VECTOR

MICROSAR Runtime Measurement

Technical Reference

Version 3.01.01

Authors	Alexander Zeeb, Oliver Reineke, David Zentner
Status	Released

Document Information

History

Author	Date	Version	Remarks
Alexander Zeeb	2013-05-02	1.00.00	Initial version
Oliver Reineke	2013-08-06	1.00.01	ESCAN00068380, ESCAN00069596
Oliver Reineke	2013-11-29	1.01.00	ESCAN00069152, ESCAN00072340, ESCAN00068381
Oliver Reineke	2014-05-21	1.01.01	ESCAN00075161
David Zentner	2014-09-22	1.02.00	ESCAN00070189, ESCAN00076463
David Zentner	2014-12-04	1.02.01	ESCAN00079842, ESCAN00079844, ESCAN00079535
David Zentner	2015-12-03	2.00.00	ESCAN00085574, ESCAN00083646
David Zentner	2016-03-10	2.00.01	ESCAN00088550
David Zentner	2016-05-17	2.01.00	
David Zentner	2016-09-21	2.02.00	ESCAN00091859, ESCAN00092231
David Zentner	2017-02-03	2.02.01	ESCAN00093857
David Zentner	2017-04-07	3.00.00	STORYC-724
David Zentner	2017-04-21	3.01.00	STORYC-671, STORYC-672
David Zentner	2017-06-13	3.01.01	ESCAN00095484, ESCAN00091207

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

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 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
1.00.00	Initial version of RTM for AUTOSAR 4.0.
1.01.00	Support Measurement of Net Runtimes.
1.02.00	CPU load can be measured in serial and parallel measurement mode.
2.00.00	Multicore support.
3.00.00	Schedule function for net runtime measurement.

Table 1-1 Component history

2 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.

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 or FlexRay network communication).

2.1 Architecture Overview

The Figure 2-2 shows the interfaces to adjacent modules of RTM. These interfaces are described in chapter 5.

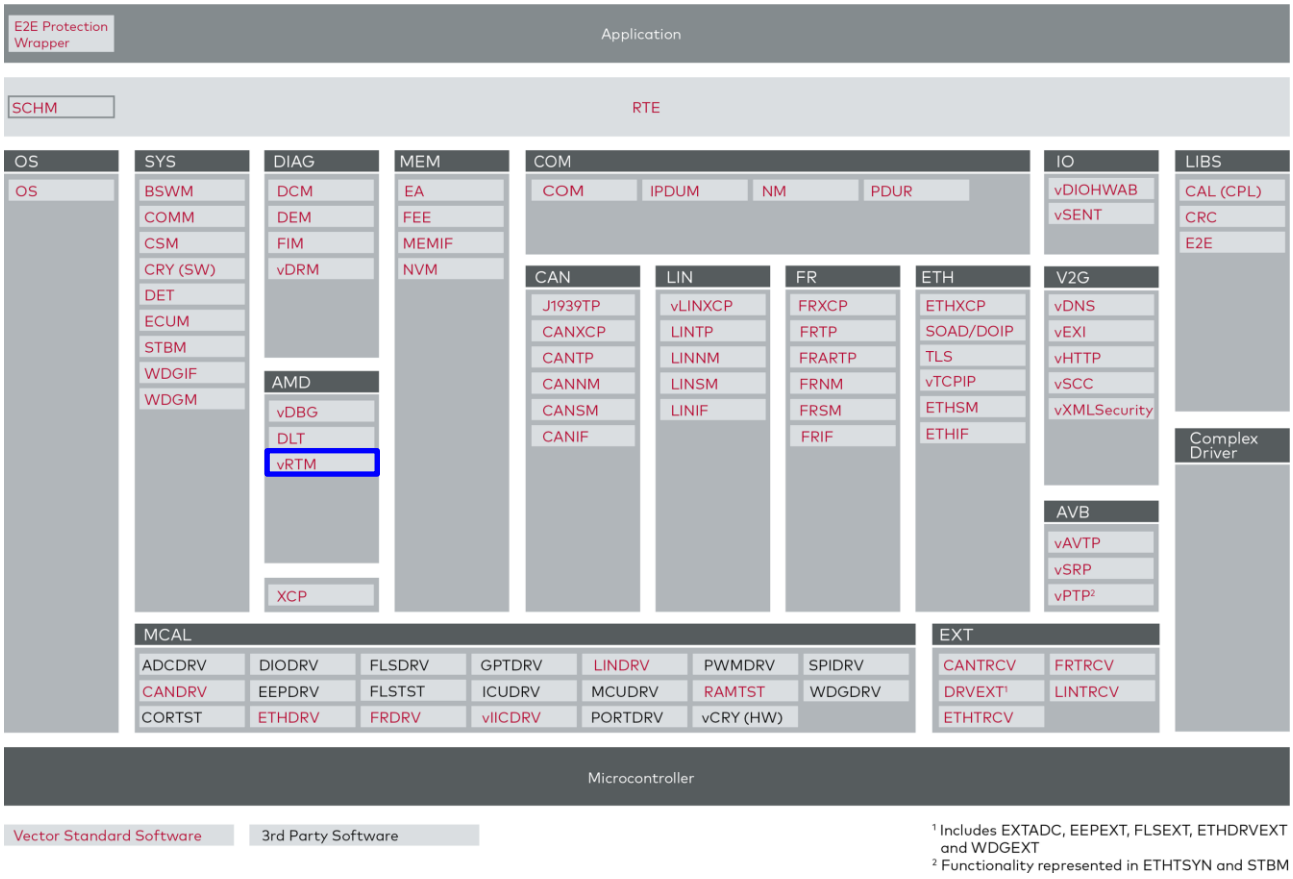


Figure 2-1 AUTOSAR 4.x Architecture Overview

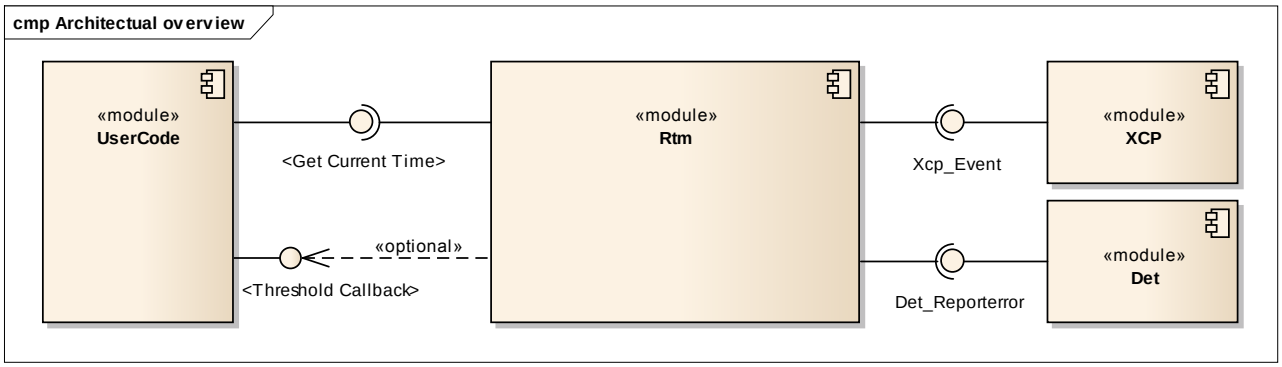


Figure 2-2 Interfaces to adjacent modules of the RTM

3 Functional Description

3.1 Features

RTM allows several measurement points within the ECU's embedded code. Each measurement point is identified uniquely, allowing measurement of several measurement points simultaneously without interference. Measurement points 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 measurement points 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: 3.1.8)).

Performing measurement with RTM means activating one or more measurement points. The measurement points can be activated within CANoe with help of RTM's frontend or a self-written Rtm application. 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 those cases, the measurement has to be repeated. This problem can be minimized by selecting long measurement duration.

To reduce RTM's code- and data size as well as runtime, measurement points 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 specific measurement point

In CANoe RTM allows individual activation of available measurement points at runtime. Thereby three different measurement variants are supported:

- Parallel Measurement
- Serial Measurement
- Live Measurement

3.1.1 Parallel Measurement

With parallel measurement, all measurement points enabled at pre-compile time and activated in CANoe are measured simultaneously. The measurement duration is specified at measurement start and applies to all selected measurements.

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.

3.1.2 Serial Measurement

With serial measurement, all measurement points enabled at pre-compile time and activated in CANoe are measured consecutively. The measurement duration is specified at measurement start and applies for each measurement individually. .

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.

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

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

3.1.3 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:

- <Measurement Point Name>_cpuload
- <Measurement Point 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.

3.1.4 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.

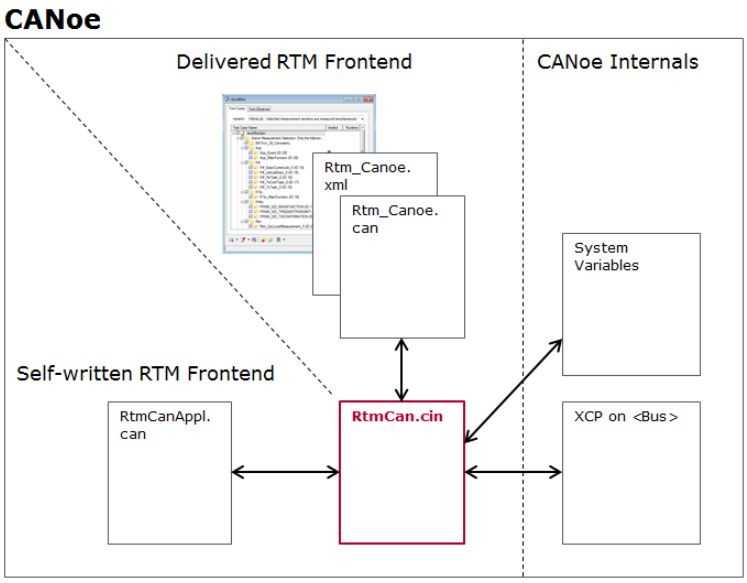


Figure 3-1 RTM control within CANoe

The delivered RTM frontend provides static control of the runtime and overall CPU load measurement. The active measurement points, the measurement variant and measurement duration are selectable. A html test report can be generated after measurement end. More details follow in chapter 3.1.5.

With the self-written RTM application the same measurement points, 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 Measurement Point was executed. */
  }
  else {
    /* This Measurement Point was not executed. */
  }
…
```

A short overview of RtmCan.cin is given in chapter 4.6.3. The detailed description follows in chapter 8.

3.1.5 CANoe Frontend

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

Measurement points 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.

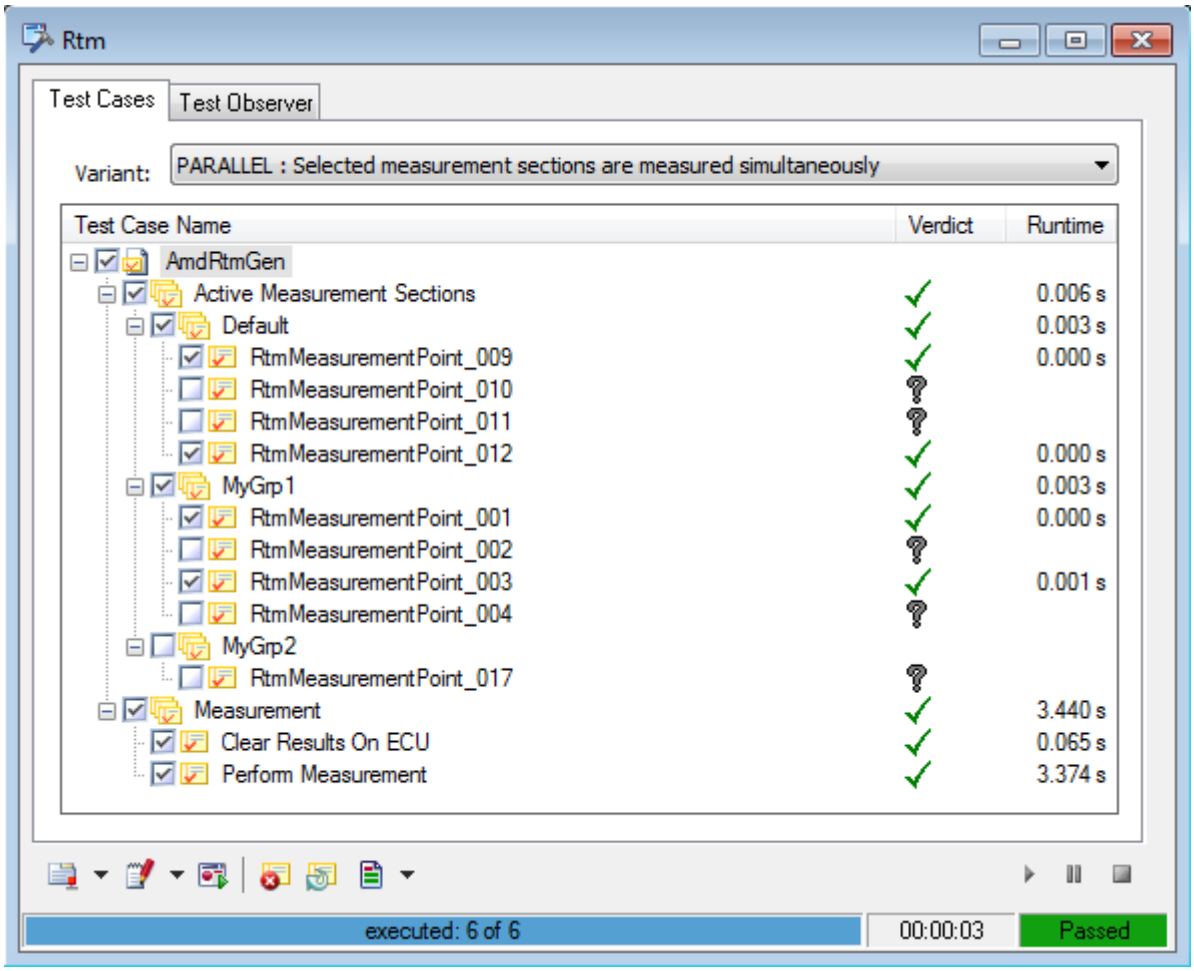


Figure 3-2 RTM CANoe Frontend

3.1.6 Report Generation

For serial and parallel measurement variants from chapter 3.1.1 and 3.1.2 a report is generated. The test report contains the following information for each measurement point:

- 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 3.1.13.1.

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

3.1.6.1 Report for Timing-Architects™

A second report is generated, containing only measurement points 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 measurement points.

Note
RTM does not check measurement points 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.

3.1.7 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/RtmGeneral/RtmCpuLoadControlMode`.

Off	The CPU load measurement is turned off and introduces no additional memory and runtime.
C_API	The CPU load measurement requires a special measurement point for each core with short name `Rtm_CpuLoadMeasurement(_CoreId)`. This measurement point has the same configuration parameter as any other measurement point. The measurement is started and stopped by API that can be called within the BSW and SWCs (`Rtm_Start_CpuLoadMeasurement` and `Rtm_Stop_CpuLoadMeasurement`). The result can be requested via `Rtm_GetMeasurementItem`.
Xcp	The CPU load measurement requires a special measurement point with the short name `Rtm_CpuLoadMeasurement(_CoreId)`. This measurement point has the same configuration parameter as any other measurement point. 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 3-1 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 measurement point `Rtm_CpuLoadMeasurement(_CoreId)` cannot be used as usual measurement point. It is reserved for the CPU load measurement, independent of the used control mode.

3.1.8 Auto start Measurement Points

Measurement points can be configured to start measurement right at ECU start up before the XCP connection to CANoe is established. Auto start measurement points perform measurement until any other measurement is started by RTM's frontend. To enable auto start for a measurement point, 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. Therefor the measurement point 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 measurement points 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.

3.1.9 Runtime Threshold Callbacks

Each measurement point can be configured to have a runtime threshold, selectable via DaVinci Configurator parameter:

`/MICROSAR/Rtm/RtmMeasurementPoint/RtmRuntimeThreshold`

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.

3.1.10 Calibration

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

3.1.11 Measurement Point Types

There are three different types of measurement points:

1. ResponseTime
2. ExecutionTime_NonNested
3. ExecutionTime_Nested

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

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

3.1.11.1 ResponseTime

Measurement points measuring the response time, measure the absolute time from measurement start to stop. These measurement points do not consider interruptions in their measurements.

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

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

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

Table 3-2 Measurement Config – all MPs ResponseTime

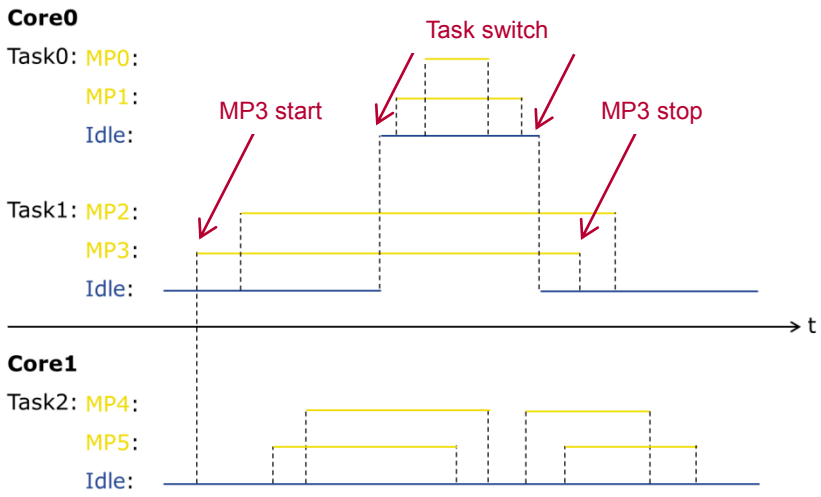


Figure 3-3 All measurement points of type ResponseTime

Each measurement point measures the time from start until stop, this is the response time. Measurement point `MP3` is started on `Core1` and stopped on `Core0`. This is only possible for measurement points set to ResponseTime. 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.

3.1.11.2 ExecutionTime

Measurement points measuring execution time remove the execution times of interrupting tasks and ISRs from their measurement results.

To enable these measurements RTM provides two mechanisms:

1. Use of OS Pre-/PostTaskHooks and Pre-/PostISRHooks:

The hook functions have to be implemented by the user and call the following RTM functions:

Hook Function	RTM Function
PreTaskHook	Rtm_EnterTaskFct
PostTaskHook	Rtm_LeaveTaskFct
PreISRHook	Rtm_EnterIsrFct
PostISRHook	Rtm_LeaveIsrFct

Table 3-3 OS Hook Function mapping (Pre-/PostTaskHooks and Pre-/PostISRHooks)

2. Use of OS TimingHooks:

The following Vector OS hook function has to be implemented which calls the following RTM function:

Hook Function	RTM Function
OS_VTH_SCHEDULE	Rtm_Schedule

To enable this RTM feature, the OS TimingHooks have to be enabled by including the Rtm.h in the OS configuration (/MICROSAR/Os/OsOS/OsDebug/OsTimingHooksIncludeHeader).

In the Rtm.h the hook function is already implemented.

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.

Caution
Use one mechanism exclusively to measure execution time.

3.1.11.2.1 ExecutionTime_NonNested

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

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

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

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

Table 3-4 Measurement Config – all MPs ExecutionTime_NonNested

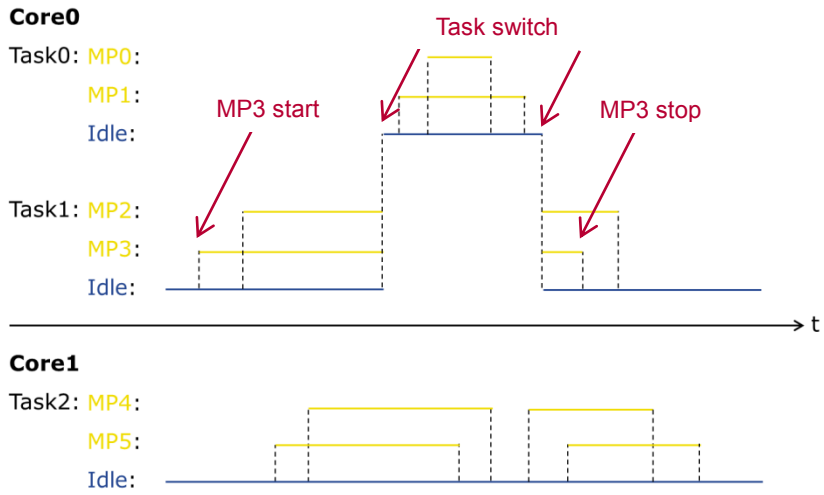


Figure 3-4 All measurement points of type ExecutionTime_NonNested

Each measurement point can be interrupted by task switches and interrupts. Thus, they only measure the runtime on their own task.

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

3.1.11.2.2 ExecutionTime_Nested

Measurement points of type ExecutionTime_Nested consider the execution time of nested measurement points additionally to the execution time of interrupting tasks and ISRs.

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

Caution
The execution time of a nested measurement point is only subtracted by the outer measurement point, if the nested measurement point is of type ExecutionTime_NonNested or ExecutionTime_Nested.

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

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	ExecutionTime_(Non)Nested	0	0
MP1	ExecutionTime_Nested	0	0
MP2	ExecutionTime_(Non)Nested	0	1
MP3	ExecutionTime_Nested	0	1
MP4	ExecutionTime_(Non)Nested	1	2
MP5	ExecutionTime_Nested	1	2

Table 3-5 Measurement Config – all MPs ExecutionTime_Nested

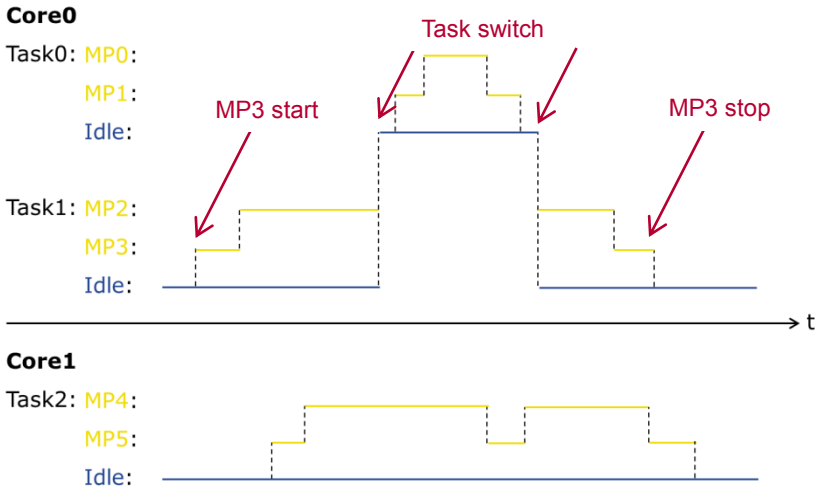


Figure 3-5 All measurement points of type ExecutionTime_Nested

Each measurement point can be interrupted by task switches, interrupts and other measurement points 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`.

3.1.11.3 Mixed measurement point types

Meas. Point Name	Meas. Type	Assigned to core	Assigned to task
MP0	ExecutionTime_Nested	0	0

MP1	ExecutionTime_NonNested	0	0
MP2	ResponseTime	0	1
MP3	ExecutionTime_Nested	0	1
MP4	ExecutionTime_Nested	1	2
MP5	ExecutionTime_Nested	1	2

Table 3-6 Measurement Config – mixed measurement point types

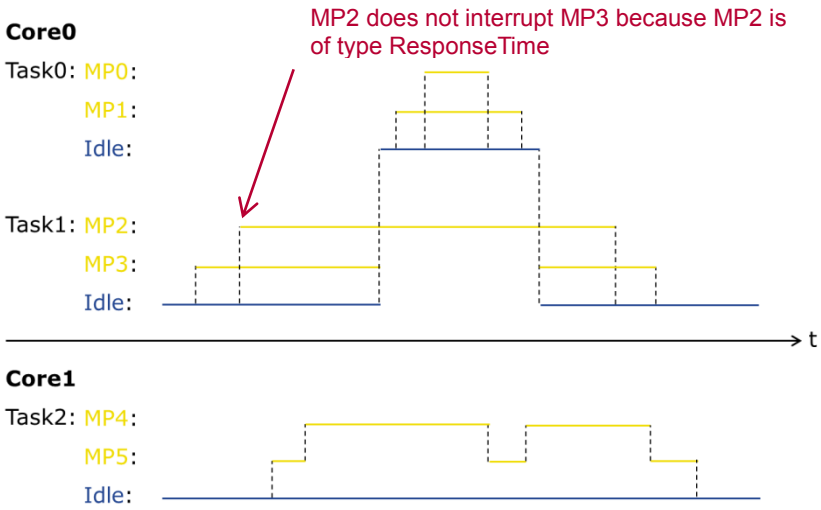


Figure 3-6 Mixed measurement types

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

Even though `MP3` is set `ExecutionTime_Nested` 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 `ResponseTime`. MPs of type `ResponseTime` do not affect the runtime of other measurement points. 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 `ExecutionTime_NonNested`.

3.1.11.4 Functionality of nested measurement points

If the measurement point's type measuring `Func_1` is set to `ExecutionTime_Nested` and the called function `Func_2` is set to `ExecutionTime_NonNested` or `ExecutionTime_Nested`, the execution time of `Func_1` is decreased by the execution time of `Func_2`. This is shown in Figure 3-7 on the right.

If the called function is not measurement by a separate measurement point, or one of both measurement points is of type `ResponseTime`, the execution time of `Func_1` does not consider the execution time of the inner function `SubFunc_1`. This is shown in Figure 3-7 on the left.

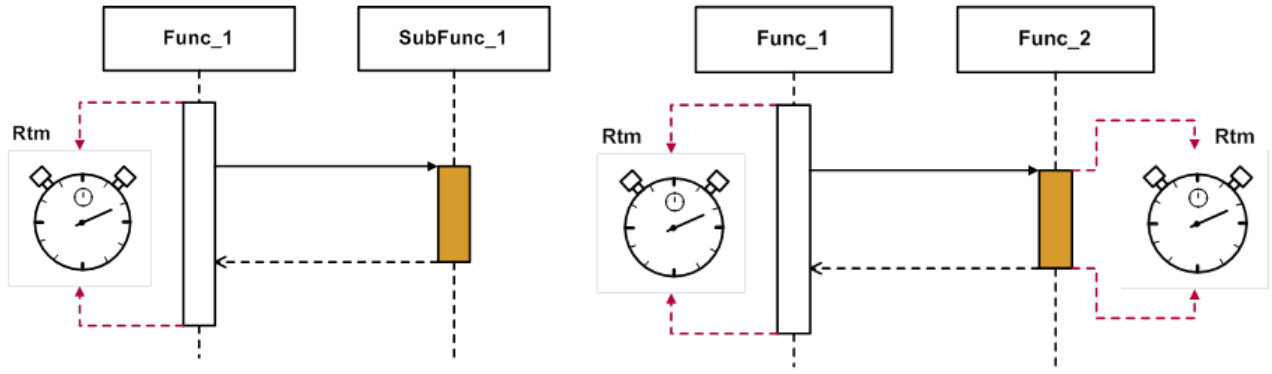


Figure 3-7 On the left: complete runtime of Func_1 is measured, on the right: the runtime of Func_2 is subtracted from Func_1

Caution
Because serial measurement does only measure one measurement point 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.

3.1.12 Nested counter

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

3.1.13 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 parameter `RtmMulticoreSupport`.

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
  - The timer request must be reentrant OR
  - The access to the timer value must be atomic OR
  - OS Spinlocks have to be used
- The RTE and the OS have to support multi core

Caution
The GPT-APIs Gpt_GetTimeElapsed and Gpt_GetTimeRemaining are not reentrant, therefore OS Spinlocks have to be used if RTM's multi core feature is enabled.

3.1.13.1 Assignment of measurement points to a core

Each measurement point can be assigned to one core by setting the parameter `RtmAssignedToCore`. This is only effective in multi core systems and if `RtmMulticoreSupport` is enabled.

If the measurement point is not assigned to any core, the measurement point can be started on any core and stopped on any core. It is allowed to start and stop the measurement point on the same core. But it is recommended to use always the same core for starting and always the same core for stopping a specific measurement point.

If the same measurement point 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 measurement point is not assigned to any core, because it has to be verified that a measurement point is never started and stopped simultaneously.

If this parameter is set to a valid core ID, this measurement point 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.

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

3.1.13.2 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 3-8 Measurement point execution on multi core systems

Up to now the RTM does not support execution time measurement on multi cores.

3.1.14 Safe RTM

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

The RTM provides a mechanism to be used within ASIL software. For this purpose the RTM's functionality can be disabled 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`

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

3.1.15 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 measurement point for each runnable. These measurement points can be started and stopped by using VFB trace hook function generated by the RTE.

The configuration of measurement points, VFB trace functions and their implementation is assisted through the RTE comfort editor and a script provided by RTM. The necessary configuration steps are explained in the following.

1. Generate the RTE.
2. In the RTE comfort editor, open the 'Import Assistant' and select the generated file Rte_Hook.h.
3. Select all Start/Return pairs for those runnable 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.

3.2 Initialization

Before calling any other functionality (Except auto start measurement points) of the RTM module, the initialization function `Rtm_Init()` has to be called.

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 5.2.5 ‘Rtm_InitMemory’.

3.3 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.

3.4 Error Handling

3.4.1 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
1	Rtm_CpuLoadMeasurementFunction
2	Rtm_GetVersionInfo
3	Rtm_Init
4	Rtm_InitMemory
5	Rtm_MainFunction(_<CoreID>)
6	Rtm_StartMeasurementFct
7	Rtm_StopMeasurementFct
8	Rtm_GetMeasurementItem
9	Rtm_EnterTaskFct
10	Rtm_LeaveTaskFct
11	Rtm_EnterIsrFct
12	Rtm_LeaveIsrFct
13	Rtm_Schedule

Service ID	Service
14	Rtm_StartNetMeasurementFct
15	Rtm_StopNetMeasurementFct

Table 3-7 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)

Table 3-8 Errors reported to DET

3.5.2 Production Code Error Reporting

No production error codes are currently defined for RTM.

4 Integration

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

4.1 Scope of Delivery

The delivery of the RTM contains the files which are described in the chapters 4.1.1 and 4.1.2:

4.1.1 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 measurement points. It is located in ‘./external/DaVinciConfigurator/AutomationScripts’.

Table 4-1 Static files

4.1.2 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'.

Table 4-2 Generated files

4.2 Include Structure

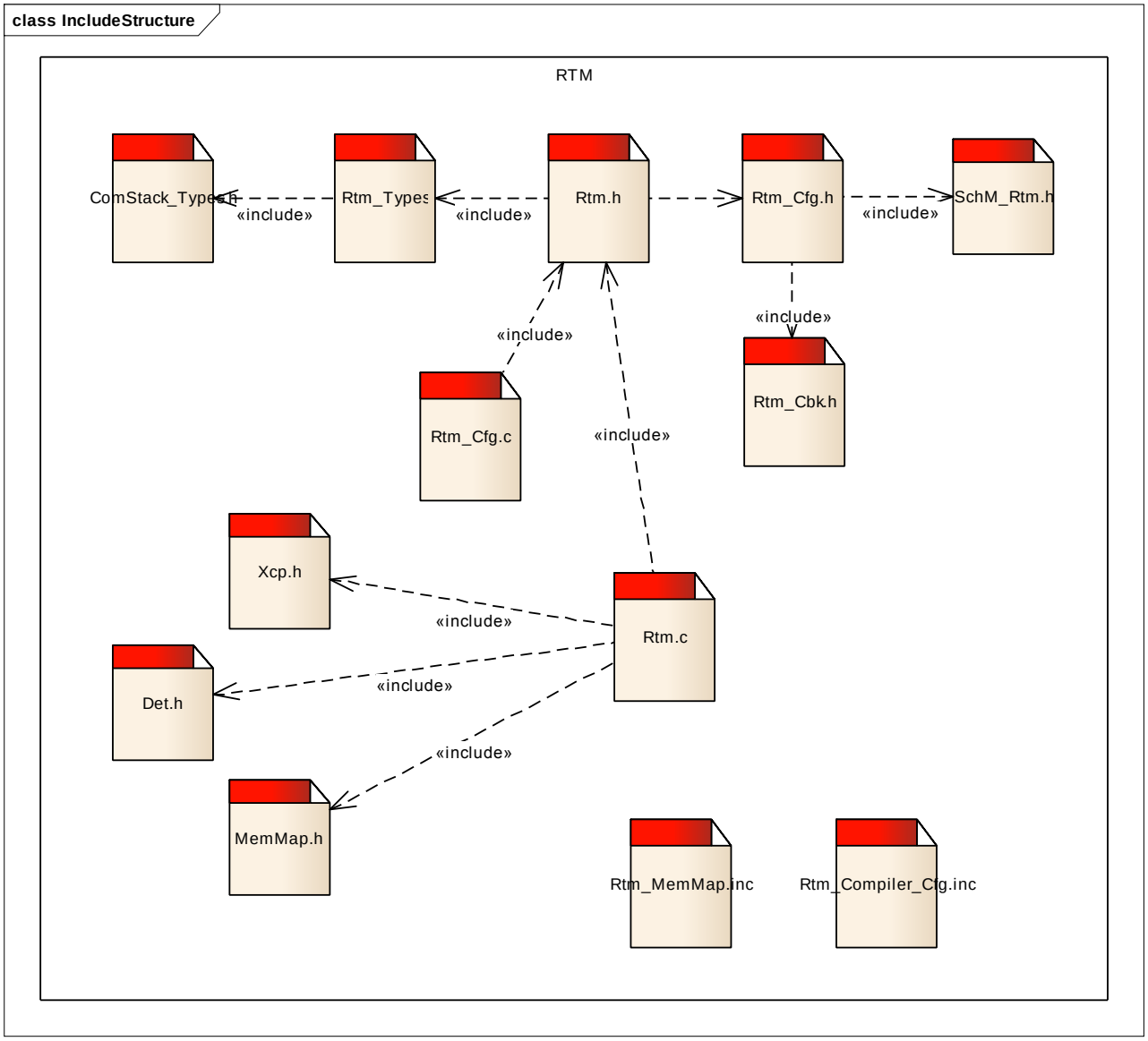


Figure 4-1 Include Structure

4.3 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 measurement points. 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 measurement points. 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 measurement points the reduction of runtime can be great.

Example configuration:

- 100 measurement points
- All measurement points are enabled
- Parallel measurement is chosen
- Each measurement point is executed 100 times per second.
  - Rtm_Start(…) is called 100 times and
  - Rtm_Stop(…) is called 100 times
- All measurement points 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:

$$ExeTime[s] = \#MeasPts * \#APIsRtmStartStop * \#Calls * \#APIsExclusiveArea * ExeTimeExclusiveAreaAPIs$$

$$100 * 2 * 100 * 2 * 1\mu s = 0.04s$$

execution time for AREA_0.

This means the CPU load introduced by calling AREA_0 is:

$$\mathrm{CPUloadAREA}_0 = \frac{0.04s * 100}{1s} = 4\%$$

Note
This additional CPU load is only introduced if the measurement is active. As long as 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.

4.4 Embedded Code

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

4.4.1 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.

It is recommended to set the counter to a frequency of 10000 ticks per millisecond (DaVinci Configurator parameter: `/MICROSAR/Rtm/RtmGeneral/RtmMeasurementCtrFrequency` **[kHz]**) since this setting provides a good tradeoff between measurement resolution (0.1µs) and measurable runtimes (~6.5 ms).

However, it is possible to use alternative frequencies (f_{ctr}). The resolution is then: $1/f_{ctr}$ [ms] while the maximum runtime is: 2^16 / f_{ctr} [ms]

Caution
The result of a time measurement is incorrect if the measured runtime exceeds 2^16 / f_{ctr}.

RTM expects a callback function or function like macro which returns a 16-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 default setting is: RtmGetMeasurementTimestampFct.

The prototype is:

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

This user implemented function does not have to handle counter overflows. It merely shall return the current counter value.

It is possible to detect counter overruns. This is required if the measurement duration of each active measurement point is higher than the maximum counter value. Additionally Rtm_CheckTimerOverrun has to be called at least once per counter period.

The overrun detection can be enabled via:

`/MICROSAR/Rtm/RtmGeneral/RtmTimerOverrunSupport`.

The overrun detection increases the runtime overhead, thus it is recommended that it is only active if necessary.

4.4.2 Measurement Points

Measurement points 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 measurement point. 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.

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

Caution
Measurement points 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).

Nested measurement points (i.e. a measurement point within a function which is called recursively) are allowed, however only the outmost measurement will be evaluated.

Note
In case of nested measurement points, 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 measurement points. It is known that older CANoe versions only run stable with less measurement points.

4.4.3 CPU-Load Measurement

Measurement of the CPU's overall load (see chapter 3.1.7) requires a dedicated "idle task" to be configured in the Operating System. This task has to fulfill the following requirements:

- There may be no other task with a lower priority
- Scheduling is fully preemptive
- The task shall call the following function within an endless loop without delay: `Rtm_CpuLoadMeasurementFunction()`
- No other code is executed in the context of this task

Example code:

```
#include "Rtm.h"

TASK(MyIdleTask) /* Prio: system's lowest */
{
    while(1)
      Rtm_CpuLoadMeasurementFunction();
}
```

Figure 4-2 Example: Overall CPU-Load Measurement

Note
In multi core systems each core requires a background task with lowest priority on its core. All background tasks have to call the same function of RTM (`Rtm_CpuLoadMeasurementFunction()`).

4.5 A2L

Open the template-file: _Master.a2l (.\external\Misc\McData) and adapt all sections marked with "TODO:".

Most important for RTM is, that the following files from folder .\Config\McData are included:

- McData.a2l
- McData_Events.a2l

Both files contain (amongst others) RTM's configuration of the CANoe frontend.

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 (.\ASAP2Updater\Exec\ASAP2Updater.exe). This Updater requires the following arguments:

- Master.a2l (the previously prepared 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)

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

Please refer to the document **User Manual AUTOSAR Calibration** ([1]) for a more detailed description on how to set up a2l-files.

Note
The file McData.a2l is rewritten during each generation step 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.
```

4.6 CANoe

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

4.6.1 XCP configuration in CANoe

Precondition for this step is that the XCP driver on the ECU is configured appropriately. Please refer to document: **UserManual AMD – MICROSAR 4** ([2]) 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 4.5 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)**".

> Set **Hierachical partial identifiers** to "**Compatible with CANoe < 9.0**"

> Set **Extended Datatypes** to "**Compatible with CANoe < 9.0**"

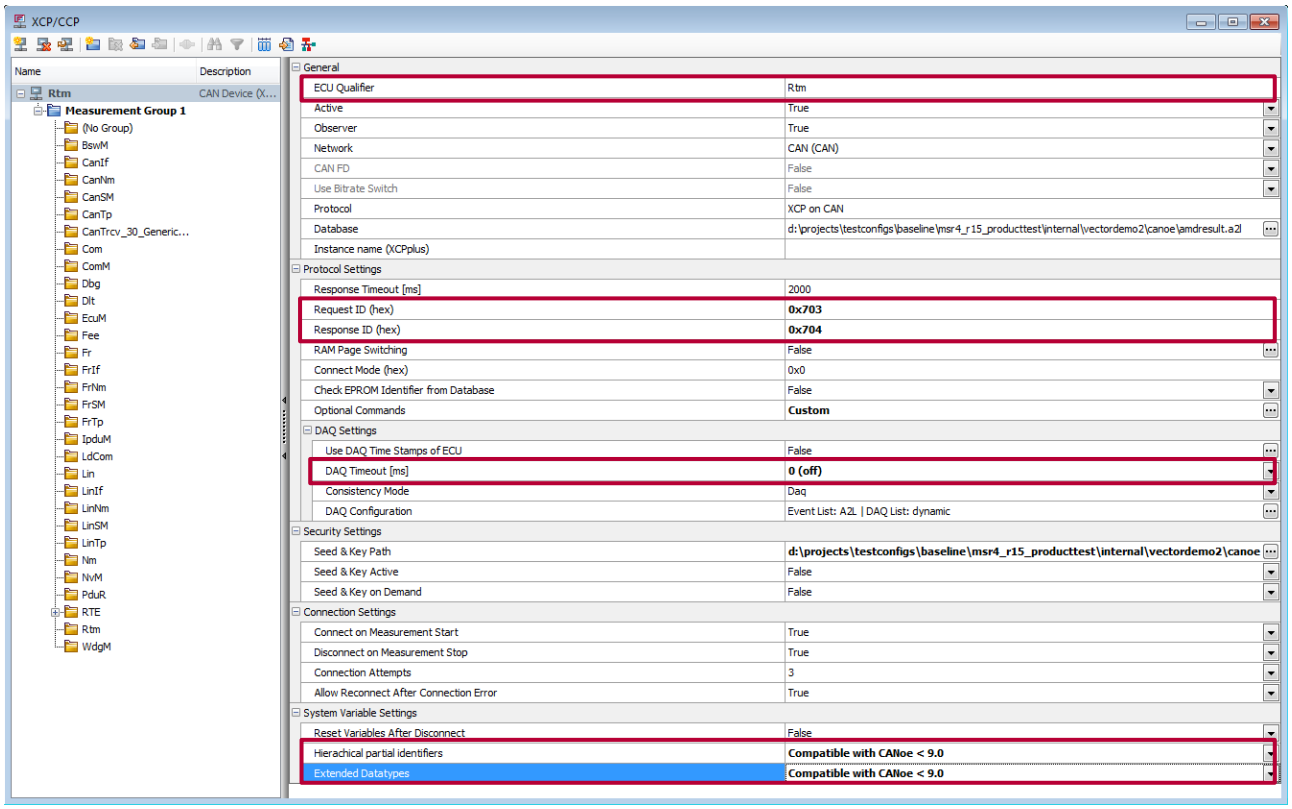


Figure 4-3 XCP/CCP Device Configuration

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**.

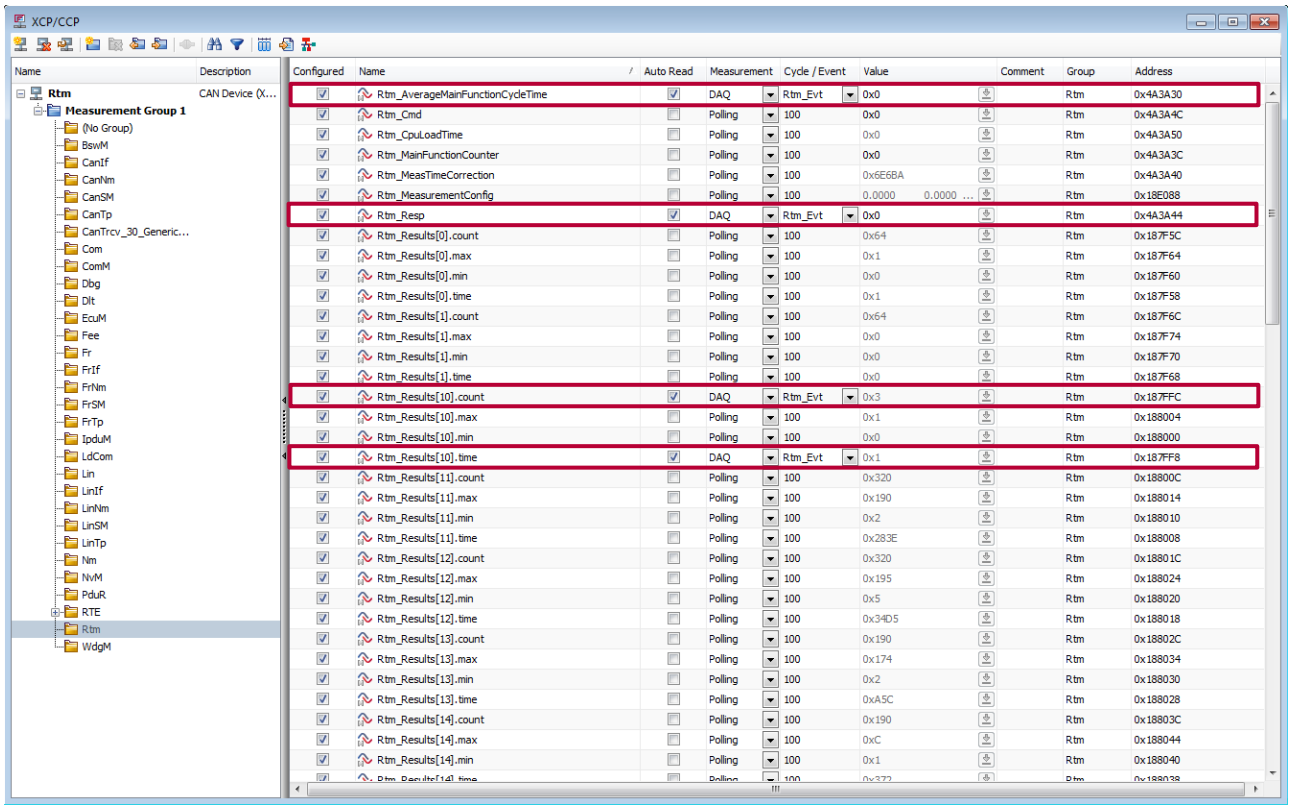


Figure 4-4 XCP/CCP Signal Configuration

Note

To identify which Rtm_Results[<MeasurementID>] belongs to which measurement point please refer to chapter 4.6.5.

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 measurement points simultaneously.

4.6.2 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 measurement points, choosing the measurement mode and starting the measurement.

4.6.2.1 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**

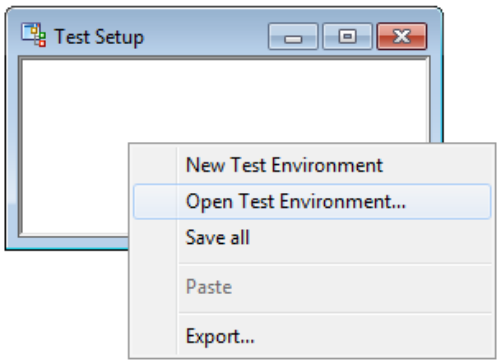


Figure 4-5 Test Setup: Import

> Choose GenData\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

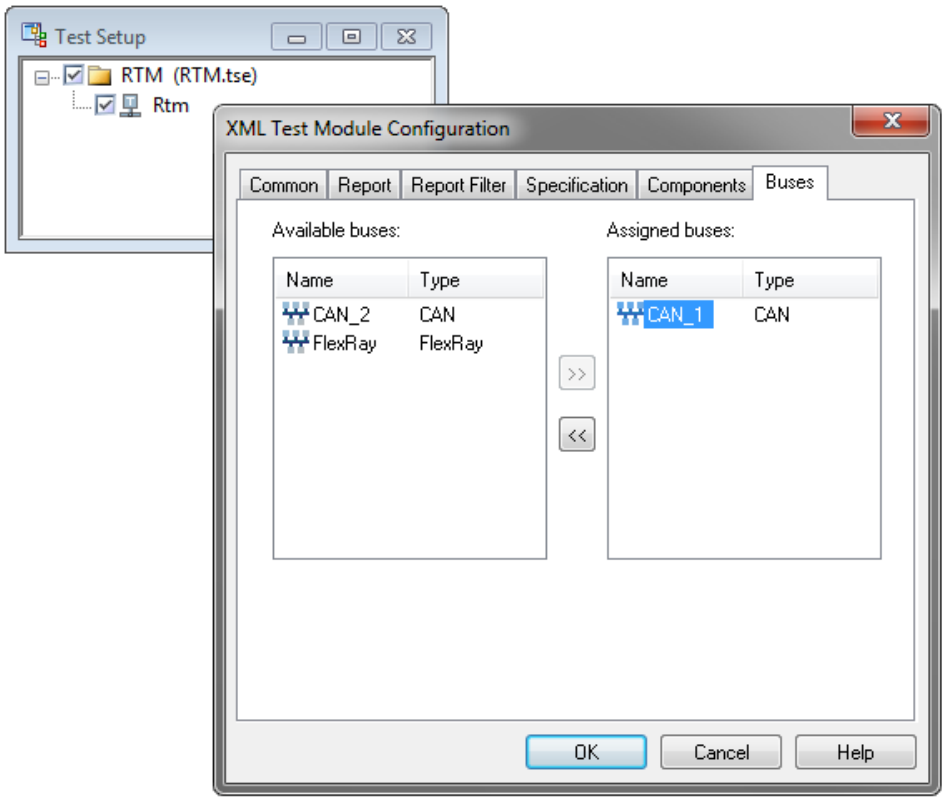


Figure 4-6 Test Setup: Configuration

4.6.2.2 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: 3.1.1, 3.1.2 and 3.1.3 for a description of the different variations.

> Tick the measurement points in the **Active Measurement Selection** which shall be measured. Only selected measurement points 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

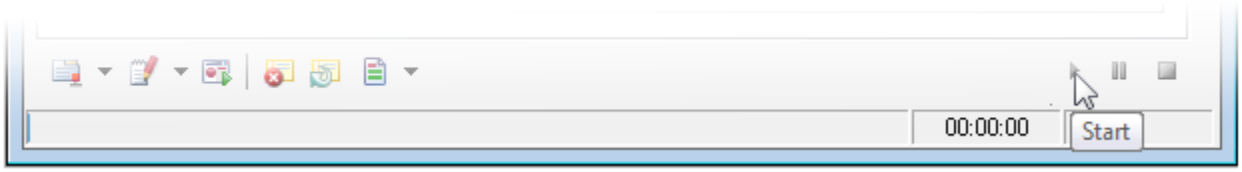


Figure 4-7 Measurement Start

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 measurement points, the calculated CPU-load is likely to be (close to) zero.

4.6.3 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 8.5.

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 8.

4.6.4 Result Report in Live Measurement Mode

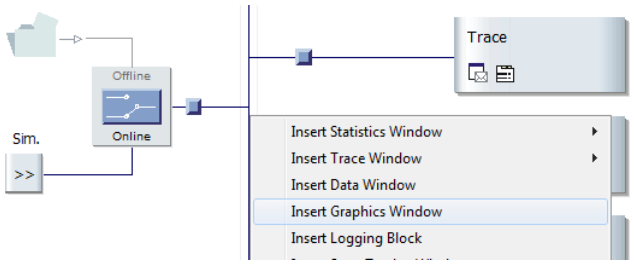
In live measurement mode no result report is generated, but there are two options to access the results.

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 4-8 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**.

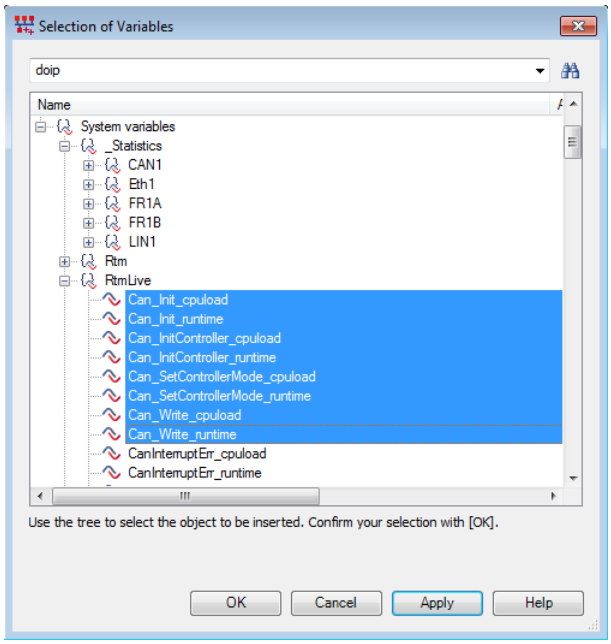


Figure 4-9 Add Measurement Points for Live Measurement

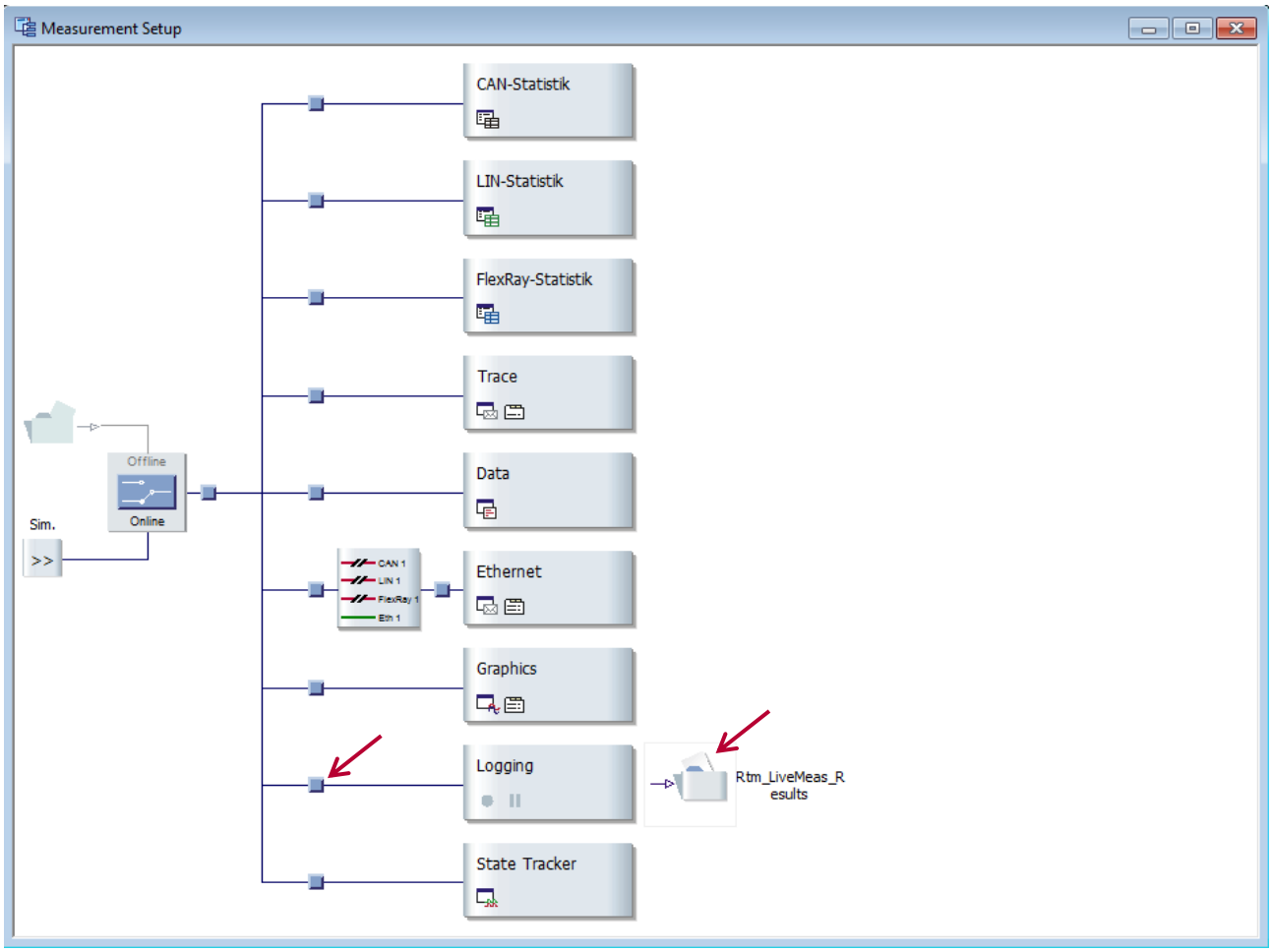
> If the **RtmLive** is not available, select the **Variant** “LIVE: Selected measurement sections …” in Rtm **Test Module**. And execute the Rtm measurement once. This is required because the live measurement signals are created at runtime.

To use the Logging Feature of CANoe:

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

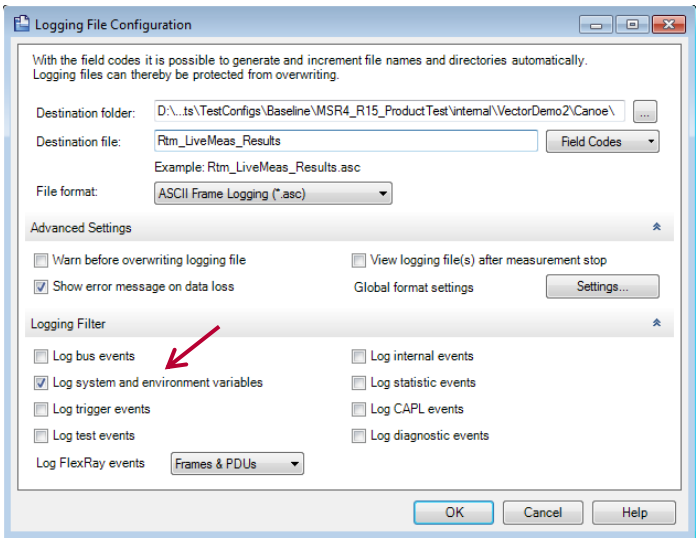
> Activate the Logging block by double click on the left square.

> 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 measurement point Rtm_CpuLoadMeasurement_runtime is µs, the unit of Rtm_CpuLoadMeasurement_cpuload is percent.

4.6.5 Mapping Measurement ID to Measurement Point

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

1. Via the **Test Module**

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

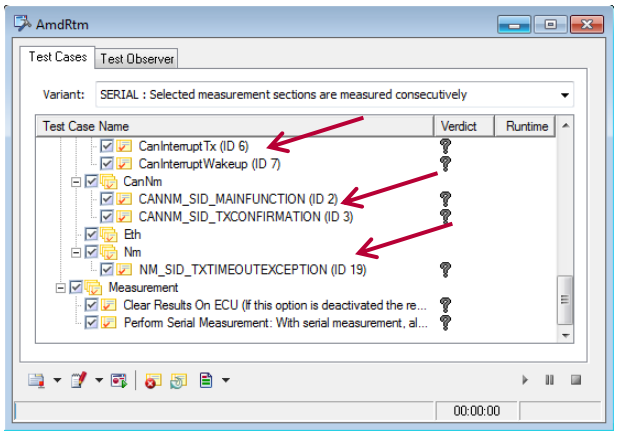


Figure 4-10 Measurement ID stands behind the name

2. Look in the RtmCan.cin

> Open the file with text editor or via CANoe.

> Look for the measurement point 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 4-11 Mapping between Measurement ID and Measurement Point

5 API Description

For an interfaces overview please see Figure 2-2.

5.1 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 5-1 Type definitions

Rtm_DataSet

This structure contains the measurement result of a measurement point. This structure is only cleared if explicitly requested. (Please refer to chapter 4.6.2.2 – Clear Results On ECU)

Struct Element Name	C-Type	Description	Value Range
Time	c-type	Accumulated runtime of this measurement point.	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 measurement point.	0 4294967295
Min	c-type	Absolute minimum runtime of this measurement point.	0 4294967295

Table 5-2 Rtm_DataSet

Rtm_ItemType

Defines the requested measurement result.

Enumeration	C-Type	Description	Value
RTM_ITEM_CPU_LOAD_AVERAGE	c-type	State of average CPU load	0
RTM_ITEM_CPU_LOAD_CURRENT	c-type	State of current CPU load	1
RTM_ITEM_MIN	c-type	State of minimum CPU load	2

Enumeration	C-Type	Description	Value
RTM_ITEM_MAX	c-type	State of maximum CPU load	3

Table 5-3 Rtm_ItemType

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
		Data type used for timestamp acquisition. If `(Rtm32BitTimer == ON)`	0
			4294967296

Table 5-4 Rtm_TimestampType

5.2 Services provided by RTM

5.2.1 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 5-5 Rtm_ConvertTicksToUs

5.2.2 Rtm_CpuLoadMeasurementFunction

Prototype	
`void Rtm_CpuLoadMeasurementFunction(void)`	
Parameter	
`void`	`N.A.`
Return code	
`void`	`N.A.`
Functional Description	
This function calculates the CPU's overall load. It has to be used as described in chapter 4.4.3.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' > This function is synchronous. > This function is reentrant.	
Expected Caller Context	
> This function requires task context for correct results.	

Table 5-6 Rtm_CpuLoadMeasurementFunction

5.2.3 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 5-7 Rtm_GetVersionInfo

5.2.4 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 5-8 Rtm_Init

5.2.5 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 5-9 Rtm_InitMemory

5.2.6 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 5-10 Rtm_MainFunction

5.2.7 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 5-11 Rtm_Start_CpuLoadMeasurement

5.2.8 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 5-12 Rtm_Stop_CpuLoadMeasurement

5.2.9 Rtm_GetMeasurementItem

Prototype	

`Std_ReturnType Rtm_GetMeasurementItem(`
`  CONST(uint32, AUTOMATIC) measurementId,`
`  CONST(Rtm_ItemType, AUTOMATIC) itemType,`
`  P2VAR(uint32, AUTOMATIC, RTM_APPL_VAR) itemValuePtr)`	

Parameter	
`measurementId`	Defines the measurement object.
`itemType`	Defines which result of the measurement object is requested.
`itemValuePtr`	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. In case of overall CPU load, all requested items are interpreted in percent.
Return code	
`Std_ReturnType`	The function returns E_NOT_OK if an invalid parameter was passed. Else E_OK is returned.
Functional Description	
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. The measurement result has to be calculated in calling function because only raw data is stored in Rtm. The time has to be divided through the count.	

Particularities and Limitations
> This function is synchronous. > This function is not reentrant. > This function only supports overall CPU load to be requested. > In trace mode `Off` E_NOT_OK is returned and `itemValuePtr` is not changed.
Call context
> This function shall be called on task or interrupt level.

Table 5-13 Rtm_GetMeasurementItem

5.3 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	GetTaskID

Table 5-14 Services used by the RTM

5.4 Configurable Interfaces

5.4.1 Callback Functions

5.4.1.1 Rtm_EnterTaskFct

Prototype	
`void Rtm_EnterTaskFct(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function call shall be added by the integrator to the OS Enter Task Hooks in order to measure RTM net runtimes without interruption times of higher prior tasks.	
Particularities and Limitations	
> This function is only available if at least one measurement point type `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType` is set to `ExecutionTime_NonNested` or `ExecutionTime_Nested`. > This function should not be called if `Rtm_Schedule` is used.	

Call context
> Task context > Only during measurements

Table 5-15 Enter Task Callback

5.4.1.2 Rtm_LeaveTaskFct

Prototype	
`void Rtm_LeaveTaskFct(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This 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 prior tasks.	
Particularities and Limitations	
> This function is only available if at least one measurement point type `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType` is set to `ExecutionTime_NonNested` or `ExecutionTime_Nested`. > This function should not be called if `Rtm_Schedule` is used.	
Call context	
> Task context > Only during measurements	

Table 5-16 Leave Task Callback

5.4.1.3 Rtm_EnterIsrFct

Prototype	
`void Rtm_EnterIsrFct(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function call shall be added by the integrator to the OS Enter CAT2 ISR Hooks in order to measure RTM net runtimes without interruption times of higher prior interrupts. For CAT1 interrupts the ISR must call this function directly because the OS does not provide any hooks in this case.	

Particularities and Limitations
> This function is only available if at least one measurement point type `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType` is set to `ExecutionTime_NonNested` or `ExecutionTime_Nested`. > This function should not be called if `Rtm_Schedule` is used.
Call context
> Interrupt context > Only during measurements

Table 5-17 Enter ISR Callback

5.4.1.4 Rtm_LeaveIsrFct

Prototype	
`void Rtm_LeaveIsrFct(void)`	
Parameter	
`void`	N.A.
Return code	
`void`	N.A.
Functional Description	
This function call shall be added by the integrator to the OS Leave CAT2 ISR Hooks in order to measure RTM net runtimes without interruption times of higher prior interrupts. For CAT1 interrupts the ISR must call this function directly because the OS does not provide any hooks in this case.	
Particularities and Limitations	
> This function is only available if at least one measurement point type `/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType` is set to `ExecutionTime_NonNested` or `ExecutionTime_Nested`. > This function should not be called if `Rtm_Schedule` is used.	
Call context	
> Interrupt context > Only during measurements	

Table 5-18 Leave ISR Callback

5.4.1.5 Rtm_Schedule

Prototype	
`void Rtm_Schedule( Os_TraceThreadIdType FromThreadId, Os_TraceThreadIdType ToThreadId, CoreIdType 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. 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.	
Particularities and Limitations	
> This function is only available if at least one measurement point type /MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType is set to ExecutionTime_NonNested or ExecutionTime_Nested. > This function should not be called if the APIs Rtm_EnterTaskFct, Rtm_leaveTaskFct, Rtm_EnterIsrFct and Rtm_LeaveIsrFct are used.	
Call context	
> Interrupt or task context > Only during measurements	

5.4.2 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:

5.4.2.1 Rtm_<Measurement Name>_ThresholdCbk

Prototype	
void Rtm_<Measurement Name>_ThresholdCbk(Rtm_MeasurementTimestampType runtime)	
Parameter	
runtime	Current runtime of the associated measurement point.
Return code	
void	N.A.
Functional Description	
This function is called after the runtime of the associated measurement point 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 5-19 Threshold Callback

5.4.2.2 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 5-20 RTM_GET_TIME_MEASUREMENT_FCT

6 Configuration

6.1 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.

7 Limitations

7.1 Runtime impact

To minimize the impact of RTM's own code to the runtime behavior of the ECU, all measurement points 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: 0)).

7.2 Measurement success

Performing measurement with RTM's frontend means activating one or more measurement points. 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.

7.3 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.

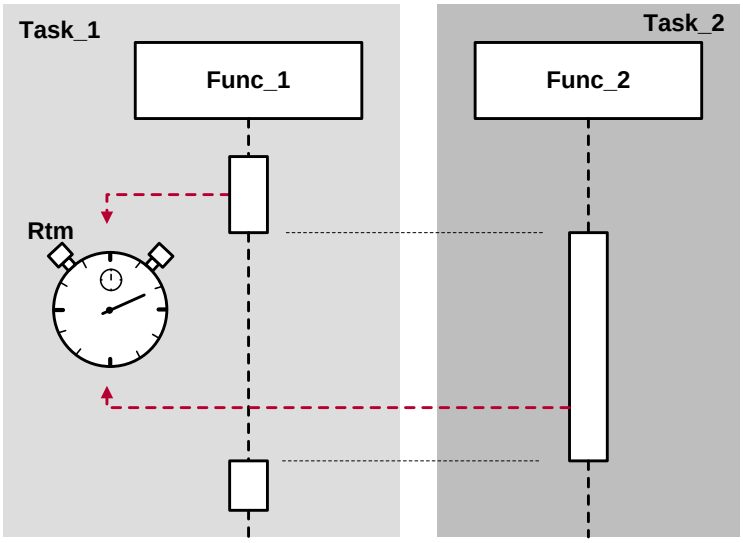


Figure 7-1 Inter-Task Measurement

7.4 Auto start Measurement

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

7.5 Net runtime measurement

The runtime of all functions that are called while the active runtime measurement continues is assigned to the currently active measurement. To receive the net runtime of a measurement point, the runtime that should not be assigned to the current measurement has to get an own measurement point. Additionally, net runtime measurement must be active. The functionality is shown in Figure 7-2. In this case, the runtime of the ISR should not be added to the runtime of Func_1. If the ISR occurs, the runtime measurement of Func_1 is stopped, its result is saved, and the runtime measurement of the ISR is started. After the execution of the ISR the result of Func_1 is restored and its runtime measurement is resumed.

Because the runtime measurements of Func_1 and ISR have to be active simultaneously, the net runtime measurement is only available for parallel measurement.

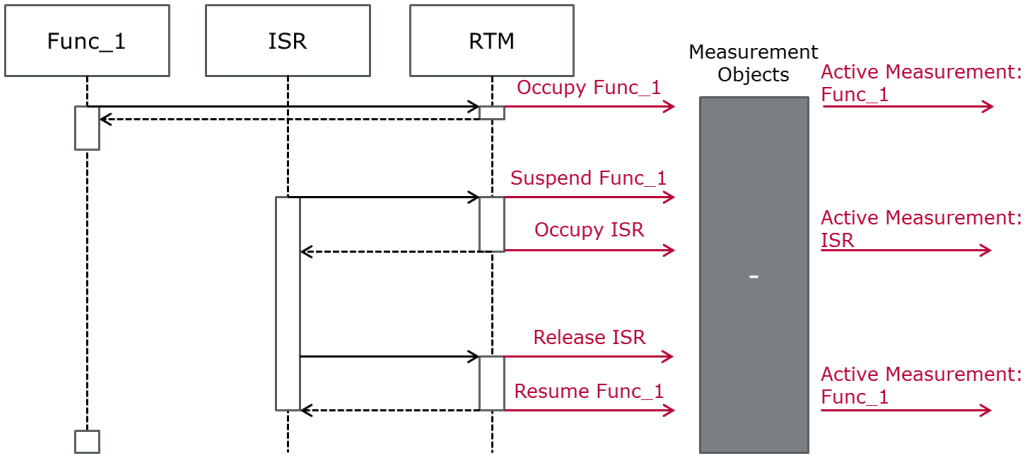


Figure 7-2 Net runtime support

7.6 Request measurement results

The function Rtm_GetMeasurementItem only supports the request of CPU load measurement results. In future, all measurement point results can be requested via this interface.

7.7 Report for Timing-Architects™

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

Additionally it is currently not supported to generate the runnable measurement points in the RTE hooks. They have to be implemented manually.

7.8 Limitations for multi core

- The measurement point types (`/MICROSAR/Rtm/RtmMeasurementPoint/RtmMeasurementType`) `ExecutionTime_NonNested` or `ExecutionTime_Nested` are not supported.
- 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.

8 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.

8.1 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 measurement points.
Runtime configuration of measurement points. Single, multiple and all measurement points can be de-/activation at once.
Result requesting at runtime.
En-/Disabling Debug output at runtime.

Table 8-1 Supported Features of RtmCan

8.1.1 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 8-2 Available Measurement Modes

8.2 RtmCan states

The states of RtmCan are shown in Figure 8-1. The RtmCan is automatically initialized. After initialization, the RtmCan is in state `RTMCAN_STATE_WAITFORACTION`. Within this state almost all services are available.

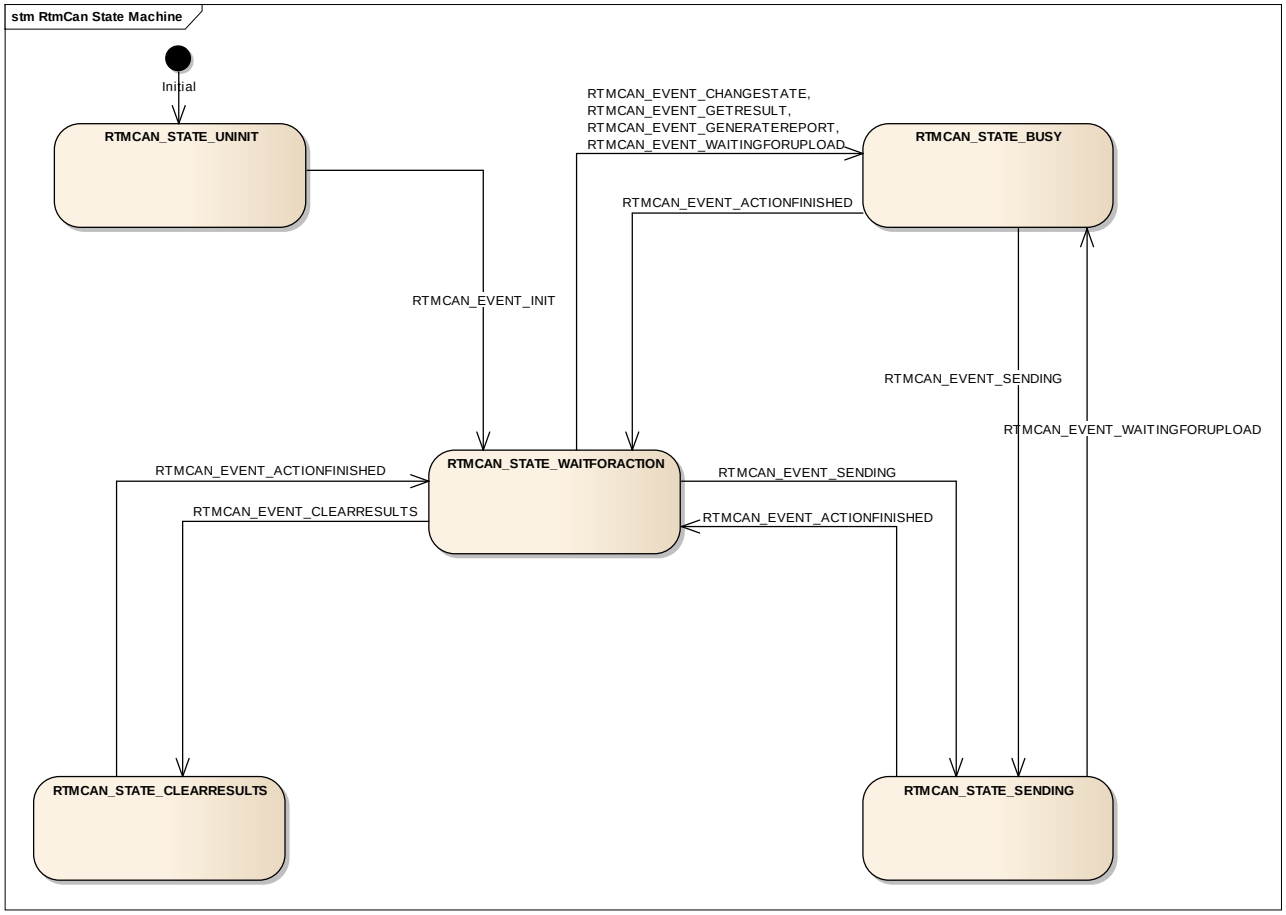


Figure 8-1 RtmCan state machine

8.2.1 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 measurement point
  - `RtmCan_GetMPResultByName`
  - `RtmCan_GetMPResultByID`
- Change state of measurement points
  - `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`

8.3 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.

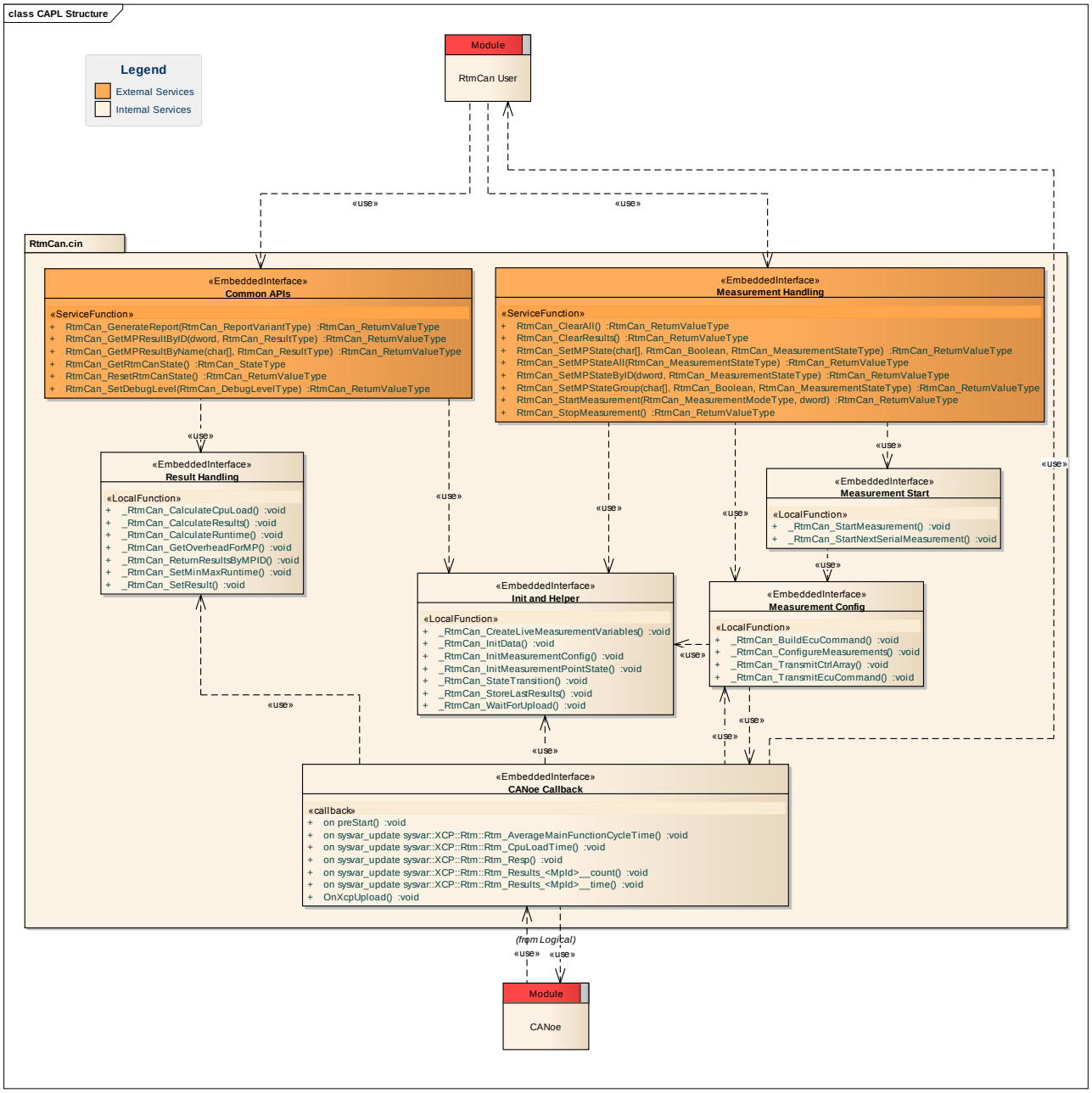


Figure 8-2 RtmCan.cin architecture

The detailed description of RtmCan's external APIs can be found in chapter 8.5.

8.4 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_MeasurementStateType	enum	State of the measurement points.	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_MU
			LTICORE_SYSTEM
			RTMCAN_E_NOT_AVAILABLE_WITH_16BIT_COUNTER
			RTMCAN_E_NOT_SUPPORTED
RtmCan_ReportVariantType	enum	The report variants.	RTMCAN_REPORT_ALL
			RTMCAN_REPORT_OVERALL_CSV
			RTMCAN_REPORT_RUNNABLE_CSV
RtmCan_DebugLevelType	enum	The debug level.	RTMCAN_DEBUGLEVEL_OFF
			RTMCAN_DEBUGLEVEL_ERROR

Type Name	C-Type	Description	Value Range
			RTMCAN_DEBUGLEVEL_WARNING
			RTMCAN_DEBUGLEVEL_INFO

Table 8-3 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 measurement point.	Max. 50 bytes.
Group	char array	The group name of the measurement point.	Max. 50 bytes.
DisableInterrupts	enum RtmCan_Boolean	Defines if flag RtmDisableInterrupts is set for this measurement point.	RTMCAN_FALSE
			RTMCAN_TRUE
IsRunnableMP	enum RtmCan_Boolean	Defines if measurement point is part of Runnable group.	RTMCAN_FALSE
			RTMCAN_TRUE
AssignedToCore	dword	Defines to which core the measurement point 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	0

Struct Element Name	C-Type	Description	Value Range
		start/stop.	4294967295
MeasurementRuntime	double	Resulting runtime of this measurement point in µs.	-1.7E +/- 308
			1.7E +/- 308
MeasurementCpuLoad_PerCent	double	Resulting CPU load of this measurement point in %.	-1.7E +/- 308
			1.7E +/- 308
MeasurementCycleTimeCorrection	int	Measurement correction value.	-32768
			+32767

Table 8-4 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 measurement point.	Max. 50 bytes.
RtmCan_Result_Group	char array	The group name of the measurement point.	Max. 50 bytes.
RtmCan_Result_MinRuntime_us	double	Minimum runtime of this measurement point in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_MaxRuntime_us	double	Maximum runtime of this measurement point in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_AverageRuntime_us	double	Average runtime of this measurement point in µs.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_NumberOfExecution	dword	Number of executions of this measurement point.	0
			4294967295
RtmCan_Result_CpuLoad	double	Average CPU load of this measurement point.	-1.7E +/- 308
			1.7E +/- 308
RtmCan_Result_AssignedToCore	dword	The core the measurement point is assigned to. If it is not assigned, this value is set to RTMCAN_NUMBER_OF_CORES.	0
			4294967295
RtmCan_Result_MeasurementDuration_us	qword	The overall measurement duration while this measurement	0
			18446744073709551615

Struct Element Name	C-Type	Description	Value Range
		point was active.	

Table 8-5 RtmCan_ResultType

8.5 Services provided by RtmCan (CAPL)

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

8.5.1 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 8-6 RtmCan_GetRtmCanState

8.5.2 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 measurement points. 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 8-7 RtmCan_GenerateReport

8.5.3 RtmCan_GetMPResultByName

Prototype	
`enum RtmCan_ReturnValueType RtmCan_GetMPResultByName(char measPointName[], struct RtmCan_ResultType result)`	
Parameter	
`measPointName`	The name of the measurement point for which the measurement results have to be returned.
`result`	The structure containing all measurement results of the requested measurement point.
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 measurement point name does not exist in the current configuration.
Functional Description	
This function returns a structure containing alle measurement results of the requested measurement point.	
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 8-8 RtmCan_GetMPResultByName

8.5.4 RtmCan_GetMPResultByID

Prototype	
`enum RtmCan_ReturnValueType RtmCan_GetMPResultByID(dword measPointID[], struct RtmCan_ResultType result)`	
Parameter	
`measPointID`	The identifier of the measurement point for which the measurement results have to be returned.
`result`	The structure containing all measurement results of the requested measurement point.
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 measurement point name does not exist in the current configuration.
Functional Description	
This function returns a structure containing alle measurement results of the requested measurement point.	
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 8-9 RtmCan_GetMPResultByID

8.5.5 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 8-10 RtmCan_ResetRtmCanState

8.5.6 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: The clear requested was executed successfully.
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 8-11 RtmCan_SetDebugLevel

8.5.7 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 measurement points 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 8-12 RtmCan_ClearResults

8.5.8 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 measurement points 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 8-13 RtmCan_ClearAll

8.5.9 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 measurement point until last measurement point was executed. Results are updated after each measurement point's measurement end. - RTMCAN_MODE_PARALLEL: Executes measurement for all active measurement points simultaneously. Results are updated after measurement end. - RTMCAN_MODE_LIVE: Executes measurement for all active measurement points 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 clear 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 8-14 RtmCan_StartMeasurement

8.5.10 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 8-15 RtmCan_StopMeasurement

8.5.11 RtmCan_SetMPStateAll

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPStateAll(enum RtmCan_MeasurementStateType state)`	
Parameter	
`state`	The new state of all measurement points. Possible values: - RTMCAN_MP_STATE_INACTIVE: All measurement points are deactivated. - RTMCAN_MP_STATE_ACTIVE: All measurement points 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 measurement points.	
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 8-16 RtmCan_SetMPStateAll

8.5.12 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 measurement points that are part of this group change their state to new value.
`equal`	Possible values: - RTMCAN_FALSE: Only change states of measurement points within the measurement group `measPointGroupName`. - RTMCAN_TRUE: Change states of all measurement points that are in a group containing the string of `measPointGroupName`.
`state`	The new state of all matching measurement points. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching measurement points are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching measurement points 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 measurement point is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of all measurement points 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 8-17 RtmCan_SetMPStateGroup

8.5.13 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 measurement points that are part of this group change their state to new value.
`equal`	Possible values: - RTMCAN_FALSE: Only change states of measurement points within the measurement group `measPointName`. - RTMCAN_TRUE: Change states of all measurement points that are in a group containing the string of `measPointName`.
`state`	The new state of all matching measurement points. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching measurement points are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching measurement points 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 measurement point is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of one measurement point 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 8-18 RtmCan_SetMPState

8.5.14 RtmCan_SetMPStateByID

Prototype	
`enum RtmCan_ReturnValueType RtmCan_SetMPStateByID(dword measPointID, enum RtmCan_MeasurementStateType state)`	
Parameter	
`measPointName`	Defines the measurement group name. All measurement points that are part of this group change their state to new value.
`state`	The new state of all matching measurement points. Possible values: - RTMCAN_MP_STATE_INACTIVE: All matching measurement points are deactivated. - RTMCAN_MP_STATE_ACTIVE: All matching measurement points 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 measurement point is part of the specified measurement group.
Functional Description	
This function sets the state (active/inactive) of one measurement point 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 8-19 RtmCan_SetMPStateByID

8.6 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.

8.6.1 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 8-20 Appl_RtmCanMeasurementFinished

8.6.2 Appl_RtmCanRespReceived

Prototype	
`void Appl_RtmCanRespReceived(enum RtmCan_StateType originState)`	
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 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 8-21 Appl_RtmCanRespReceived

8.7 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 measurement points
      retVal = RtmCan_SetMPStateAll(RTMCAN_MP_STATE_ACTIVE);

      if (retVal != RTMCAN_E_OK)
        write("RtmCan failed to set all measurement points 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 set al measurement points active.");
    }
  }
}

// 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 measurement point.
  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 measurement point.");
}
```

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 measurement points 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 measurement point is requested by calling `RtmCan_GetMPResultByID`. The returned result is printed to Write window of CANoe.

Pressing the button again, repeats the program.

8.7.1 Integrate the RtmCan_TestApplication.can file in CANoe

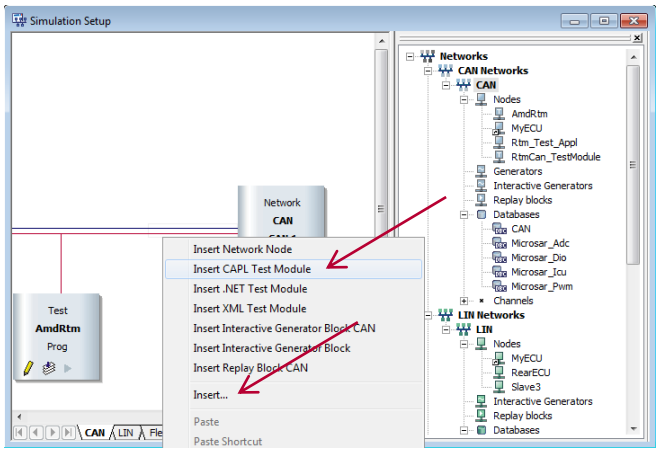
To create a new CAPL file in CANoe there is an easy way.

For this purpose open the Simulation Setup in the Simulation Ribbon. Switch to the network note 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.

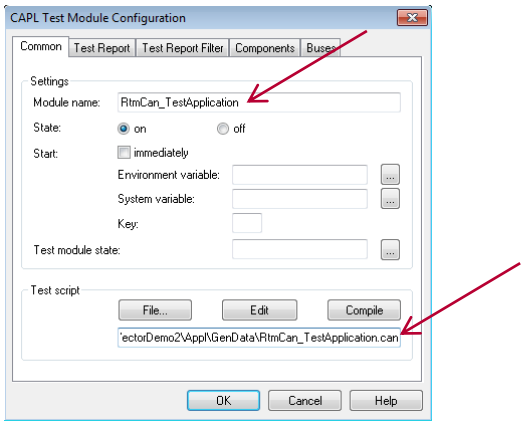


> Right click the new test node and choose **Configuration…**

> 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.

9 Glossary and Abbreviations

9.1 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 ResponseTime, ExecutionTime_NonNested and ExecutionTime_Nested.

Table 9-1 Glossary

9.2 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
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 9-2 Abbreviations

10 Contact

Visit our website for more information on

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

www.vector.com