => maximum value is 0xFF FF FF):
>
> ```
> #undef RTM_MEASUREMENT_MAX_VAL
>
> #define RTM_MEASUREMENT_MAX_VAL ((uint32)(0xFF FF FFUL))
> ```

3.5.2 Measurement Points

MPs are defined by the following function like macros:

```
Rtm_Start(RtmConf_RtmMeasurementPoint_<Measurement Name>);

Rtm_Stop(RtmConf_RtmMeasurementPoint_<Measurement Name>);
```

`<Measurement Name>` is the name of the MP. It reflects the value of the DaVinci Configurator parameter: **Short Name** (/MICROSAR/Rtm/RtmMeasurementPoint). A `<Measurement Name>` may only be used if it is also configured in DaVinci Configurator.

MPs can be implemented within any task or function. Each source file with MPs has to include “Rtm.h”.

> **Caution**
> MPs have to be implemented balanced: I.E. Rtm_Start(A) must be followed by Rtm_Stop(A). In addition, Rtm_Start(A) has to be called before Rtm_Stop(A).

It is allowed that a MP is called nested (i.e. a MP within a function which is called recursively), however only the outmost measurement will be evaluated. Therefore, there must be as many Rtm_Stop calls as Rtm_Start calls.

Note
In case of nested MPs, measured results differ from actual runtimes due to the code overhead introduced by RTM.

This effect can be avoided by using variant Serial Measurement.

Note
RTM causes additional resource usage (ROM/CPU-time) in the measured c-module.

Caution
The RTM was tested with CANoe 8.5 SP5 and 9.0 SP4 in a configuration containing more than 200 active MPs. It is known that older CANoe versions only run stable with less MPs.

3.5.3 CPU Load Measurement

Measurement of the CPU's overall load is enabled if the optional container `/MICROSAR/Rtm/RtmCpuLoadMeasurement` exists. For more details about OS timing hooks and OS configuration for RTM module, please refer to 3.4.1 and 3.4.1.1 respectively.

For each core, a separate MP is added, called `Rtm_CpuLoadMeasurement` respectively in multi core `Rtm_CpuLoadMeasurement_<CoreId>`. Their parameter must be set as following:

- `./RtmMeasurementPointType` to `CPU_load`
- `./RtmMeasurementType` to `GrossExecutionTime`

This feature uses the OS timing hook OS_VTH_SCHEDULE (refer to 4.2.24) which is automatically generated in the Rtm_Cfg.h. It is implemented to start the CPU load MP when the idle/background task is terminated, preempted, or it must wait for an event. The CPU load MP is stopped when the idle/background task is activated, resumed, or the event is set.

For this generation, the Rtm requires the name of the actually used idle/background task. The OS always provides the default idle task "IdleTask_OsCore" (resp. in multi core: IdleTask_OsCore<CoreId>), but the user can create a custom idle/background task which implements an endless loop. This causes that the "IdleTask_OsCore" will never be called. Therefore, the Rtm has to use this custom idle/background task which shall be set in the parameter:

- `/MICROSAR/Rtm/RtmCoreDefinition/RtmBackgroundTaskName`

> **Caution**
> For CPU load measurement, the Rtm requires the name of the actually used idle/background task for each core, as it is not possible to detect if there is a custom idle/background task.
>
> Therefore, set the parameter "`./RtmBackgroundTaskName`" for each "`./RtmCoreDefinition`" where "`./RtmCore`" is defined.

3.5.3.1 CPU Load Histogram - Integration

The CPU load histogram feature is enabled if the optional parameter `./RtmNumberOfCpuLoadPercentiles` exist. It defines in how many percentiles the measurement result is split.

If there are 10 percentiles configured, the following 10 percentiles and the special 100% percentile are available:

> 0-9%, 10-19%, 20-29%, 30-39%, 40-49%, 50-59%, 60-69%, 70-79%, 80-89%, 90-99%, 100%

With the parameter `/RtmHyperperiod` it is configured how often a percentile is updated. It must be a multiple (including equal) of the `/MICROSAR/Rtm/RtmGeneral/RtmMainfunctionCycleTime`.

To persist the CPU load histogram results, a NvM block must be referred by `/MICROSAR/Rtm/RtmCpuLoadMeasurement/RtmNvMCpuLoadPersistencyBlock`.

The `/NvMNvBlockLength` of the NvM block is calculated as following:

> ((4 + (`/Rtm/RtmCpuLoadMeasurement/RtmNumberOfCpuLoadPercentiles` + 1) * 4) * `NumberOfCores)`
>
> > `NumberOfCores` is defined as number of `/Rtm/RtmCoreDefinition` with configured `./RtmCore`.

To ensure that the calculation is correct, enable the parameter `/NvMBlockLengthCheck` and optionally `/NvMBlockLengthCheckType`.

The `/NvMRamBlockDataAddress` has to be specified as follows:

> `Rtm_CpuLoadPersistencyData`

Those NvM blocks are only read during ECU start-up and written during ECU shutdown, therefore the following parameter must be enabled:

> `/NvMSelectBlockForReadAll`
> `/NvMSelectBlockForWriteAll`
> `/NvMBlockUseSetRamBlockStatus`

The NvM must include the Rtm.h if any persistency feature of RTM is used. If MSR NvM is used, add 'Rtm.h' to the `/MICROSAR/NvM/NvMCommonVendorParams` list.

Caution
On multi core:

➔ If the histogram results are persisted, make sure that the NvM_ReadAll is performed in initialization phase 1 before the OS is started.

3.5.4 Task Response Time Histogram – Integration

The task response time histogram is enabled if the parameter `/MICROSAR/Rtm/RtmCoreDefinition/RtmNumberOfTaskResponseTimePercentiles` exists. For more details about OS timing hooks and OS configuration for RTM module, please refer to 3.4.1 and 3.4.1.1 respectively.

The first parameter defines in how many percentiles the measurement result is split.

Note
This parameter can also be found via the 'Runtime Measurement' comfort editor. Click on 'Measurement Points', on 'Task Service Routines', and on "Number of Task Response Time Percentiles".

If there are 10 percentiles configured, the following 10 percentiles and the special 200% percentile are available:

> 0-19%, 20-39%, 40-59%, 60-79%, 80-99%, 100-119%, 120-139%, 140-159%, 160-179%, 180-199%, >=200%

For each task in `/Os/OsTask` there must be one MP with the same name as the task, but with the prefix "Task_".

> Their `/RtmMeasurementPointType` must be set to `Task`

- Their `/RtmMeasurementType` must be set to `GrossExecutionTime`
- Their `/RtmAutostartEnabled` must be enabled
- They must assign the correct core with `/RtmAssignedToCore`

The target runtime, defining 100% runtime, is specified for each task separately on its MP with the parameter `/MICROSAR/Rtm/RtmMeasurementPoint/RtmTargetRuntime`.

To persist the task response time histogram results, a NvM block must be referred by `/MICROSAR/Rtm/RtmCoreDefinition/RtmNvMTaskResponseTimePersistencyBlock`.

The `/NvMNvBlockLength` of the NvM block is calculated as following:

- (8 + (`/MICROSAR/Rtm/RtmCoreDefinition/RtmNumberOfTaskResponseTimePercentiles` + 1) * 4) * <NumberOfTasksOnThisCore>

To ensure that the calculation is correct, enable the parameter `/NvMBlockLengthCheck` and optionally `/NvMBlockLengthCheckType`.

The `/NvMRamBlockDataAddress` has to be set to `Rtm_TaskResponseTimeStruct_<CoreId>`.

Note: in single core the name is `Rtm_TaskResponseTimeStruct_0`.

Those NvM blocks are only read during ECU start-up and shutdown, therefore the following parameter must be enabled:

- `/NvMSelectBlockForReadAll`
- `/NvMSelectBlockForWriteAll`
- `/NvMBlockUseSetRamBlockStatus`

The NvM must include the Rtm.h if any persistency feature of RTM is used. If MSR NvM is used, add ‘Rtm.h’ to the `/MICROSAR/NvM/NvMCommonVendorParams` list.

Caution

On multi core:

➔ If the histogram results are persisted, make sure that the NvM_ReadAll is performed in initialization phase 1 before the OS is started.

This feature uses the OS timing hooks OS_VTH_ACTIVATION (refer to 4.2.22), OS_VTH_SETEVENT (refer to 4.2.23), and OS_VTH_SCHEDULE (refer to 4.2.24) which are automatically generated in the Rtm_Cfg.h.

3.5.5 Task Stack Usage - Integration

The task stack usage feature requires a NvM block to be enabled. Therefore, a NvM block must be referred by `/MICROSAR/Rtm/RtmGeneral/RtmNvMTaskStackUsage`.

The `/NvMNvBlockLength` of the NvM block is calculated as following:

> 4 * <NumberOfTasks>

To ensure that the calculation is correct, enable the parameter `/NvMBlockLengthCheck` and optionally `/NvMBlockLengthCheckType` for a strict check.

The `/NvMRamBlockDataAddress` has to be set to `Rtm_MaxTaskStackUsage`.

Those NvM blocks are only read during ECU start-up and written during ECU shutdown, therefore the following parameter must be enabled:

> `/NvMSelectBlockForReadAll`

> `/NvMSelectBlockForWriteAll`

> `/NvMBlockUseSetRamBlockStatus`

The NvM must include the Rtm.h if any persistency feature of RTM is used. If MSR NvM is used, add ‘Rtm.h’ to the `/MICROSAR/NvM/NvMCommonVendorParams` list.

Caution
On multi core:

➔ If the task stack usage results are persisted, make sure that the NvM_ReadAll is performed in initialization phase 1 before the OS is started.

3.6 A2L

Copy the template-file “_Master.a2l” from .\external\Misc\McData or .\external\Components\Xcp\McData to .\internal\<ProjectName>\Config\McData and rename it to “Master.a2l”. Open “Master.a2l” in a text editor and adapt all sections marked with “TODO:”.

Refer to this Master.a2l in the project settings of DaVinci Configurator.

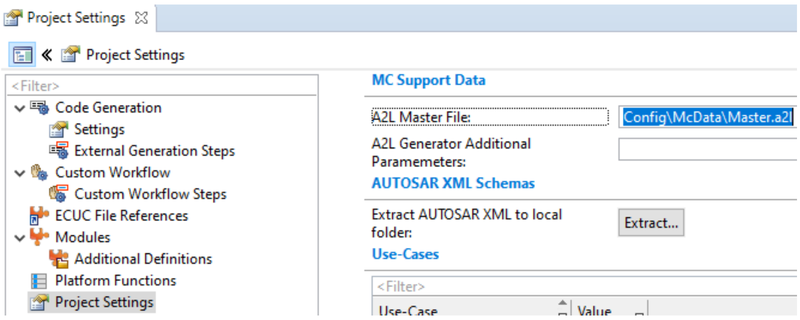


Figure 3-2 Specify Master.a2l location in project settings

When changing the Rtm configuration, always generate the “ARXML to A2L Converter” as well.

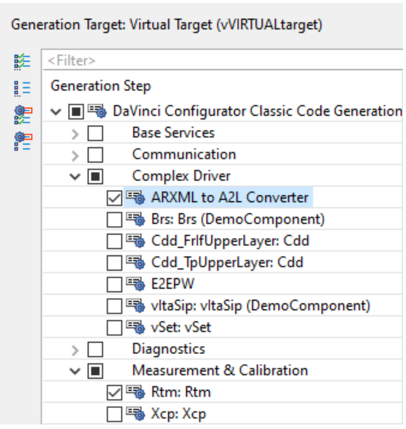


Figure 3-3 Generate Rtm and “ARXML to A2L Converter”

The A2L-Process requires an update to map the symbolic names with their memory addresses. To automate this update a batch file can be used calling the ASAP2 Updater for CANoe (<PathToCANoeInstalltion>.\ASAP2Updater\Exec\ASAP2Updater.exe). This Updater requires the following arguments:

> McData.a2l (the previously generated a2l-File)

> Updater.ini (contains some important settings for Updater)

> The map file (*.elf, *.map, *.pdb, …)

> The result a2l file (<ResultName>.a2l)

> A log File (*.log)

Example

Example content of A2L update batch file if it is placed in .\internal\<ProjectName>\CANoe:

"C:\Program Files\<TheCANoeVersion>\ASAP2Updater\Exec\ASAP2Updater.exe" -I ..\Config\McData\McData.a2l -A ..\<LocationOfYourMapFile>\<TheMapFile> -O .\AmdResult.a2l -T .\Updater.ini -L .\AsapUpdater.log

TODO: Adapt the <TheCANoeVersion>, the <LocationOfYourMapFile>, and the <TheMapFile>.

This update must be performed after every build of your project.

Please refer to the document [1] for a more detailed description on how to set up a2l-files.

Note

The file McData.a2l is rewritten each time the “ARXML to A2L Converter” is generated in DaVinci Configurator. Make sure to always use the latest version in CANoe.

Example of Updater.ini

```
[ELF]
ELF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=120

[UBROF]
UBROF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=120

[OMF]
OMF_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=120

[PDB]
PDB_ARRAY_INDEX_FORM=1
MAP_MAX_ARRAY=120  ;Must be larger than size of Rtm_Results array

[OPTIONS]
FILTER_MODE=1  ;Update only items found in MAP file. Otherwise, illegal addresses might be used
MAP_FORMAT=54  ;Refer to ASAP Updater User Manual for description of MAP file format. 54 correlates to PDB-File, 31 correlates to ELF-File.
```

3.7 CANoe

Practical Procedure
The following chapters describe the steps necessary to set up RTM's frontend in CANoe.

3.7.1 XCP configuration in CANoe

Precondition for this step is that the XCP driver on the ECU is configured appropriately. Please refer to the document [1] for a more detailed description on how to configure XCP on the ECU.

> Open **Diagnostics & XCP | XCP/CCP…** and click on **Add Device …** to add a new device.

> Select your Master-A2L-file (e.g. **AmdResult.a2l**) as adapted in chapter 3.6 and open it.

> Select the network by which XCP can connect to the ECU (e.g. **XCP on CAN**).

> Set the **ECU Qualifier** as specified in `/MICROSAR/Rtm/RtmGeneral/RtmEcuName` (the default is “Rtm”).

> If XCP uses CAN as network, adapt **Request**- and **Response ID** as configured in DaVinci Configurator 5.

> Set **DAQ Timeout [ms]** to “**0 (off)**”.

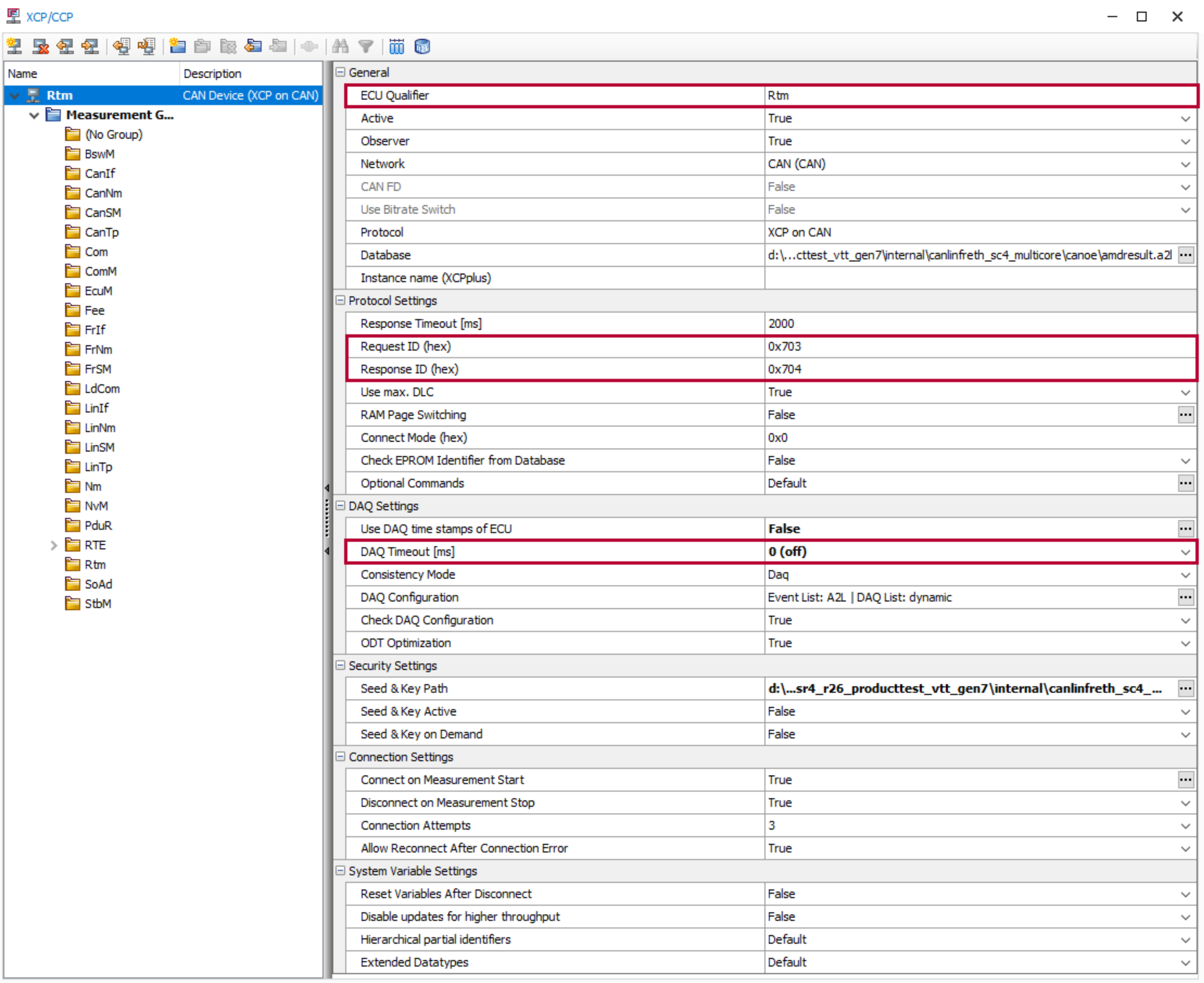


Figure 3-4 XCP/CCP Device Configuration

Next step is to add the signals provided by RTM for measurement control and result acquisition.

> Open **View | XCP/CCP** and click on folder "**Rtm**".

> Select **all** signals listed.

> While still highlighted enable all in the **Configured** column.

> Some signals have to be assigned to DAQ-Events of XCP. To assign a signal to a DAQ-Event check the **Auto Read** box, select **DAQ** in the **Measurement** column and select the event **Rtm_Evt** in the **Cycle / Event** column:

  > Rtm_Resp must always be assigned to **Rtm_Evt**.

  > If Live measurement is used, assign all measurement symbols (always Rtm_Results[<MeasurementID>].time and Rtm_Results[<MeasurementID>].count, .min and .max can be left) which have to be measured.

  > If endless measurement should be supported (measurement duration = 0s), the variable Rtm_AverageMainFunctionCycleTime must be assigned to **Rtm_Evt**.

> If LET feature is used (activated in the RTE and there are RTM MPs starting with “Rte_LET_”), in the XCP/CCP window of CANoe, the Rte_LET_Error_... variables must be set to configured. They can be found in “Rte”.

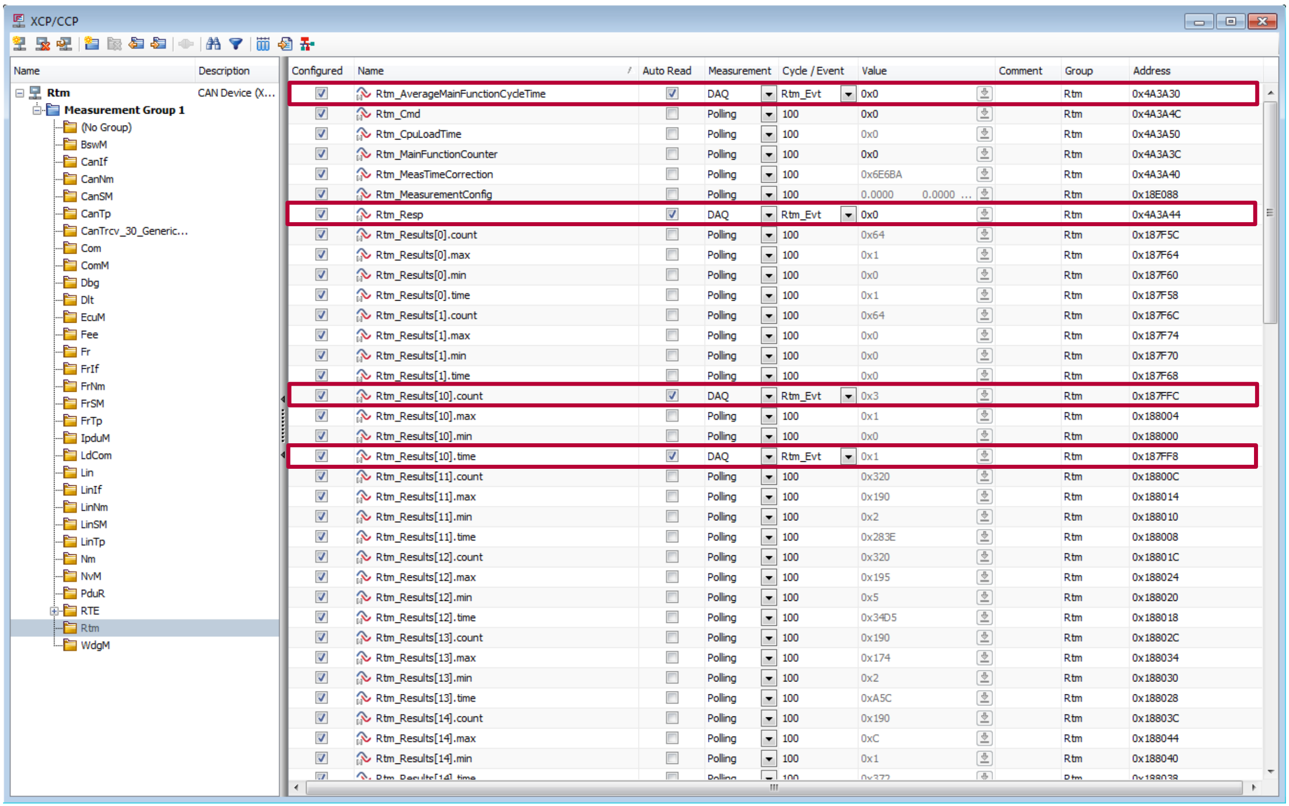


Figure 3-5 XCP/CCP Signal Configuration

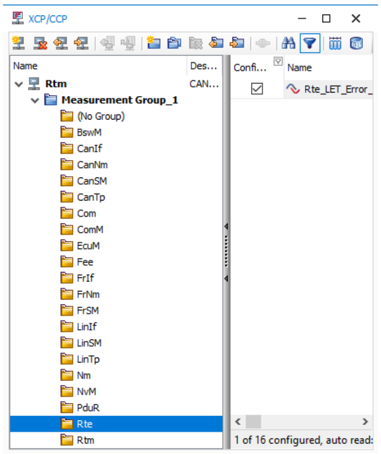


Figure 3-6 RTE system variables for LET

Note
To identify which Rtm_Results[<MeasurementID>] belongs to which MP please refer to chapter 3.7.5.

Caution
Assigning all signals to DAQ will cause RTM to work properly in any measurement mode. However it will greatly increase bus load (especially on CAN) and also affects the ECU's CPU-load.

Therefore DAQ should only be selected for live mode and/or for only a few MPs simultaneously.

3.7.2 RTM Control via Test Module

The control of RTM can be done in multiple ways.

To use the provided RTM test module is the easiest way to execute runtime measurements. How to integrate the RTM test module is described in this chapter.

The RTM test module provides a GUI for measurement control. All measurement steps must be done manually, like selecting active MPs, choosing the measurement mode and starting the measurement.

3.7.2.1 Test Setup

The next step is to set up the actual RTM frontend.

> Open **View | Test Setup**

> Right click in the white area and select **Open Test Environment**

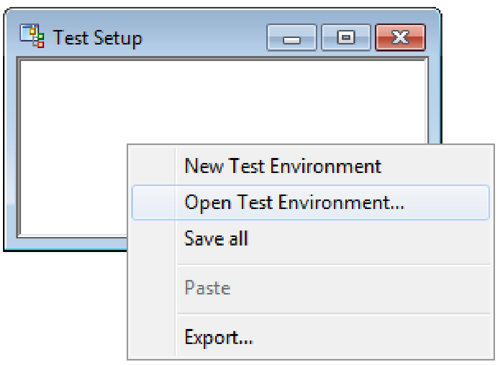


Figure 3-7 Test Setup: Import

> Choose GenData(Vtt)\Rtm_TestSetup.tse.

> Right click on the icon labeled “Rtm” and open the Configuration.

> In the tab **Buses**: Assign the bus by which CANoe shall connect to ECU via XCP

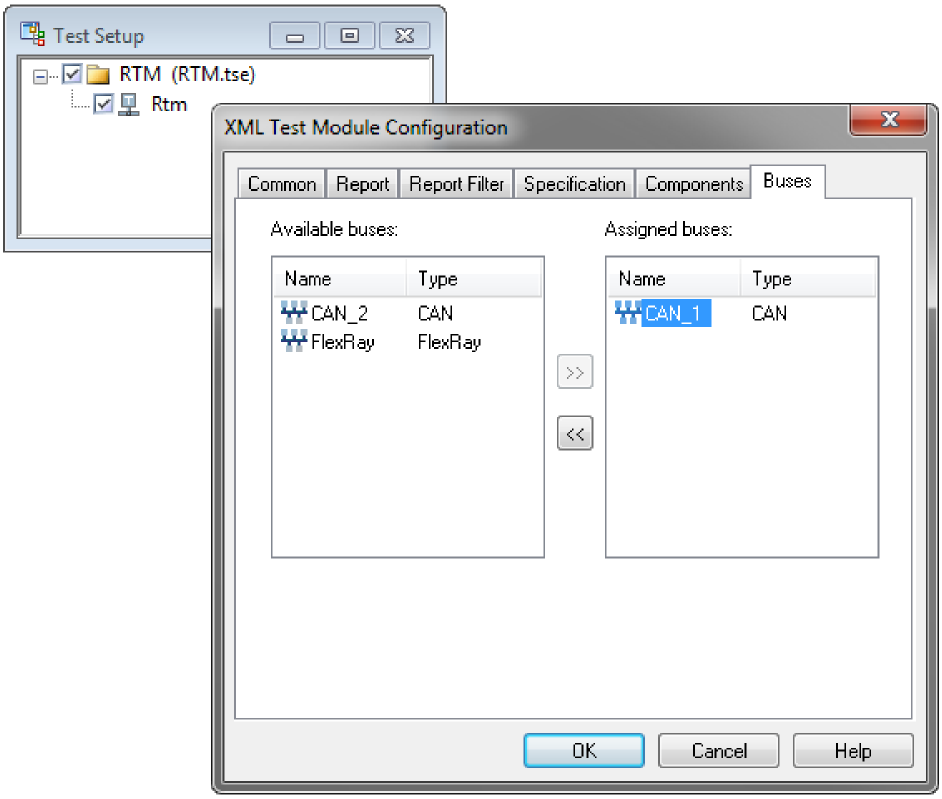


Figure 3-8 Test Setup: Configuration

3.7.2.2 Measurement

This chapter gives a brief overview about the different measurement variants and how measurement is performed.

> Click **View | Rtm** to open RTM’s control panel.

> Select the desired measurement variant with the drop-down box: **Variant**. Please refer to chapters: 2.1.1.1, 2.1.1.2 and 0 for a description of the different variations.

> Tick the MPs in the **Active Measurement Selection** which shall be measured. Only selected MPs or groups will be considered in the RTM measurement session.

Note: All auto start measurements are registered for measurement by default.

> Tick **Clear Results On ECU** if the result buffers on the ECU shall be cleared prior to measurement. If this option is deactivated the results of the previous measurement session affect upcoming measurement results.

Note: If **Clear Results On ECU** is active, also the results of auto start measurements are overwritten.

> Ensure that **Perform Measurement** is selected and click on **Start** to perform measurement. If **Perform Measurement** is deactivated the measurement report won't be updated.

> Depending on the selected Variant:

  > **Serial/Parallel:** Enter the desired measurement duration [s]

  > **Live:** Click on **Yes** to finish measurement

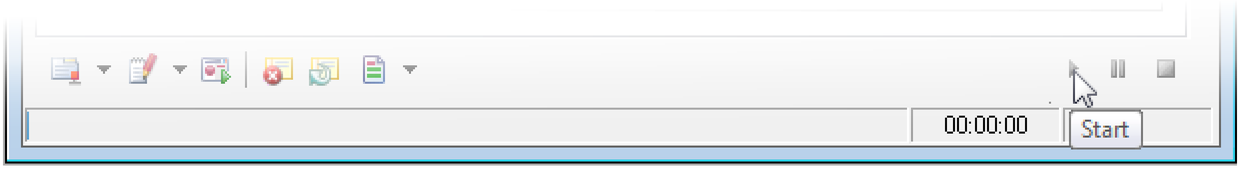


Figure 3-9 Measurement Start

Note
The average CPU-load results from the cumulated runtime during a measurement divided by the measurement duration. For long measurement durations and sporadically called MPs, the calculated CPU-load is likely to be (close to) zero.

3.7.3 RTM control via CAPL/.net

If the usage of the RTM test module is too static, RTM provides a file for dynamic usage. This file is called RtmCan.cin and provides services to control the measurement with a CAPL or .Net script. For this purpose, the file RtmCan.cin can be included by the script.

In a CAPL file the inclusion looks like following:

```
includes
{
  #include "RtmCan.cin"
}
```

Afterwards all services of RtmCan can be used. The provided external services of RtmCan are described in chapter 7.5.

Caution
Only the external services should be called by application to ensure a correct measurement behavior.

The external services start always with "RtmCan_..." whereas the internal services start with "_RtmCan_...".

The detailed description of RTM on CANoe is in chapter 7.

3.7.4 Result Report in Live Measurement Mode

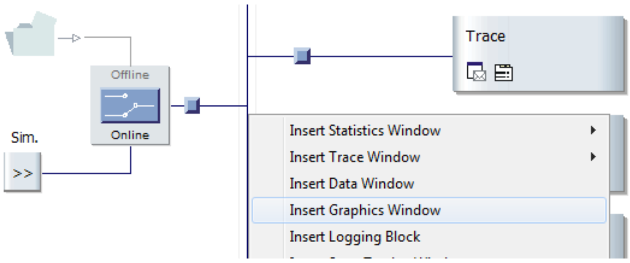
In live measurement mode no result report is generated, but there are two options to access the results.

1. Display the results in Graphics Window.
2. Use the Logging Feature of CANoe.

To display the live measurement results the Graphics Window of CANoe can be used:

> Open **View | Grahics**.

> If the **Graphics Window** is not available, open **View | Measurement Setup**, right click on the line and select **Insert Graphics Window**.



>

Figure 3-10 Insert Graphics Window

> Right click in the blank field on the left and select **Add Variables…**.

> Switch to **System variables | RtmLive**.

> Select all variables which should be measured (multiple selection is possible) and click **OK**.

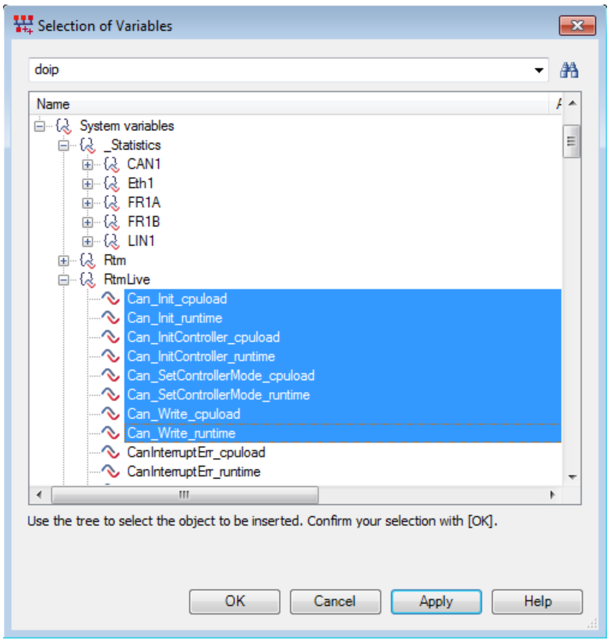


Figure 3-11 Add MPs for Live Measurement

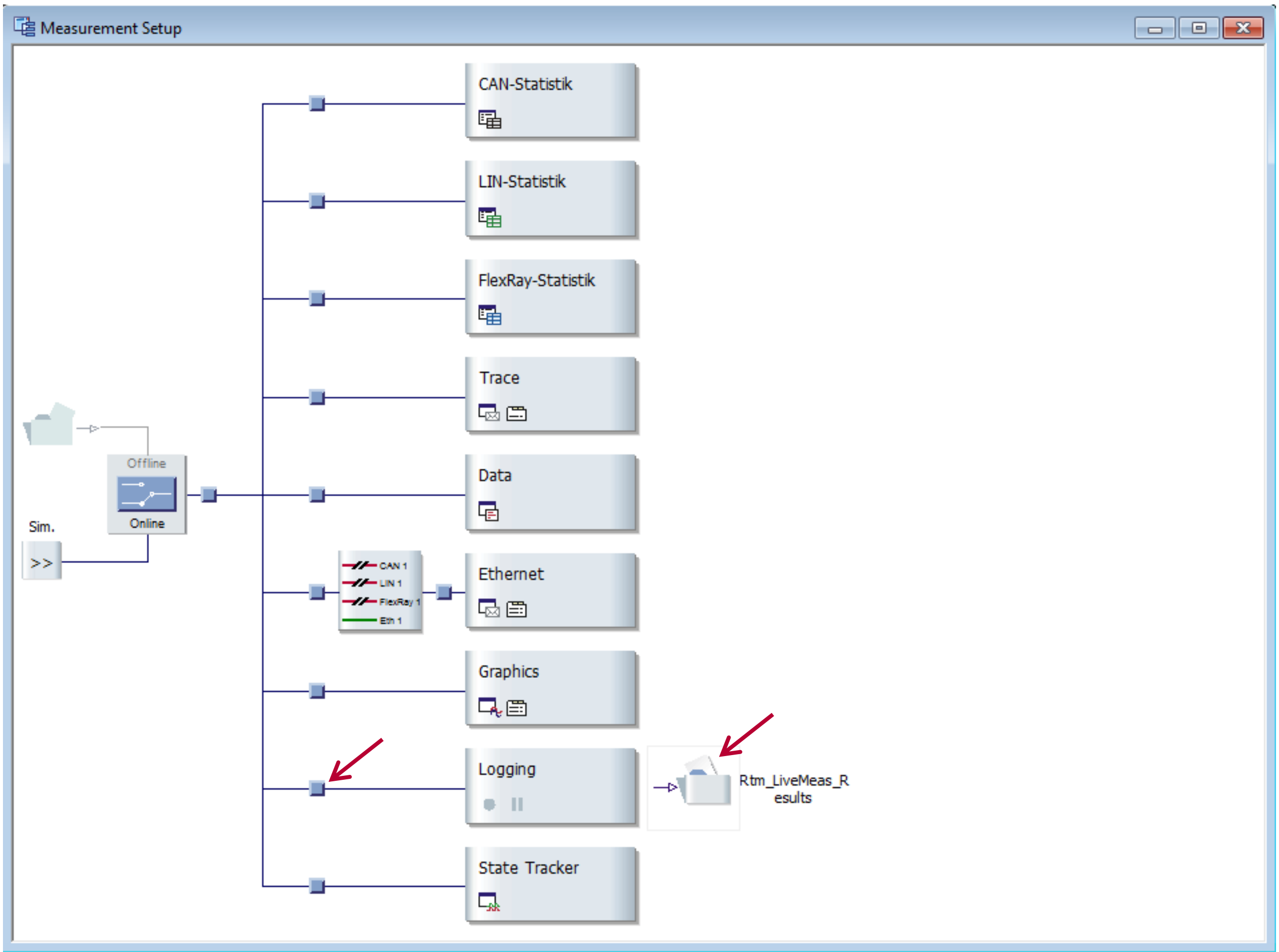
> If the **RtmLive** is not available, select the **Variant** “LIVE: Selected measurement sections …” in RTM **Test Module**. And execute the measurement once. This is required because the live measurement signals are created at runtime.

To use the Logging Feature of CANoe:

> Open **Analysis | Measurement Setup** in CANoe.

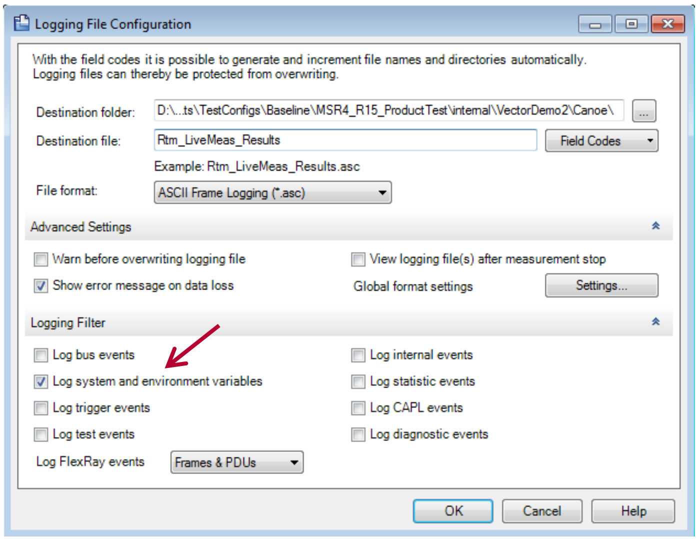
> Activate the Logging block by double click on the left square.

> Double click the folder on the right of the Logging block.



> Specify the **Destination folder**, **Destination file** and the **File format** as required.

> In the **Logging Filter** section, only select **Log system and environment variables**.



> The specified file (here Rtm_LiveMeas_Results.asc) contains the following example results after CPU load measurement in live measurement mode:

```
10.221972    SV: 2 0 0 ::XCP::Rtm::Rtm_Resp = 60000000
10.222950    SV: 2 0 0 ::XCP::Rtm::Rtm_Resp = 0
10.223930    SV: 2 0 0 ::XCP::Rtm::Rtm_Cmd = 0
10.231472    SV: 2 0 0 ::XCP::Rtm::Rtm_Results_20__time = 5a447d
10.231724    SV: 2 0 0 ::XCP::Rtm::Rtm_Results_20__count = 4d
10.231724    SV: 2 0 0 ::XCP::Rtm::Rtm_AverageMainFunctionCycleTime = 0
10.231724    SV: 1 0 1 ::RtmLive::Rtm_CpuLoadMeasurement_runtime = 7657.2
10.231724    SV: 1 0 1 ::RtmLive::Rtm_CpuLoadMeasurement_cpuload = 76.572
10.231976    SV: 2 0 0 ::XCP::Rtm::Rtm_Resp = 60000000
10.232954    SV: 2 0 0 ::XCP::Rtm::Rtm_Resp = 0
10.233934    SV: 2 0 0 ::XCP::Rtm::Rtm_Cmd = 0
10.241470    SV: 2 0 0 ::XCP::Rtm::Rtm_Results_20__time = 5b6fdd
10.241722    SV: 2 0 0 ::XCP::Rtm::Rtm_Results_20__count = 4e
10.241722    SV: 2 0 0 ::XCP::Rtm::Rtm_AverageMainFunctionCycleTime = 0
10.241722    SV: 1 0 1 ::RtmLive::Rtm_CpuLoadMeasurement_runtime = 7664
10.241722    SV: 1 0 1 ::RtmLive::Rtm_CpuLoadMeasurement_cpuload = 76.64
```

> The unit of MP Rtm_CpuLoadMeasurement_runtime is µs, the unit of Rtm_CpuLoadMeasurement_cpuload is percent.

3.7.5 Mapping Measurement ID to MP

There are two ways to figure out the relation between a MPs name and its ID.

1. Via the **Test Module**

> Open the Test Module via **Test** -> **Test Module**, choose the RTM test module (here AmdRtm).

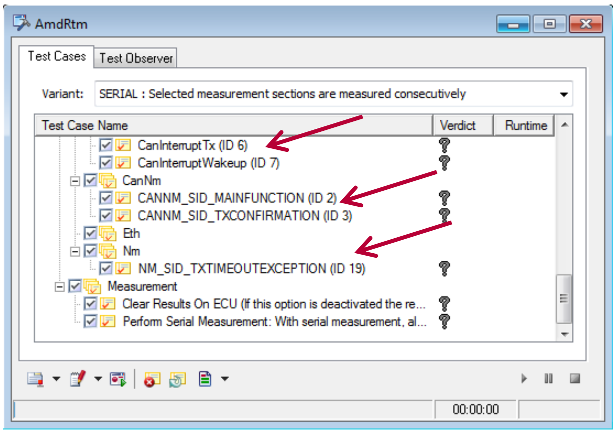


Figure 3-12 Measurement ID stands behind the name

2. Look in the RtmCan.cin

> Open the file with text editor or via CANoe.

> Look for the MP name, e.g. “Can_Init”. This name is assigned to a struct array called “RtmCan_Measurements”. The index of the array is the <MeasurementID>.

```
RtmCan_Measurements[8].ID                                    = 8;
RtmCan_Measurements[8].ByteIndex                             = 1;
RtmCan_Measurements[8].BitMask                               = 1;
RtmCan_Measurements[8].DisableInterrupts                     = RTMCAN_FALSE;
RtmCan_Measurements[8].IsRunnableMP                          = RTMCAN_FALSE;
Snprintf(RtmCan_Measurements[8].Name, elCount(RtmCan_Measurements[8].Name), "Can_Init");
Snprintf(RtmCan_Measurements[8].Group, elCount(RtmCan_Measurements[8].Group), "Can");
RtmCan_Measurements[8].AssignedToCore                        = 1;
```

Figure 3-13 Mapping between Measurement ID and MP

4 API Description

For an interfaces overview please see Figure 1-2.

4.1 Type Definitions

The types defined by the RTM are described in this chapter.

Type Name	C-Type	Description	Value Range
Rtm_MeasurementTimestampType	c-type	Data type used for measurement results	0 4294967296

Table 4-1 Type definitions

4.1.1 Rtm_DataSet

This structure contains the measurement result of a MP. This structure is only cleared if explicitly requested. (Please refer to chapter 3.7.2.2 – Clear Results On ECU)

Struct Element Name	C-Type	Description	Value Range
Time	c-type	Accumulated runtime of this MP.	0 4294967295
Count	c-type	Contains the count of how often Rtm_Start(A) – Rtm_Stop(A) was called during measurement	0 4294967295
Max	c-type	Absolute maximum runtime of this MP.	0 4294967295
Min	c-type	Absolute minimum runtime of this MP.	0 4294967295

Table 4-2 Rtm_DataSet

4.1.2 Rtm_ItemType

Defines the requested measurement result.

Enumeration	C-Type	Description	Value
RTM_ITEM_CPU_LOAD_AVERAGE	c-type	State of average CPU load. N/A for runtime MPs.	0
RTM_ITEM_CPU_LOAD_CURRENT	c-type	State of current CPU load. N/A for runtime MPs.	1
RTM_ITEM_MIN	c-type	State of minimum CPU load in percent for CPU load MPs. Or minimum runtime in microseconds for runtime MPs.	2
RTM_ITEM_MAX	c-type	State of maximum CPU load in percent for CPU load MPs. Or maximum runtime in microseconds for runtime MPs.	3

Enumeration	C-Type	Description	Value
RTM_ITEM_RUNTIME_AVERAGE	c-type	State of average runtime in microseconds. N/A for CPU load MPs.	4
RTM_ITEM_RUNTIME_OVERALL	c-type	State of overall runtime in microseconds. N/A for CPU load MPs.	5
RTM_ITEM_RELATIVE_MAX	c-type	State of maximum value compared to optionally configured /RtmTargetRuntime in percent. For CPU load MPs, this can be set e.g. to 80% of the Rtm main function cycle time and in combination with /RtmThresholdCallback to get notified if a wanted maximum of CPU load is exceeded.	6
RTM_ITEM_RELATIVE_REMAINING	c-type	State of relative remaining runtime. Calculated by 100% - RTM_ITEM_RELATIVE_MAX. Requires the optionally configured /RtmTargetRuntime.	7

Table 4-3 Rtm_ItemType

4.1.3 Rtm_TimestampType

This type is used for timestamp acquisition. Its size depends on Boolean parameter `Rtm32BitTime`.

Type Name	C-Type	Description	Value Range
Rtm_TimestampType	c-type	Data type used for timestamp acquisition. If `(Rtm32BitTimer == OFF)`	0 65535
Rtm_TimestampType	c-type	Data type used for timestamp acquisition. If `(Rtm32BitTimer == ON)`	0 4294967296

Table 4-4 Rtm_TimestampType

4.1.4 Rtm_CpuLoadHistogramType

This type is used to get the results of the CPU load histogram.

Struct Element Name	C-Type	Description	Value Range
Percentiles	c-type	Array containing the histogram results over all percentiles.	0 255

Table 4-5 Rtm_CpuLoadHistogramType

4.1.5 Rtm_TaskResponseTimeHistogramType

This type is used to get the results of the task response time histogram.

Struct Element Name	C-Type	Description	Value Range
MaxRuntimeInUs	c-type	Maximum runtime of this task in micro seconds.	0
			4294967295
Percentiles	c-type	Array containing the histogram results over all percentiles.	0
			4294967295

Table 4-6 Rtm_TaskResponseTimeHistogramType

4.1.6 Rtm_TaskStackUsageInfoType

This type is used to get the results of the task stack usage.

Struct Element Name	C-Type	Description	Value Range
MaxStackUsage	c-type	The maximum stack usage required during runtime of the task.	0
			4294967295
TaskStackSize	c-type	The maximum available stack size of the task.	0
			4294967295

Table 4-7 Rtm_TaskStackUsageType

4.1.7 Rtm_MpSettingType

Defines the requested MP setting for next measurement.

Enumeration	C-Type	Description	Value
RTM_MP_SETTING_DISABLE_ALL	c-type	Disables all MPs in the prepare structure Rtm_MeasurementConfig[].	0
RTM_MP_SETTING_ENABLE_ALL	c-type	Enables all MPs in the prepare structure Rtm_MeasurementConfig[].	1
RTM_MP_SETTING_DEFAULT	c-type	Enables all MPs configured as autostart (.\RtmAutostartEnabled) and disables all other MPs in the prepare structure Rtm_MeasurementConfig[].	2
RTM_MP_SETTING_DISABLE_ONE_MP	c-type	Disables one MP in the prepare structure Rtm_MeasurementConfig[]. All other MPs are unchanged.	3
RTM_MP_SETTING_ENABLE_ONE_MP	c-type	Enables one MP in the prepare structure Rtm_MeasurementConfig[]. All other MPs are unchanged.	4

Table 4-8 Rtm_MpSettingType

4.2 Services provided by RTM

4.2.1 Rtm_ConvertTicksToUs

Prototype	
`void Rtm_ConvertTicksToUs(ticks)`	
Parameter	
`ticks`	Ticks of the measurement counter
Return code	
`µs`	Parameter value converted to µs
Functional Description	
This function like macro converts counter ticks to microseconds. It may be used within alarm threshold callback functions.	
Particularities and Limitations	
> This macro is synchronous. > This macro is reentrant.	
Expected Caller Context	
> This macro can be called on interrupt and task level.	

Table 4-9 Rtm_ConvertTicksToUs

4.2.2 Rtm_GetVersionInfo

Prototype	
`void Rtm_GetVersionInfo(Std_VersionInfoType *Versioninfo)`	
Parameter	
`Versioninfo`	Pointer to where to store the version information of this module.
Return code	
`void`	N.A.
Functional Description	
Returns the version information, vendor ID and AUTOSAR module ID of the component. The versions are BCD-coded.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant.	

Expected Caller Context
> This function can be called on interrupt and task level.

Table 4-10 Rtm_GetVersionInfo

4.2.3 Rtm_Init

Prototype	
`void Rtm_Init(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function initializes RTM. CANoe controlled Measurements cannot be performed before calling this function.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant.	
Expected Caller Context	
> This function can be called on interrupt and task level.	

Table 4-11 Rtm_Init

4.2.4 Rtm_InitMemory

Prototype	
`void Rtm_InitMemory(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function initializes variables, which cannot be initialized with the startup code.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant.	

Expected Caller Context
> This function can be called on interrupt and task level.

Table 4-12 Rtm_InitMemory

4.2.5 Rtm_Shutdown

Prototype	
`Std_ReturnType Rtm_Shutdown(void)`	
Parameter	
`void`	N.A.
Return code	
`Std_ReturnType`	`E_OK`: Always indicating success.
Functional Description	
If `/MICROSAR/Rtm/RtmGeneral/RtmNvMTaskStackUsage` is set, this function updates the task stack usage results and requests to persist them. If `/MICROSAR/Rtm/RtmGeneral/RtmNvMResults` is set, this function requests to persist all Rtm measurement results (Rtm_Results[]). It must be called during ECU shutdown.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is not reentrant.	
Expected Caller Context	
> This function can be called on interrupt and task level.	

Table 4-13 Rtm_Shutdown

4.2.6 Rtm_MainFunction

Prototype	
`void Rtm_MainFunction(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function processes measurement requests from RTM's CANoe frontend.	

Particularities and Limitations
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is not reentrant.
Expected Caller Context
> This function shall be called on task level.

Table 4-14 Rtm_MainFunction

4.2.7 Rtm_Start

Prototype	
`void Rtm_Start(<MP_Name>)`	
Parameter	
`<MP_Name>`	String name of MP.
Return code	
`void`	N.A
Functional Description	
This function indicates the entering of the measurement section of the given MP.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > Must be called with valid MP name starting with “RtmConf_RtmMeasurementPoint_”. > Must be called in balance with Rtm_Stop for each MP name.	
Call context	
> This function shall be called on task or interrupt level.	

Table 4-15 Rtm_Start

4.2.8 Rtm_Stop

Prototype	
`void Rtm_Stop(<MP_Name>)`	
Parameter	
`<MP_Name>`	String name of MP.
Return code	
`void`	N.A.
Functional Description	
This function indicates the leaving of the measurement section of the given MP.	

Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > Must be called with valid MP name starting with “RtmConf_RtmMeasurementPoint_”. > Must be called in balance with Rtm_Start for each MP name.	
Call context	
> This function shall be called on task or interrupt level.	

Table 4-16 Rtm_Stop

4.2.9 Rtm_Start_CpuLoadMeasurement

Prototype	
`void Rtm_Start_CpuLoadMeasurement(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A
Functional Description	
This function starts the measurement of CPU’s overall load.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant.	
Call context	
> This function shall be called on task or interrupt level. > This function only starts the CPU load measurement if the CPU load control mode „C_API“ is active.	

Table 4-17 Rtm_Start_CpuLoadMeasurement

4.2.10 Rtm_Stop_CpuLoadMeasurement

Prototype	
`void Rtm_Stop_CpuLoadMeasurement(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function stops the measurement of CPU’s overall load.	

Particularities and Limitations
> This function is synchronous. > This function is not reentrant.
Call context
> This function shall be called on task or interrupt level. > This function only stops the CPU load measurement if the CPU load control mode „C_API“ is active.

Table 4-18 Rtm_Stop_CpuLoadMeasurement

4.2.11 Rtm_GetMeasurementItem

<table>
<tr><th colspan="2">Prototype</th></tr>
<tr><td colspan="2"><pre>Std_ReturnType Rtm_GetMeasurementItem(
  const uint32 ConfiguredMPId,
  const Rtm_ItemType ItemType,
  P2VAR(uint32, AUTOMATIC, RTM_APPL_VAR) ItemValuePtr)</pre></td></tr>
<tr><th colspan="2">Parameter</th></tr>
<tr><td><code>ConfiguredMPId</code></td><td>The configured measurement id. The same defines used for calling Rtm_Start()/Rtm_Stop() can be used to get the measurement result.
> RtmConf_RtmMeasurementPoint_<MP_Name></td></tr>
<tr><td><code>ItemType</code></td><td>Defines which result of the MP is requested:
> RTM_ITEM_CPU_LOAD_AVERAGE
> RTM_ITEM_CPU_LOAD_CURRENT
> RTM_ITEM_MIN
> RTM_ITEM_MAX
> RTM_ITEM_RUNTIME_AVERAGE
> RTM_ITEM_RUNTIME_OVERALL
> RTM_ITEM_RELATIVE_MAX
- Compares max value to optionally configured “/RtmTargetRuntime”
> RTM_ITEM_RELATIVE_REMAINING
- Returns 100% - RTM_ITEM_RELATIVE_MAX
- Returns 0% if RTM_ITEM_RELATIVE_MAX is higher than 100%</td></tr>
<tr><td><code>ItemValuePtr</code></td><td>It is a reference to the variable to which the result is written.
The meaning of the return value depends on the requested measurement result (ItemType). In case of overall CPU load, all requested items are interpreted in percent. In case of runtime, all requested items are interpreted in microseconds, except of RTM_ITEM_RELATIVE_MAX and RTM_ITEM_RELATIVE_REMAINING which are reported in percent.</td></tr>
<tr><th colspan="2">Return code</th></tr>
<tr><td><code>Std_ReturnType</code></td><td>The following return values are possible:
> E_OK: The service succeeded.
> E_NOT_OK: The service failed due to generic error (e.g. DET)
> RTM_RETVAL_MP_NOT_EXECUTED_YET: The requested MP was not executed yet.
> RTM_RETVAL_ITEM_NOT_AVAILABLE_FOR_MP: The requested ItemType is not available for the MP or invalid.
> RTM_RETVAL_MP_NOT_ACTIVE: The requested MP is not activated, therefore it will never provide results until activated in the configuration.</td></tr>
<tr><th colspan="2">Functional Description</th></tr>
<tr><td colspan="2">This function returns the current result of Rtm measurement. It can be called while measurement is active or after a measurement or before the first measurement is started.</td></tr>
</table>

Particularities and Limitations
> This function is synchronous. > This function is not reentrant.
Call context
> This function shall be called on task or interrupt level.

Table 4-19 Rtm_GetMeasurementItem

4.2.12 Rtm_PrepareMPSettings

Prototype	

`Std_ReturnType Rtm_PrepareMPSettings(`
`  Rtm_MpSettingType MpSetting,`
`  const uint32 ConfiguredMPId)`	

Parameter	
`MpSetting`	The action to be executed on Rtm_MeasurementConfig[] to prepare the required MPs for next measurement start: > RTM_MP_SETTING_DISABLE_ALL > RTM_MP_SETTING_ENABLE_ALL > RTM_MP_SETTING_DEFAULT > RTM_MP_SETTING_ENABLE_ONE_MP > RTM_MP_SETTING_DISABLE_ONE_MP
`ConfiguredMPId`	The configured measurement id. The same defines used for calling Rtm_Start()/Rtm_Stop() can be used to get the measurement result. > RtmConf_RtmMeasurementPoint_<MP_Name> The configured MP is only required for RTM_MP_SETTING_DISABLE_ONE_MP and RTM_MP_SETTING_ENABLE_ONE_MP, otherwise set to default value(0).
Return code	
`Std_ReturnType`	The following return values are possible: > E_OK: The service succeeded. > E_NOT_OK: The service failed due to generic error (e.g. DET or there is no activated MP in configuration) > RTM_RETVAL_ANY_COMMAND_ALREADY_ACTIVE: No error, there is just any command already executed. Therefore, try again later. > RTM_RETVAL_MEASUREMENT_CONFIG_INVALID: The MpSetting is invalid. > RTM_RETVAL_MP_NOT_ACTIVE: The requested MP is not activated, therefore it will never provide results until activated in the configuration.
Functional Description	
This function prepares all activated MPs for next start of measurement (triggered by Rtm_StartMeasurement). This function may be called multiple times before next measurement is started to prepare all required MPs.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > This function must only be called on BSW core (configured with . /RtmBSWCore).	

Call context
> This function shall be called on task or interrupt level.

Table 4-20 Rtm_PrepareMPSettings

4.2.13 Rtm_ClearMeasurementResults

Prototype	
`Std_ReturnType Rtm_ClearMeasurementResults(void)`	
Parameter	
`void`	N/A
Return code	
`Std_ReturnType`	The following return values are possible: > E_OK: The service succeeded. > E_NOT_OK: The service failed due to generic error (e.g. DET) > RTM_RETVAL_ANY_COMMAND_ALREADY_ACTIVE: No error, there is just any command already executed. Therefore, try again later.
Functional Description	
This function triggers clear of all MP results. The actual clear is executed in the next call of Rtm_MainFunction.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > This function must only be called on BSW core (configured with . /RtmBSWCore).	
Call context	
> This function shall be called on task or interrupt level.	

Table 4-21 Rtm_ClearMeasurementResults

4.2.14 Rtm_StartMeasurement

Prototype	
`Std_ReturnType Rtm_StartMeasurement ( Rtm_MeasurementTimestampType MeasurementDuration )`	
Parameter	
`MeasurementDuration`	Defines how many Rtm_MainFunction cycles the measurement is active. The value 0 starts an endless measurement that only stops by calling Rtm_StopMeasurement. The maximum allowed value is 0xFFFF.
Return code	
`Std_ReturnType`	The following return values are possible: > E_OK: The service succeeded. > E_NOT_OK: The service failed due to generic error (e.g. DET) > RTM_RETVAL_ANY_COMMAND_ALREADY_ACTIVE: No error, there is just any command already executed. Therefore, try again later.
Functional Description	
This function triggers the start of a parallel measurement in endless or time-limited mode. The actual measurement start is executed in the next call of Rtm_MainFunction.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > This function must only be called on BSW core (configured with . /RtmBSWCore).	
Call context	
> This function shall be called on task or interrupt level.	

Table 4-22 Rtm_StartMeasurement

4.2.15 Rtm_StopMeasurement

Prototype	
`Std_ReturnType Rtm_StopMeasurement(void)`	
Parameter	
`void`	N/A
Return code	
`Std_ReturnType`	The following return values are possible: > E_OK: The service succeeded. > E_NOT_OK: The service failed due to generic error (e.g. DET) > RTM_RETVAL_MEASUREMENT_CONFIG_INVALID: Measurement cannot be stopped (no measurement active, or measurement is not endless … only endless measurements can be explicitly stopped).
Functional Description	
This function triggers stop of measurement. He actual stop is executed in the next call of Rtm_MainFunction. Disables all MPs. The previously prepared MPs remain prepared (this means, when calling Rtm_StartMeasurement next, the same MPs are measured as before).	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > This function must only be called on BSW core (configured with ./RtmBSWCore).	
Call context	
> This function shall be called on task or interrupt level.	

Table 4-23 Rtm_StopMeasurement

4.2.16 Rtm_CheckTimerOverrun

Prototype	
`void Rtm_CheckTimerOverrun(void)`	
Parameter	
`void`	N/A
Return code	
`void`	N/A
Functional Description	
This function must be called if the feature ./RtmTimerOverrunSupport is enabled. It must be called at least twice per counter overrun to ensure that each overrun is detected.	

Particularities and Limitations
> This macro is synchronous. > This macro is reentrant.
Expected Caller Context
> This function shall be called on task level.

Table 4-24 Rtm_CheckTimerOverrun

4.2.17 Rtm_GetCpuLoadHistogram

Prototype	

`Std_ReturnType Rtm_GetCpuLoadHistogram(`
`  const uint16 CoreId,`
`  Rtm_CpuLoadHistogramType CpuLoadHistogram)`	

Parameter	
`CoreId`	The core id for which the result is requested.
`CpuLoadHistogram`	The CPU load histogram results of requested core.
Return code	
`E_OK`	The service succeeded.
`E_NOT_OK`	The service failed.
Functional Description	
This function calculates and returns the CPU load histogram results. This function is only successful if the feature is enabled, please refer to 3.5.3.1.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-25 Rtm_GetCpuLoadHistogram

4.2.18 Rtm_GetTaskResponseTimeHistogram

Prototype	

`Std_ReturnType Rtm_GetTaskResponseTimeHistogram(`
`  const uint16 TaskId,`
`  Rtm_TaskResponseTimeHistogramType TaskResponseTimeHistogram)`	

Parameter	
`TaskId`	The task id for which the result is requested.
`TaskResponseTimeHistogram`	The task response time histogram results of requested task.
Return code	
`E_OK`	The service succeeded.
`E_NOT_OK`	The service failed.
Functional Description	
This function calculates and returns the task response time histogram results. This function is only successful if the feature is enabled, please refer to 3.5.4.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-26 Rtm_GetTaskResponseTimeHistogram

4.2.19 Rtm_GetTaskStackUsage

Prototype	

`Std_ReturnType Rtm_GetTaskStackUsage(`
`  const uint16 TaskId,`
`  Rtm_TaskStackUsageInfoType TaskStackUsage)`	

Parameter	
`TaskId`	The task id for which the result is requested.
`TaskStackUsage`	The task stack usage results of requested task.
Return code	
`E_OK`	The service succeeded.
`E_NOT_OK`	The service failed.
Functional Description	
This function calculates and returns the task stack usage results. This function is only successful if the feature is enabled, please refer to 3.5.4. Note that there is no result available before the ECU is shutdown for the first time because the actual task stacks are only read from OS in the service Rtm_Shutdown.	

Particularities and Limitations
> This service is synchronous. > This service is reentrant.
Expected Caller Context
> This function shall be called on task level.

Table 4-27 Rtm_GetTaskStackUsage

4.2.20 Rtm_ClearHistogramResults

Prototype	
`Std_ReturnType Rtm_ClearHistogramResults( const uint16 CoreId, Rtm_ClearResultsType ResultsToBeCleared)`	
Parameter	
`CoreId`	The core id for which the clear results is requested.
`ResultsToBeCleared`	The results to be cleared: > RTM_ALL_HISTOGRAM_AND_TASK_STACK_RESULTS: Clears all available results. > RTM_CPU_LOAD_HISTOGRAM_RESULTS: Clears the CPU load histogram results. > RTM_TASK_RESPONSE_TIME_RESULTS: Clears the task response time histogram results > RTM_TASK_STACK_USAGE_RESULTS: Clears the task stack usage results
Return code	
`E_OK`	The service succeeded. Also returned if the feature of requested result is disabled.
`E_NOT_OK`	The service failed.
Functional Description	
This function clears the results of CPU load histogram and/or task response time histogram and/or task stack usage on the given core. Notes: 1. The task stack usage can only be cleared by the BSW core. 2. Clearing the task response time histogram will also clear the corresponding cached raw values in Rtm_Results[], which may effect other measurement reports. Therefore, it is recommended to use the histogram features not in combination with the common measurement handling (refer to section 2.1.20).	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant.	

Expected Caller Context
> This function shall be called on task level.

Table 4-28 Rtm_ClearHistogramResults

4.2.21 Rtm_TriggerReading

Prototype	
`void Rtm_TriggerReading(void)`	
Parameter	
`void`	N/A
Return code	
`void`	N/A
Functional Description	
If the RTE supports LET, this function is called by the RTE. It can be called by application as well. Calling this function means, that the endless parallel measurement is stopped. It has no effect if no measurement, a live measurement, or a time-limited measurement is currently active. If XCP is available, next time the Rtm_MainFunction() of BSW core is called, Xcp_Event() is invoked. The CAPL script triggers then the reading of all measurement results and all csv reports (report.csv, runnable_report.csv, Report_LET.csv) are generated with current date and time within report name. If the measurement handling is done via C-API (and not via XCP), it is recommended to not call this API. Instead, re-define RTE's macro Rte_LET_ReportError to call an application function. Please refer to section 2.1.22.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant.	
Expected Caller Context	
> This function shall be called on task level.	

Table 4-29 Rtm_TriggerReading

4.2.22 Rtm_OsVthActivation

Prototype	
`void Rtm_OsVthActivation(uint32 TaskId, uint16 DestCoreId, uint16 CallerCoreId)`	
Parameter	
`TaskId`	The activated task.
`DestCoreId`	The destination core on which the task will be running.
`CallerCoreId`	The caller core from which the task is started.
Return code	
`void`	N/A
Functional Description	
If the task response time histogram feature (refer to 3.5.4) is enabled, this function is generated. Starts the MP corresponding to the TaskId, if the DestCoreId is equal to CallerCoreId. It is highly recommended to call this function in context of the OS timing hook OS_VTH_ACTIVATION. For more details about OS configuration for RTM module, please refer to 3.4.1.1. The Rtm defines the macro OS_VTH_ACTIVATION if not defined before. If it is already defined, this function has to be called, otherwise the measurement will not work.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant for different tasks. > Each task must have a configured MP of `/RtmMeasurementPointType` "Task" > The task MPs must be of `/RtmMeasurementType` "GrossExecutionTime"	
Expected Caller Context	
> This function shall be called in OS context.	

Table 4-30 Rtm_OsVthActivation

4.2.23 Rtm_OsVthSetEvent

Prototype	
`void Rtm_OsVthSetEvent(uint32 TaskId, boolean StateChanged, uint16 DestCoreId, uint16 CallerCoreId)`	
Parameter	
`TaskId`	The activated task.
`StateChanged`	Indicates if the state to activated the task changed.
`DestCoreId`	The destination core on which the task will be running.
`CallerCoreId`	The caller core from which the task is started.
Return code	
`void`	N/A
Functional Description	
If the task response time histogram feature (refer to 3.5.4) is enabled, this function is generated. Starts the MP corresponding to the TaskId, if the DestCoreId is equal to CallerCoreId and StateChanged is TRUE. It is highly recommended to call this function in context of the OS timing hook OS_VTH_SETEVENT. For more details about OS configuration for RTM module, please refer to 3.4.1.1. The Rtm defines the macro OS_VTH_SETEVENT if not defined before. If it is already defined, this function has to be called, otherwise the measurement will not work.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant for different tasks. > Each task must have a configured MP of `/RtmMeasurementPointType` “Task” > The task MPs must be of `/RtmMeasurementType` “GrossExecutionTime”	
Expected Caller Context	
> This function shall be called in OS context.	

Table 4-31 Rtm_OsVthSetEvent

4.2.24 Rtm_OsVthSchedule

Prototype	
`void Rtm_OsVthSchedule(uint32 TaskId, uint16 DestCoreId, uint16 CallerCoreId)`	
Parameter	
`FromThreadId`	The thread (task or ISR2) which is stopped.
`FromThreadReason`	The reason why the FromThreadId is stopped: - OS_VTHP_TASK_TERMINATION - OS_VTHP_TASK_WAITEVENT - OS_VTHP_THREAD_PREEMPT
`ToThreadId`	The thread (task or ISR2) which is started.
`ToThreadReason`	The reason why the ToThreadId is started: - OS_VTHP_TASK_ACTIVATION - OS_VTHP_TASK_SETEVENT - OS_VTHP_THREAD_RESUME
`CallerCoreId`	The core on which the scheduling is done.
Return code	
`void`	N/A
Functional Description	
This function is generated if the task response time histogram feature (refer to 3.5.4) is enabled, or there is at least one net or flat execution time MP (refer to 2.1.8.2), or the CPU load measurement is enabled (refer to 3.5.3). Stops the task related MP if a task is terminated or it waits for an event. Starts the CPU load related idle task MP if the idle task is preempted, and stops this MP if the idle task is resumed. Handles the interruption of net and flat execution time MPs. It is highly recommended to call this function in context of the OS timing hook OS_VTH_SCHEDULE. For more details about OS configuration for RTM module, please refer to 3.4.1.1. The Rtm defines the macro OS_VTH_SCHEDULE if not defined before. If it is already defined, this function has to be called, otherwise the measurement will not work.	
Particularities and Limitations	
> This service is synchronous. > This service is reentrant for different tasks. > Each task must have a configured MP of `/RtmMeasurementPointType` “Task” > The task MPs must be of `/RtmMeasurementType` “GrossExecutionTime” > The CPU load MPs must be of `/RtmMeasurementType` “GrossExecutionTime”	
Expected Caller Context	
> This function shall be called in OS context.	

Table 4-32 Rtm_OsVthSchedule

4.3 Services used by RTM

In the following table services provided by other components, which are used by the RTM are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DET	Det_ReportError
XCP	Xcp_Event
OS	Os_GetTaskStackUsage, GetCoreId
NvM	NvM_GetErrorStatus, NvM_SetRamBlockStatus

Table 4-33 Services used by the RTM

4.4 Configurable Interfaces

4.4.1 Callback Functions

4.4.1.1 Rtm_Schedule

Prototype	
`void Rtm_Schedule( uint32 FromThreadId, uint32 ToThreadId, uint16 CoreId )`	
Parameter	
`FromThreadId`	The thread which is preempted/terminated.
`ToThreadId`	The thread which is entered (now running).
`CoreId`	The core on which the scheduling is performed.
Return code	
`void`	N.A.
Functional Description	
Preempts a thread and starts another thread. Number of ThreadIds is the sum by adding the numbers of Task Ids and Isrs Ids. If a net measurement section is active on preempted thread, this section is also preempted. If a net measurement section was preempted on the entered thread, this section is also started. When alternate method instead of OS timing hooks is used, Rtm_Schedule() is used as follows in pre/post hooks which are avaiable, Pre/Post Task Hooks: • A variable shall be defined to store the information of the task Id in OS Enter Task Hooks, which provides information about the prempted thread. • The function call shall be added by the integrator to the OS Leave Task Hooks in order to measure RTM net runtimes without interruption times of higher priority task. This function should be provided with two parameters, one containing information about the preempted thread from the OS Enter Task Hooks and the other containing information about the current thread from the OS Leave Task Hook. Pre/Post ISR Hooks: • A variable shall be defined to store the information of the thread Id in OS Enter CAT2 ISR Hooks, which provides information about the prempted thread. • The function call shall be added by the integrator to the OS Leave CAT2 ISR Hooks in order to measure RTM net runtimes. This function should be provided with two parameters, one containing information about the preempted thread from the OS Enter ISR Hook and the other containing information about the current thread from the OS Leave ISR Hook.	
Particularities and Limitations	
> This function is only available if at least one MP type `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType` is set to `NetExecutionTime` or `FlatExecutionTime`.	
Call context	
> Interrupt or task context > Only during measurements	

Table 4-34 Rtm_Schedule

4.4.2 Callout Functions

At its configurable interfaces the RTM defines callout functions. The declarations of the callout functions are provided by RTM. It is the integrator's task to provide the corresponding function definitions. The definitions of the callouts can be adjusted to the system's needs. The RTM callout function declarations are described in the following table:

4.4.2.1 Rtm_<Measurement Name>_ThresholdCbk

Prototype	
`void Rtm_<Measurement Name>_ThresholdCbk(Rtm_MeasurementTimestampType runtime)`	
Parameter	
`runtime`	Current runtime of the associated MP.
Return code	
`void`	N/A
Functional Description	
This function is called after the runtime of the associated MP has exceeded the configured threshold.	
Particularities and Limitations	
> This function has to be implemented by the user	
Call context	
> interrupt or task context > Only during measurements	

Table 4-35 Threshold Callback

4.4.2.2 RTM_GET_TIME_MEASUREMENT_FCT

Prototype	
`Rtm_TimestampType RTM_GET_TIME_MEASUREMENT_FCT(void)`	
Parameter	
`void`	N/A
Return code	
`Rtm_TimestampType`	The current timer value.
Functional Description	
This macro is a place holder for a callout function, implemented by application. It is called to get the current timer value. The name replacing this macro should be used to implement the function. This name has to be specified in DaVinci Configurator 5 /MICROSAR/Rtm/RtmGeneral/RtmGetMeasurementTimestampFct. Within this function, a timer value has to be requested and returned. As timer a GPT channel can be used.	

Particularities and Limitations
> This function has to be implemented by the user
Call context
> interrupt or task context > Only during measurements

Table 4-36 RTM_GET_TIME_MEASUREMENT_FCT

5 Configuration

5.1 Configuration Variants

The RTM supports the configuration variants

- `VARIANT-PRE-COMPILE`

The configuration classes of the RTM parameters depend on the supported configuration variants. For their definitions please see the Rtm_bswmd.arxml file.

6 Limitations

6.1 Runtime impact

To minimize the impact of RTM's own code to the runtime behavior of the ECU, all MPs are "inactive" by default. In this state, RTM does require very little CPU-load but also does not record measurement data (exceptions are auto start measurements which are active by default. (Chapter: 2.1.5)).

6.2 Measurement success

Performing measurement with RTM's frontend means activating one or more MPs. Now measurement data is collected if the corresponding code section is executed. However, if the code section was not executed during the measurement, RTM cannot provide any data. In this case, the measurement has to be repeated. This problem can be minimized by selecting long measurement duration.

6.3 Inter-Task Measurement

The feature to start a runtime measurement in one task and stop it within another task is not supported. The measurement behavior is shown in the following figure.

Use case:

Func_1 of Task_1 executes an algorithm. Afterwards, Task_2 is activated. Because Task_2 has the higher priority, Task_1 is interrupted and Task_2 is running. Task_2 executes Func_2 that accesses the results of Func_1. The runtime between executing an algorithm in one task and using its results in another task could be measured with this feature.

This feature is not supported because it always has to be granted that the measurement was already started before it can be stopped.

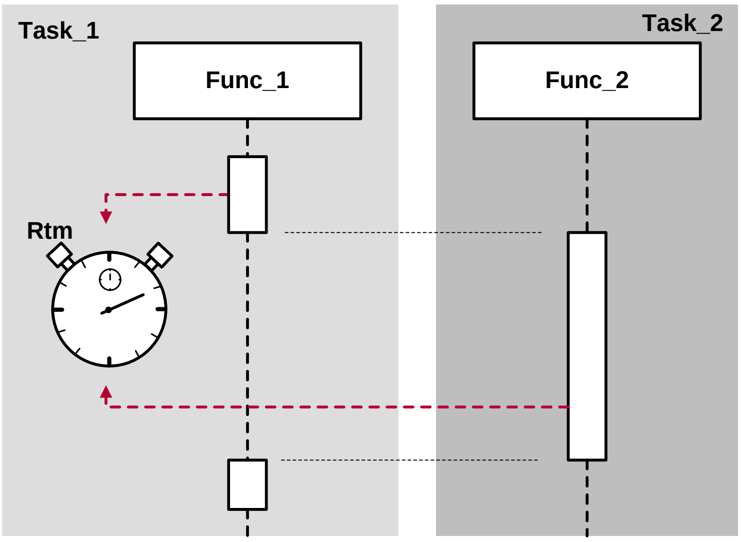


Figure 6-1 Inter-Task Measurement

6.4 Auto start Measurement

The results of auto start measurement are reported if the CANoe flag ‘Clear Results On Ecu’ is not set.

6.5 Flat execution time MPs

The flat MPs shall be started and stopped in the correct order, otherwise the DET RTM_E_UNBALANCED is reported. The latest started MP must be the first MP to be stopped.

Example
Good example (MP1 and MP2 are of type flat):

1. Rtm_Start(<MP1>);
2. Rtm_Start(<MP2>);
3. Rtm_Stop(<MP2>);
4. Rtm_Stop(<MP1>);

Example
Bad example (MP1 and MP2 are of type flat):

1. Rtm_Start(<MP1>);
2. Rtm_Start(<MP2>);
3. Rtm_Stop(<MP1>); // DET RTM_E_UNBALANCED is reported!
4. Rtm_Stop(<MP2>);

The above limitation is only applicable for Flat execution time MPs and not for gross and net execution time MPs.

6.6 Report for Timing-Architects™

The report for Timing-Architects™ does not contain the information which SWC the runnables are assigned to.

Additionally, RTM does not generate the runnable MPs in the RTE hooks. Instead, RTM provides the groovy script ‘CreateRunnableMeasurementPoints’ which can be executed in DaVinci Configurator. Please refer to section 2.1.14.

6.7 Limitations for multi core

> The feature `RtmTimerOverrunSupport` is not supported

> Live measurement is not supported.

- The time source is identical for all cores, therefore the timer request must be reentrant. Thus the use of GPT APIs (Gpt_GetTimeElapsed and Gpt_GetTimeRemaining) is not allowed.
- Does not support a single core configuration of RTM (only one RtmCoreDeifintion), if there are multiple cores configured in Os.

6.8 Vector OS

All RTM features requiring an OS, require the Vector OS. The affected features are:

- CPU load measurement
- CPU load histogram
- Task response time histogram
- Task stack usage
- Net- and flat execution time

7 Rtm on CANoe (RtmCan)

The RtmCan provides services to control the Rtm via XCP in CANoe without the also provided Test Module. These services are implemented in the file RtmCan.cin. This file can be included in any CAPL file.

Caution
The Rtm Test Module also uses the RtmCan.cin file for Rtm control. If an application file is used, the Rtm Test Module must not be used.

Hint: It is sufficient to deactivate the Test Module in CANoe. It is not required to remove it from CANoe configuration.

Note
The Rtm application can also be implemented in a **.Net** project, but there are three things to consider:

1. A *.cin file cannot be directly referenced by a .Net application. Therefore a *.can must be created that includes RtmCan.cin.
2. The .Net application cannot resolve enumerations of CAPL files. Thus, the services containing an enumeration as return value or parameter cannot be called from .Net.
   Therefore, expand the CAPL file *.can with wrapper services converting enumerations to unsigned integer (dword in CAPL).
   Interesting enums:
   - `RtmCan_MeasurementModeType`
   - `RtmCan_MeasurementStateType`
   - `RtmCan_StateType`
   - `RtmCan_ReturnValueType`
   - `RtmCan_ReportVariantType`
   - `RtmCan_DebugLevelType`
3. The .Net application cannot resolve structures of CAPL. Therefore write a serialization function in CAPL for the structure `RtmCan_ResultType`. Transform the structure members to a string. Copy its content to a string system variable and add a function within the .Net application that is triggered after this system variable was changed. Within .Net the string must be de-serialized.
   This is only required if measurement results have to be requested at runtime.

7.1 RtmCan Features

The supported features of RtmCan are listed in the following table.

Supported Features
Runtime and overall CPU load measurement.
Starting measurement in serial, parallel and live measurement mode.
Endless measurement requiring explicit stop request.
Time limited measurement.
Clear all data on ECU and RtmCan or only clear all measurement results on ECU and RtmCan.
Generation of reports containing all measurement results or only runnable MPs.
Runtime configuration of MPs. Single, multiple and all MPs can be de-/activation at once.
Result requesting at runtime.
En-/Disabling Debug output at runtime.

Table 7-1 Supported Features of RtmCan

7.1.1 Available measurement modes

In the following table shows all available measurement modes in combination with possible measurement duration. The combination of measurement mode and duration is only “Available” if the “Condition” is fulfilled.

Measurement Mode	Measurement duration	Condition	Available	Expected return value of RtmCan_StartMeasurement
Serial	Time Limited (>0s)	-	x	RTMCAN_E_OK
	Endless (=0s)	-	-	RTMCAN_E_NOT_SUPPORTED
Parallel	Time Limited (>0s)	-	x	RTMCAN_E_OK
	Endless (=0s)	32 Bit Timer	x	RTMCAN_E_OK
Live	Time Limited (>0s)	Single Core	x	RTMCAN_E_OK
	Endless (=0s)	Single Core and 32 Bit Timer	x	RTMCAN_E_OK

Table 7-2 Available Measurement Modes

7.2 RtmCan states

The states of RtmCan are shown in Figure 7-1. The RtmCan is automatically initialized. After initialization, the RtmCan is in state `RTMCAN_STATE_WAITFORACTION`. Within this state almost all services are available.

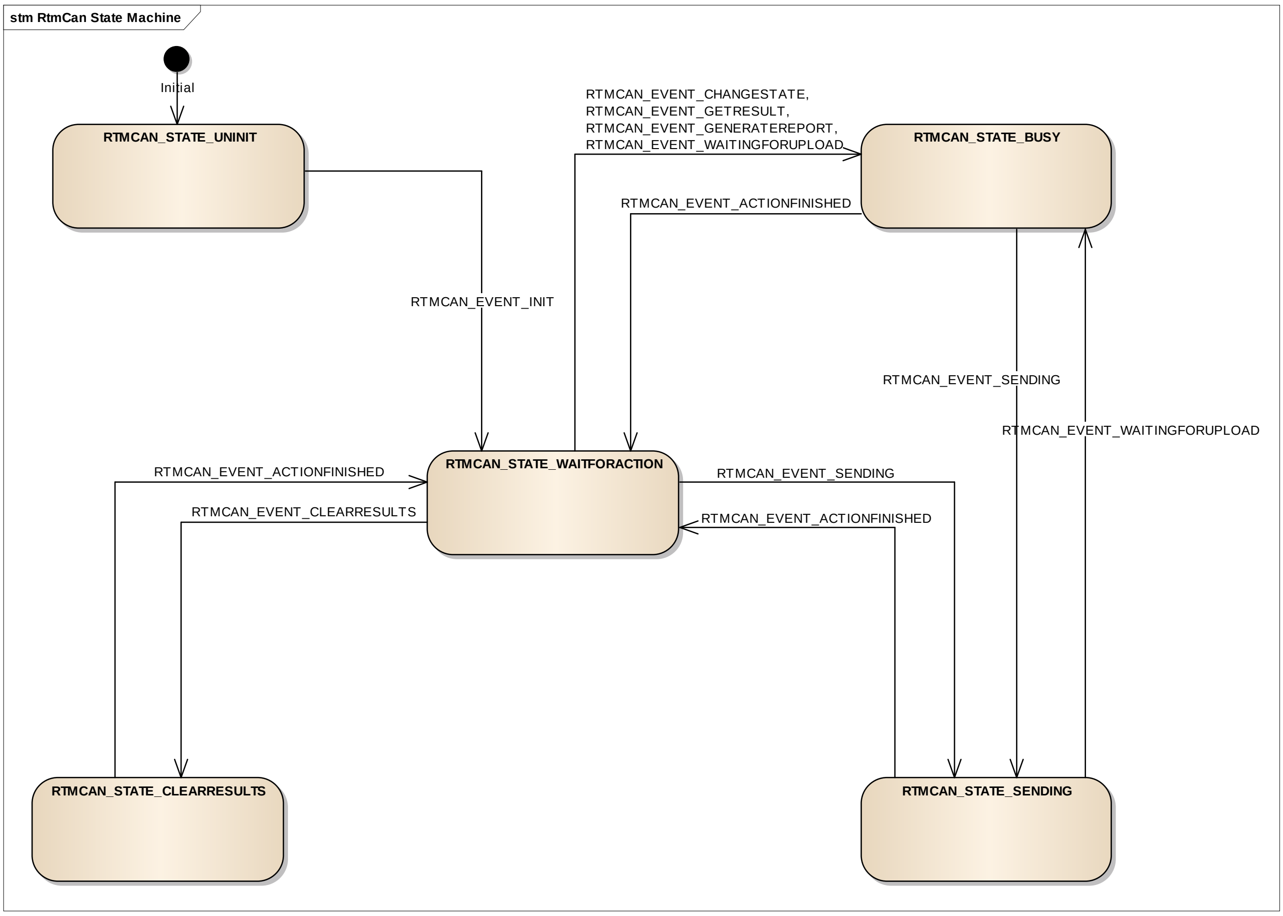


Figure 7-1 RtmCan state machine

7.2.1 Available actions within the states

The following actions are always available:

- Reset the RtmCan state
  - `RtmCan_ResetRtmCanState`
- Request the RtmCan state
  - `RtmCan_GetRtmCanState`
- Set of debug level
  - `RtmCan_SetDebugLevel`

Within the state `RTMCAN_STATE_WAITFORACTION` the following actions are available:

- Start a measurement
  - `RtmCan_StartMeasurement`
- Clear results
  - `RtmCan_ClearResults`

  - `RtmCan_ClearAll`
- Generate report
  - `RtmCan_GenerateReport`
- Request results of one MP
  - `RtmCan_GetMPResultByName`
  - `RtmCan_GetMPResultByID`
- Change state of MPs
  - `RtmCan_SetMPStateAll`
  - `RtmCan_SetMPStateGroup`
  - `RtmCan_SetMPState`
  - `RtmCan_SetMPStateByID`

The following action is only available in state `RTMCAN_STATE_MEASURING` and if the measurement duration is unlimited:

- Stop the measurement
  - `RtmCan_StopMeasurement`

7.3 Architecture of RtmCan

The file RtmCan.cin provides services to control Rtm's measurement. The RtmCan.cin file can be included by the RtmCan user. The RtmCan user can be the test feature set Rtm_Canoe.xml/.can, any .net file or any CAPL file. The file structure is shown in the following figure.

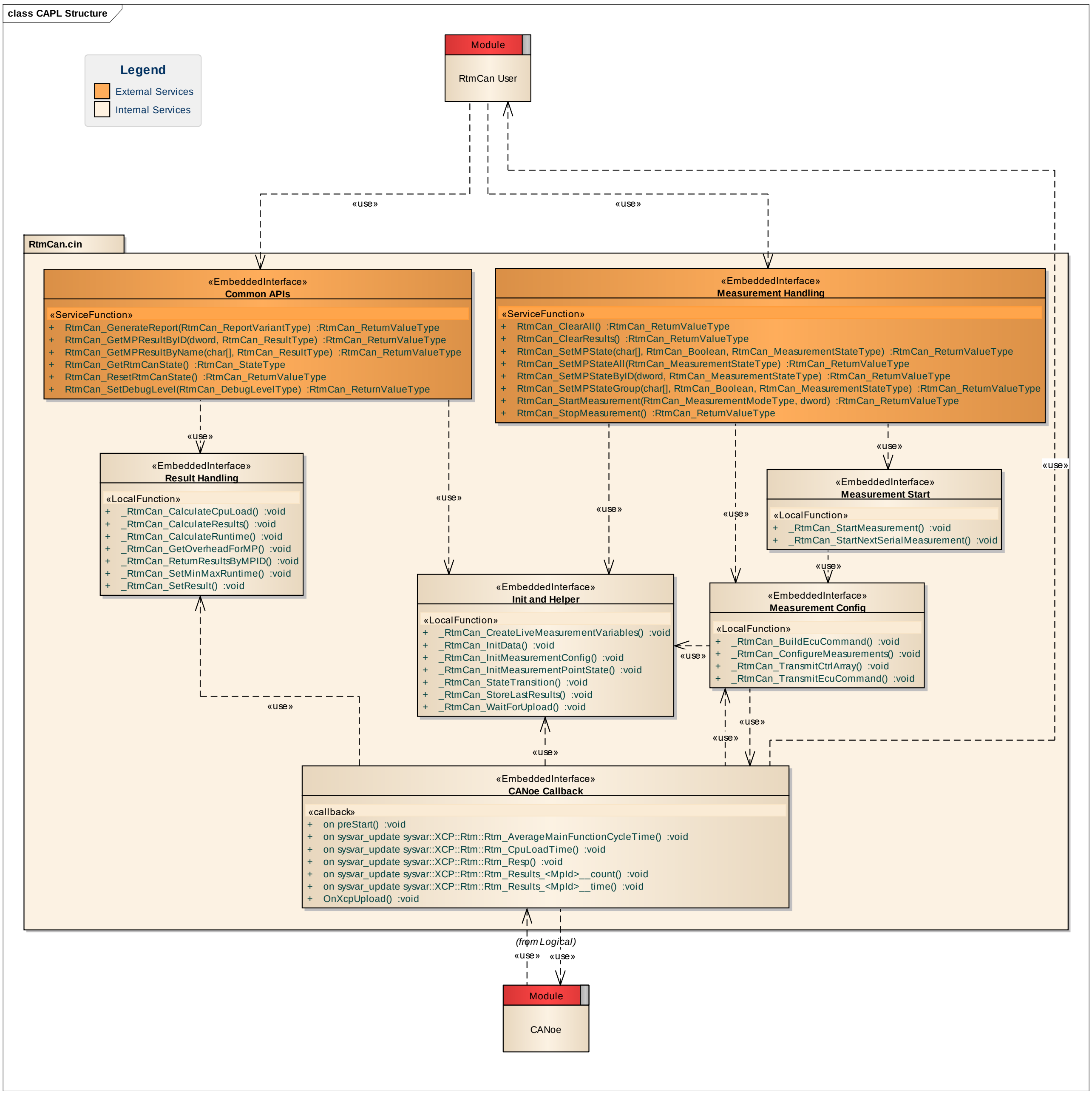


Figure 7-2 RtmCan.cin architecture

The detailed description of RtmCan's external APIs can be found in chapter 7.5.

7.4 Type Definitions

The types defined by the RtmCan are described in this chapter.

Type Name	C-Type	Description	Value Range
RtmCan_MeasurementCommandType	enum	Command type.	RTMCAN_CMD_NULL
			RTMCAN_CMD_SERIAL

Type Name	C-Type	Description	Value Range
			RTMCAN_CMD_PARALLEL RTMCAN_CMD_LIVE RTMCAN_CMD_STOP RTMCAN_CMD_CLEAR
RtmCan_MeasurementModeType	enum	The measurement modes.	RTMCAN_MODE_SERIAL RTMCAN_MODE_PARALLEL RTMCAN_MODE_LIVE
RtmCan_MeasurementPointType	enum	The measurement point type.	RTMCAN_MP_TYPE_GROSS_EXECUTIONTIME RTMCAN_MP_TYPE_FLAT_EXECUTIONTIME RTMCAN_MP_TYPE_NET_EXECUTIONTIME
RtmCan_MeasurementStateType	enum	State of the MPs.	RTMCAN_MP_STATE_INACTIVE RTMCAN_MP_STATE_ACTIVE
RtmCan_Boolean	enum	Result of conditions.	RTMCAN_FALSE RTMCAN_TRUE
RtmCan_StateType	enum	State of the RtmCan.	RTMCAN_STATE_UNINIT RTMCAN_STATE_WAITFORACTION RTMCAN_STATE_BUSY RTMCAN_STATE_MEASURING RTMCAN_STATE_CLEARRESULTS
RtmCan_EventType	enum	Events occuring to trigger state changes.	RTMCAN_EVENT_INIT RTMCAN_EVENT_ACTIONFINISHED RTMCAN_EVENT_MEASURING RTMCAN_EVENT_CLEARRESULTS RTMCAN_EVENT_WAITINGFORUPLOAD RTMCAN_EVENT_CHANGESTATE RTMCAN_EVENT_GETRESULT RTMCAN_EVENT_GENERATEREPORT
RtmCan_ReturnValueType	enum	All return values.	RTMCAN_E_OK RTMCAN_E_MEASUREMENT_NOT_STOPPABLE RTMCAN_E_INTERFACE_IS_BUSY RTMCAN_E_NO_MATCH_FOUND RTMCAN_E_INVALID_EVENT RTMCAN_E_INVALID_STATE RTMCAN_E_NOT_AVAILABLE_IN_MULTICORE_SYSTEM RTMCAN_E_NOT_AVAILABLE_WITH_16BIT_COUNTER RTMCAN_E_NOT_SUPPORTED

Type Name	C-Type	Description	Value Range
RtmCan_ReportVariantType	enum	The report variants.	RTMCAN_REPORT_ALL
			RTMCAN_REPORT_OVERALL_CSV
			RTMCAN_REPORT_RUNNABLE_CSV
RtmCan_LetErrorType	enum	The LET errors.	RTMCAN_LET_ERROR_OK
			RTMCAN_LET_ERROR_INTERVAL_EXCEEDED
			RTMCAN_LET_ERROR_UNEXPECTED_RELEASE
			RTMCAN_LET_ERROR_UNEXPECTED_TERMINATE
			RTMCAN_LET_ERROR_UNEXPECTED_RUNNABLE_EXECUTION
RtmCan_DebugLevelType	enum	The debug level.	RTMCAN_DEBUGLEVEL_OFF
			RTMCAN_DEBUGLEVEL_ERROR
			RTMCAN_DEBUGLEVEL_WARNING
			RTMCAN_DEBUGLEVEL_INFO

Table 7-3 Types defined by RtmCan

RtmCan_MeasurementType

Struct Element Name	C-Type	Description	Value Range
ID	word	The measurement ID.	0
			65535
ByteIndex	word	Index of array to calculate mask ID.	0
			65535
BitMask	char	Index of array element to calculate mask ID.	-128
			127
Name	char array	The name of the MP.	Max. 128 bytes.
Group	char array	The group name of the MP.	Max. 50 bytes.
LetErrorVar	char array	The LET error variable name provided by RTE.	Max. 256 bytes.
DisableInterrupts	enum RtmCan_Boolean	Defines if flag RtmDisableInterrupts is set for this MP.	RTMCAN_FALSE
			RTMCAN_TRUE
IsRunnableMP	enum RtmCan_Boolean	Defines if MP is part of Runnable group.	RTMCAN_FALSE
			RTMCAN_TRUE
IsLetMP			RTMCAN_FALSE

Struct Element Name	C-Type	Description	Value Range
	enum RtmCan_Boolean	Defines if MP is part of LET group.	RTMCAN_TRUE
LetError	enum RtmCan_LetErrorType	Defines which LET error occurred for this MP. Only applicable for LET MPs.	RTMCAN_LET_ERROR_OK
			RTMCAN_LET_ERROR_INTERVAL_EXCEEDED
			RTMCAN_LET_ERROR_UNEXPECTED_RELEASE
			RTMCAN_LET_ERROR_UNEXPECTED_TERMINATE
			RTMCAN_LET_ERROR_UNEXPECTED_RUNNABLE_EXECUTION
AssignedToCore	dword	Defines to which core the MP is assigned.	0
			RTMCAN_NUMBER_OF_CORES
MeasurementTime_current	dword	The leatest measurement result: measured ticks.	0
			4294967295
MeasurementTime_last	dword	The last measurement result: measured ticks.	0
			4294967295
MeasurementCount_current	dword	The leatest measurement result: number of start/stops.	0
			4294967295
MeasurementCount_last	dword	The leatest measurement result: number of start/stops.	0
			4294967295
MeasurementMin	dword	Minimum ticks of one start/stop.	0
			4294967295
MeasurementMax	dword	Maximum ticks of one start/stop.	0
			4294967295
TargetRuntime	dword	Target runtime of MP in us.	0
			1000000
MeasurementRuntime	double	Resulting runtime of this MP in µs.	-1.7E +/- 308
			1.7E +/- 308
MeasurementCpuLoad_PerCent	double	Resulting CPU load of this MP in %.	-1.7E +/- 308
			1.7E +/- 308
MeasurementCycleTimeCorrection	int	Measurement correction value.	-32768
			+32767

Table 7-4 RtmCan_MeasurementType

RtmCan_ResultType

Struct Element Name	C-Type	Description	Value Range
RtmCan_Result_ID	dword	The measurement ID.	0
			4294967295
RtmCan_Result_Name	char array	The name of the MP.	Max. 50 bytes.
RtmCan_Result_Group	char array	The group name of the MP.	Max. 50 bytes.
RtmCan_Result_MP_Type	enum RtmCan_MeasurementPointType	The MP type.	RTMCAN_MP_TYPE_GROSS_EXECUTIONTIME
			RTMCAN_MP_TYPE_FLAT_EXECUTIONTIME
			RTMCAN_MP_TYPE_NET_EXECUTIONTIME
RtmCan_Result_MinRuntime_us	double	Minimum runtime of this MP in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_MaxRuntime_us	double	Maximum runtime of this MP in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_RelativeMax_percent	double	Relative maximum (compared to configured TargetRuntime) of this MP in percent. If configured TargetRuntime is 0, this is always 0.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_RelativeRemaining_percent	double	Relative remaining (compared to configured TargetRuntime) of this MP in percent. If configured TargetRuntime is 0, this is always 0.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_AverageRuntime_us	double	Average runtime of this MP in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_NumberOfExecution	dword	Number of executions of this MP.	0
			4294967295
RtmCan_Result_CpuLoad	double	Average CPU load of this MP.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_AssignedToCore	dword	The core the MP is assigned to. If it is not assigned, this value is set to RTMCAN_NUMBER_OF_CORES.	0
			4294967295
	qword		0

Struct Element Name	C-Type	Description	Value Range
RtmCan_Result_MeasurementDuration_us		The overall measurement duration while this MP was active.	18446744073709551615

Table 7-5 RtmCan_ResultType

7.5 Services provided by RtmCan (CAPL)

All following APIs must not be called before “on prestart” function of RtmCan.cin was called.

7.5.1 RtmCan_GetRtmCanState

Prototype	
`enum RtmCan_StateType RtmCan_GetRtmCanState(void)`	
Parameter	
`void`	N/A.
Return code	
`enum RtmCan_StateType`	The current state of the RtmCan.
Functional Description	
This function returns the current state of RtmCan and can be called independent of the RtmCan state.	
Particularities and Limitations	
> This function is synchronous.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-6 RtmCan_GetRtmCanState

7.5.2 RtmCan_GenerateReport

Prototype	
`enum RtmCan_ReturnValueType RtmCan_GenerateReport(enum RtmCan_ReportVariantType reportVariant)`	
Parameter	
`reportVariant`	Defines the type of report to be generated. Possible values: - RTMCAN_REPORT_ALL: All available reports are generated. - RTMCAN_REPORT_OVERALL_CSV: The usual report containing all measurement results. In csv format. - RTMCAN_REPORT_RUNNABLE_CSV: Report containing all measurement results of runnable MPs. In csv format. - RTMCAN_REPORT_LET_CSV: Report containing all measurement results of LET MPs. In csv format.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The requested report was generated successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the generation request was omitted. - RTMCAN_E_INVALID_EVENT: The resource lock of RtmCan failed, thus the generation request was omitted. - RTMCAN_E_INVALID_STATE: The resource lock of RtmCan failed, thus the generation request was omitted.
Functional Description	
This function generates the requested report with the latest measurement results. If no measurement result is available the generated reports are empty respectively filled with zeros.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-7 RtmCan_GenerateReport

7.5.3 RtmCan_GetMPResultByName

Prototype	
`enum RtmCan_ReturnValueType RtmCan_GetMPResultByName(char measPointName[], struct RtmCan_ResultType result)`	
Parameter	
`measPointName`	The name of the MP for which the measurement results have to be returned.
`result`	The structure containing all measurement results of the requested MP.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The requested result was copied successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the result copying was omitted. - RTMCAN_E_NO_MATCH_FOUND: The requested MP name does not exist in the current configuration.
Functional Description	
This function returns a structure containing alle measurement results of the requested MP.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-8 RtmCan_GetMPResultByName

7.5.4 RtmCan_GetMPResultByID

Prototype	
`enum RtmCan_ReturnValueType RtmCan_GetMPResultByID(dword measPointID[], struct RtmCan_ResultType result)`	
Parameter	
`measPointID`	The identifier of the MP for which the measurement results have to be returned.
`result`	The structure containing all measurement results of the requested MP.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The requested result was copied successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the result copying was omitted. - RTMCAN_E_NO_MATCH_FOUND: The requested MP name does not exist in the current configuration.
Functional Description	
This function returns a structure containing alle measurement results of the requested MP.	

Particularities and Limitations
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`. > The requested measurement ID must be in range: `RTMCAN_NUMBER_OF_OVERHEAD_MPS <= measPointID <= RTMCAN_MEASUREMENTS_COUNT`.
Call context
> This function shall be called on task or interrupt level.

Table 7-9 RtmCan_GetMPResultByID

7.5.5 RtmCan_ResetRtmCanState

Prototype	
`enum RtmCan_ReturnValueType RtmCan_ResetRtmCanState(void)`	
Parameter	
`void`	N/A.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The RtmCan state is successfully reseted to state RTMCAN_STATE_WAITFORACTION.
Functional Description	
This function can be used to break a deadlock in RtmCan. If a stoppable measurement is currently executed, a stop command is send to ECU. The RtmCan state is set to RTMCAN_STATE_WAITFORACTION.	
Particularities and Limitations	
> This function is synchronous. > This function does not clear the measurement results on ECU.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-10 RtmCan_ResetRtmCanState

7.5.6 RtmCan_SetDebugLevel

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetDebugLevel(enum RtmCan_DebugLevelType debugLevel)`	
Parameter	
`debugLevel`	The level of displayed debug informations. Possible value: - RTMCAN_DEBUGLEVEL_OFF: No debug information is displayed. - RTMCAN_DEBUGLEVEL_ERROR: Only ciritcal debug information is displayed. - RTMCAN_DEBUGLEVEL_WARNING: Ciritcal and important hint debug information is displayed. - RTMCAN_DEBUGLEVEL_INFO: All debug information is displayed.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: Always returned.
Functional Description	
This function sets the system variable Rtm_DebugLevel to new value. If the system variable does not exist, it is created and the specified value is used as initial value.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-11 RtmCan_SetDebugLevel

7.5.7 RtmCan_ClearResults

Prototype	
`enum RtmCan_ReturnValueType RtmCan_ClearResults(void)`	
Parameter	
`Void`	N/A.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The clear request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the clear request was omitted. - RTMCAN_E_INVALID_EVENT: The resource lock of RtmCan failed, thus the generation request was omitted. - RTMCAN_E_INVALID_STATE: The resource lock of RtmCan failed, thus the generation request was omitted.
Functional Description	
This function sends a clear command to ECU. All measurement results are removed in ECU and RtmCan. If this function is called, all activated MPs remain activated.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-12 RtmCan_ClearResults

7.5.8 RtmCan_ClearAll

Prototype	
`enum RtmCan_ReturnValueType RtmCan_ClearAll(void)`	
Parameter	
`Void`	N/A.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The clear request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the clear request was omitted. - RTMCAN_E_INVALID_EVENT: The resource lock of RtmCan failed, thus the generation request was omitted. - RTMCAN_E_INVALID_STATE: The resource lock of RtmCan failed, thus the generation request was omitted.
Functional Description	
This function sends a clear command to ECU. All measurement results are removed in ECU and RtmCan. If this function is called, all activated MPs are set to inactive.	

Particularities and Limitations
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.
Call context
> This function shall be called on task or interrupt level.

Table 7-13 RtmCan_ClearAll

7.5.9 RtmCan_StartMeasurement

Prototype	
`enum RtmCan_ReturnValueType RtmCan_StartMeasurement(enum RtmCan_MeasurementModeType measMode, dword timeToMeas)`	
Parameter	
`measMode`	Defines the measurement mode in which the measurement has to be started. Possible values: - RTMCAN_MODE_SERIAL: Executes measurement once per active MP until last MP was executed. Results are updated after each MP's measurement end. - RTMCAN_MODE_PARALLEL: Executes measurement for all active MPs simultaneously. Results are updated after measurement end. - RTMCAN_MODE_LIVE: Executes measurement for all active MPs simultaneously. Results are updated for each call to Rtm_MainFunction.
`timeToMeas`	Defines the measurement duration. Possible values: - 0: The measurement is started in endless mode. The measurement must be stopped by call to `RtmCan_StopMeasurement()`. - > 0 [ms]: The measurement is started for the specified number of milli seconds.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The start requested was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the clear request was omitted. - RTMCAN_E_INVALID_EVENT: The resource lock of RtmCan failed, thus the generation request was omitted. - RTMCAN_E_INVALID_STATE: The resource lock of RtmCan failed, thus the generation request was omitted. - RTMCAN_E_NOT_AVAILABLE_WITH_16BIT_COUNTER: The endless measurement mode is not available if 16 bit counter is used (must be 32 bit). - RTMCAN_E_NOT_AVAILABLE_IN_MULTICORE_SYSTEM: Live measurement is not available in multicore systems. - RTMCAN_E_NOT_SUPPORTED: Endless measurement in serial mode is not supported.
Functional Description	
This function starts a new measurement in the specified measurement mode and for the specified measurement duration.	

Particularities and Limitations
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`. > If multicore support is enabled, live measurement is not supported. > If 16 bit counter is used, endless measurement is not supported. > Endless measurement is only available for parallel and live measurement.
Call context
> This function shall be called on task or interrupt level.

Table 7-14 RtmCan_StartMeasurement

7.5.10 RtmCan_StopMeasurement

Prototype	
`enum RtmCan_ReturnValueType RtmCan_StopMeasurement(void)`	
Parameter	
`void`	N/A.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The stop request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The stop request failed because no measurement is running. - RTMCAN_E_MEASUREMENT_NOT_STOPPABLE: The running measurement is time limited and therefore not stoppable by this API.
Functional Description	
This function stops a running endless measurement.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_MEASURING`. > An endless measurement must be active.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-15 RtmCan_StopMeasurement

7.5.11 RtmCan_SetMPStateAll

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPStateAll(enum RtmCan_MeasurementStateType state)`	
Parameter	
`state`	The new state of all MPs. Possible values: - RTMCAN_MP_STATE_INACTIVE: All MPs are deactivated. - RTMCAN_MP_STATE_ACTIVE: All MPs are activated.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The stet state request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the set state request was omitted.
Functional Description	
This function sets the state (active/inactive) of all MPs.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-16 RtmCan_SetMPStateAll

7.5.12 RtmCan_SetMPStateGroup

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPStateGroup(char measPointGroupName[], enum RtmCan_Boolean equal, enum RtmCan_MeasurementStateType state)`	
Parameter	
`measPointGroupName`	Defines the measurement group name. All MPs that are part of this group change their state to new value.
`equal`	Possible values: - RTMCAN_FALSE: Only change states of MPs within the measurement group `measPointGroupName`. - RTMCAN_TRUE: Change states of all MPs that are in a group containing the string of `measPointGroupName`.
`state`	The new state of all matching MPs. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching MPs are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching MPs are activated.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The stet state request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the set state request was omitted. - RTMCAN_E_NO_MATCH_FOUND: No MP is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of all MPs of the specified measurement group.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-17 RtmCan_SetMPStateGroup

7.5.13 RtmCan_SetMPState

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPState(char measPointName[], enum RtmCan_Boolean equal, enum RtmCan_MeasurementStateType state)`	
Parameter	
`measPointName`	Defines the measurement group name. All MPs that are part of this group change their state to new value.
`equal`	Possible values: - RTMCAN_FALSE: Only change states of MPs within the measurement group `measPointName`. - RTMCAN_TRUE: Change states of all MPs that are in a group containing the string of `measPointName`.
`state`	The new state of all matching MPs. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching MPs are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching MPs are activated.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The stet state request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the set state request was omitted. - RTMCAN_E_NO_MATCH_FOUND: No MP is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of one MP with the specified name.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-18 RtmCan_SetMPState

7.5.14 RtmCan_SetMPStateByID

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPStateByID(dword measPointID, enum RtmCan_MeasurementStateType state)`	
Parameter	
`measPointName`	Defines the measurement group name. All MPs that are part of this group change their state to new value.
`state`	The new state of all matching MPs. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching MPs are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching MPs are activated.
Return code	
`enum RtmCan_ReturnValueType`	Possible values: - RTMCAN_E_OK: The stet state request was executed successfully. - RTMCAN_E_INTERFACE_IS_BUSY: The RtmCan is busy, thus the set state request was omitted. - RTMCAN_E_NO_MATCH_FOUND: No MP is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of one MP with the specified ID.	
Particularities and Limitations	
> This function is synchronous. > This function is not reentrant. > RtmCan must be in state `RTMCAN_STATE_WAITFORACTION`.	
Call context	
> This function shall be called on task or interrupt level.	

Table 7-19 RtmCan_SetMPStateByID

7.6 RtmCan (CAPL) callout functions

The RtmCan notifies the application (RtmCan user) about the reception of a positive response from ECU and the end of a measurement.

7.6.1 Appl_RtmCanMeasurementFinished

Prototype	
`void Appl_RtmCanMeasurementFinished(void)`	
Parameter	
`void`	N/A.
Return code	
`void`	N.A.
Functional Description	
This function is called by RtmCan after the completion of a measurement. If this function is called, the measurement results are available and can be requested via RtmCan_GetMPResultByID/-Name or the reports can be generated with RtmCan_GenerateReport.	
Particularities and Limitations	
> This function has to be implemented by the application (RtmCan user). > Only called if serial or parallel measurement was executed.	
Call context	
> interrupt or task context > Only during measurements	

Table 7-20 Appl_RtmCanMeasurementFinished

7.6.2 Appl_RtmCanRespReceived

Prototype	
`void Appl_RtmCanRespReceived(enum RtmCan_StateType originState)`	
Parameter	
`originState`	The RtmCan state before the received Rtm_Resp triggers a state transition.
Return code	
`void`	N.A.
Functional Description	
This function is called by RtmCan if following conditions are fulfilled: 1. A positive response from ECU was received. 2. The previous RtmCan state (`originState`) was `RTMCAN_STATE_CLEARRESULTS` or `RTMCAN_STATE_MEASURING` 3. The current RtmCan state is `RTMCAN_STATE_WAITFORACTION`.	
Particularities and Limitations	
> This function has to be implemented by the application (RtmCan user).	

Call context
> interrupt or task context > Only during measurements

Table 7-21 Appl_RtmCanRespReceived

7.6.3 Appl_RtmCanTriggerReadingReceived

Prototype	
`void Appl_RtmCanTriggerReadingReceived(void)`	
Parameter	
`void`	N/A.
Return code	
`void`	N.A.
Functional Description	
This function is called by RtmCan if following conditions are fulfilled: 1. A positive response from ECU was received. 2. The current RtmCan state is `RTMCAN_STATE_MEASURING`.	
Particularities and Limitations	
> This function has to be implemented by the application (RtmCan user).	
Call context	
> interrupt or task context > Only during measurements	

Table 7-22 Appl_RtmCanTriggerReadingReceived

7.7 Example Application

Example
Here is an example that has been prepared for you.

Content of example file RtmCan_TestApplication.can:

```
Includes {
#include "RtmCan.cin"
}

on preStart {
  // Set debug level to Info (print all debug messages).
  RtmCan_SetDebugLevel(RTMCAN_DEBUGLEVEL_INFO);
}

on key 's' {
```

```
  enum RtmCan_ReturnValueType retVal;

  // Clear all results and config settings of RtmCan and Rtm.
  retVal = RtmCan_ClearAll();

  if (retVal != RTMCAN_E_OK)
    write("RtmCan failed to clear results.");
}

// Callback function, called by RtmCan
void Appl_RtmCanRespReceived(enum RtmCan_StateType originState) {
  enum RtmCan_ReturnValueType retVal;

  if (RtmCan_GetRtmCanState() == RTMCAN_STATE_WAITFORACTION) {
    if (originState == RTMCAN_STATE_CLEARRESULTS) {
      // Clear request was successful.
      // Activate all MPs
      retVal = RtmCan_SetMPStateAll(RTMCAN_MP_STATE_ACTIVE);

      if (retVal != RTMCAN_E_OK)
        write("RtmCan failed to set all MPs active.");

      // Now start measurement in parallel measurement for 1s.
      retVal = RtmCan_StartMeasurement(RTMCAN_MODE_PARALLEL, 1000);

      if (retVal != RTMCAN_E_OK)
        write("RtmCan failed to start the parallel measurement.");
    }
  }
}

// Callback function, called by RtmCan
void Appl_RtmCanTriggerReadingReceived() {
}

// Callback function, called by RtmCan
void Appl_RtmCanMeasurementFinished() {
  enum RtmCan_ReturnValueType retVal;
  struct RtmCan_ResultType result;
  dword i;

  // The measurement was successfully finished.
  // Now, generate the available reports
  retVal = RtmCan_GenerateReport(RTMCAN_REPORT_ALL);

  if (retVal != RTMCAN_E_OK)
    write("RtmCan failed to generate report.");

  // Get the measurement result of last MP.
  retVal = RtmCan_GetMPResultByID((RTMCAN_MEASUREMENTS_COUNT - 1), result);

  // Print the result to Write Window of CANoe.
  if (retVal == RTMCAN_E_OK) {
    write("MP \"%s\" Name: %s", result.RtmCan_Result_Name, result.RtmCan_Result_Name);
    write("MP \"%s\" Group: %s", result.RtmCan_Result_Name, result.RtmCan_Result_Group);
    write("MP \"%s\" ID: %d", result.RtmCan_Result_Name, result.RtmCan_Result_ID);
    write("MP \"%s\" Min [us]: %f", result.RtmCan_Result_Name, result.RtmCan_Result_MinRuntime_us);
    write("MP \"%s\" Max [us]: %f", result.RtmCan_Result_Name, result.RtmCan_Result_MaxRuntime_us);
    write("MP \"%s\" Average Runtime [us]: %f", result.RtmCan_Result_Name,
result.RtmCan_Result_AverageRuntime_us);
    write("MP \"%s\" Number of execution: %d", result.RtmCan_Result_Name,
result.RtmCan_Result_NumberOfExecution);
    write("MP \"%s\" CPU Load: %f", result.RtmCan_Result_Name, result.RtmCan_Result_CpuLoad);
    write("MP \"%s\" Measurement Duration [us]: %d", result.RtmCan_Result_Name,
result.RtmCan_Result_MeasurementDuration_us);
    if (result.RtmCan_Result_AssignedToCore < RTMCAN_NUMBER_OF_CORES) {
      write("MP \"%s\" Assigned to core: %d", result.RtmCan_Result_Name,
result.RtmCan_Result_AssignedToCore);
    }
    else
      write("MP \"%s\" not assigned to any core", result.RtmCan_Result_Name);
  }
  else
    write("RtmCan failed to get the result of last MP.");
}
```

Description:

The example application includes the RtmCan.cin file to be able to use its services.

The function `on start` is called by CANoe once at the beginning of CANoe simulation. Within this function the RtmCan function `RtmCan_SetDebugLevel` is called to set the debug level. All debug information is written to Write window of CANoe.

If the button ‘s’ is pressed on the keyboard, the last measurement results are cleared on RtmCan and in the ECU.

The callback function `Appl_RtmCanRespReceived(enum RtmCan_StateType originState)` is called after successful removal of measurement results on ECU. The `originState` is `RTMCAN_STATE_CLEARRESULTS`, whereas the current state of RtmCan is `RTMCAN_STATE_WAITFORACTION`.

Within the function `Appl_RtmCanRespReceived` all MPs are set to active by the call of `RtmCan_SetMPStateAll`. Afterwards a measurement in parallel mode is started for 1s by calling `RtmCan_StartMeasurement`.

The measurement is started on ECU, after one second the measurement is finished and all measurement results are sent to CANoe via XCP. After all measurement results are available in CANoe, the callback function `Appl_RtmCanMeasurementFinished` is called by RtmCan.

Within this function the result analysis can take place. Therefore, all available *.csv reports are generated by calling the function `RtmCan_GenerateReport`. Additionally, the result of the last available MP is requested by calling `RtmCan_GetMPResultByID`. The returned result is printed to Write window of CANoe.

Pressing the button again, repeats the program.

7.7.1 Integrate the RtmCan_TestApplication.can file in CANoe

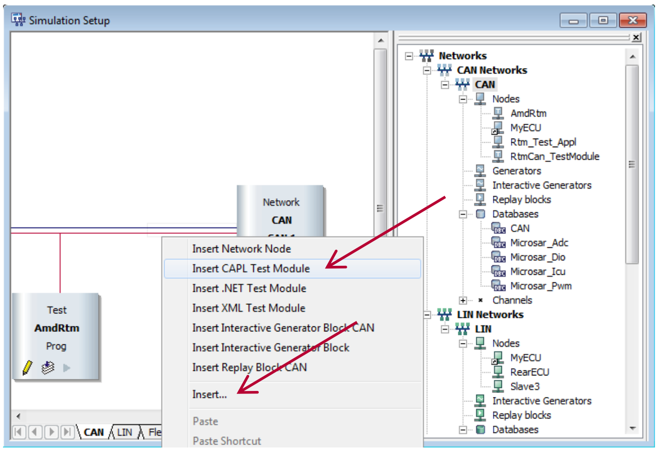
To create a new CAPL file in CANoe there is an easy way.

For this purpose, open the Simulation Setup in the Simulation Ribbon. Switch to the network node where the CAPL file should be integrated.

Right click the wire and choose:

1. **Insert CAPL Test Module** or
2. **Insert…**

Note: For detailed description of the differences between these options please refer to CANoe’s help.

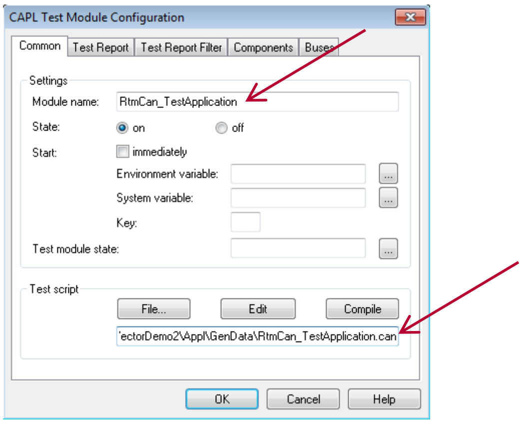


> Right click the new test node and choose **Configuration…**

> Specify a Module name (here **RtmCan_TestApplication**) and the Test script

  > If there is already a CAPL file, choose **File…** and navigate to it.

  > If there is no CAPL file, specify the directory and add the name with **.can** (here **RtmCan_TestApplication.can**) at the end. Then click **Edit**. The file is generated and opened in CAPL browser automatically.



Now, the previously introduced CAPL code can be added to this file or a new code can be written.

Note
The generated file RtmCan.cin must be located at the same location as the self-written CAPL file (in the previous example **RtmCan_TestApplication.can**).

Caution
If you have already added the Rtm Test Module (Rtm_Canoe.xml/.can) to CANoe, you must deactivate it before using the self-written CAPL file and vice versa. Because both files include RtmCan.cin.

8 Glossary and Abbreviations

8.1 Glossary

Term	Description
Measurement Point	Code sequence which runtime shall be measured. Delimiters are: Rtm_Start(A), Rtm_Stop(A).
Measurement Point Type	Indicates the measuring behavior of a measurement point. Possible types are GrossExecutionTime, NetExecutionTime and FlatExecutionTime.

Table 8-1 Glossary

8.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DET	Development Error Tracer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
LET	Logical Execution Time
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
MP	Measurement Point
RTE	Runtime Environment
RtmCan	Rtm CAPL on CANoe/CANape
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 8-2 Abbreviations

9 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com