MICROSAR WDGM

Technical Reference

Version 1.2.1

Authors	Christian Leder, Daniel Richter
Status	Released

Document Information

History

Author	Date	Version	Remarks
Daniel Richter, Christian Leder	2016-02-12	1.0.0	First version of the migrated WdgM Technical Reference
Christian Leder	2016-07-13	1.1.0	Update after introduction of native CFG5 generator
Christian Leder	2017-03-01	1.2.0	Mode Port functionality added Timebase source OsCounter added
Christian Leder	2017-07-25	1.2.1	Hint about initialization within a safety-related system added in 3.2 Initialization

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_WatchdogManager.pdf	V2.0.0
[2]	AUTOSAR	AUTOSAR_SWS_WatchdogInterface.pdf	V2.3.0
[3]	AUTOSAR	AUTOSAR_SWS_WatchdogDriver.pdf	V2.3.0
[4]	Vector Informatik	TechnicalReference_WdgIf.pdf	V1.0.0
[5]	Vector Informatik	Safety Manual	
[6]	ISO	Road vehicles – Functional safety	ISO 26262-1:2011(E)
[7]	AUTOSAR	AUTOSAR_TR_BSWModuleList.pdf	V1.4.0

> **! 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

Figures

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	Migration of the WdgM to Vector Informatik GmbH
2.00	Introduction of native CFG5 generator
2.01	Support mode ports and OsCounters as timebase

Table 1-1   Component history

2   Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module WdgM as specified in [1].

Supported AUTOSAR Release*:	4.0.1	
Supported Configuration Variants:	pre-compile	
Vendor ID:	WDGM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	WDGM_MODULE_ID	13 decimal (according to ref. [7])

* For the detailed functional specification please also refer to the corresponding AUTOSAR SWS.

The Watchdog (Wdg) Stack provides software modules to monitor the correct functioning of safety-relevant activities in systems with software modules of mixed criticality, such as

> newly developed safety-related functions,

> legacy functions, and

> basic software.

The Wdg Stack is designed to be used in automotive ECUs.

The Wdg Stack has three software modules:

> Watchdog Manager (WdgM)

> Watchdog Interface (WdgIf)

> Watchdog Driver (Wdg)

The WdgM can run on single-core and multi-core systems.

This user manual describes the WdgM, which is an AUTOSAR basic software module that is part of the AUTOSAR service layer. The WdgM checks the logical program flow and temporal behavior of the program flow of safety-relevant functions. Safety-relevant functions use checkpoint calls to send life signs to the WdgM. Internal or external watchdog hardware is used independently from the system CPU to monitor

> if the system is still alive,

> if the system functions properly, and

> if the system shows the correct temporal behavior and logical program flow.

The WdgM was developed according to AUTOSAR version 4.0.1 [1]. The WdgM is compatible with this AUTOSAR version, but not fully compliant. For the deviations, see section Deviations from the AUTOSAR 4.0.1 Watchdog Manager. If the WdgM is used in safety-related systems with AUTOSAR 4.0.1 or another version, all requirements described in the Safety Manual [5] must be fulfilled.

This user manual does not cover safety-related topics. For safety-related requirements for the integration and the application of the WdgM, refer to the Safety Manual [5].

2.1 Architecture Overview

The following figure shows where the WdgM is located in the AUTOSAR architecture.
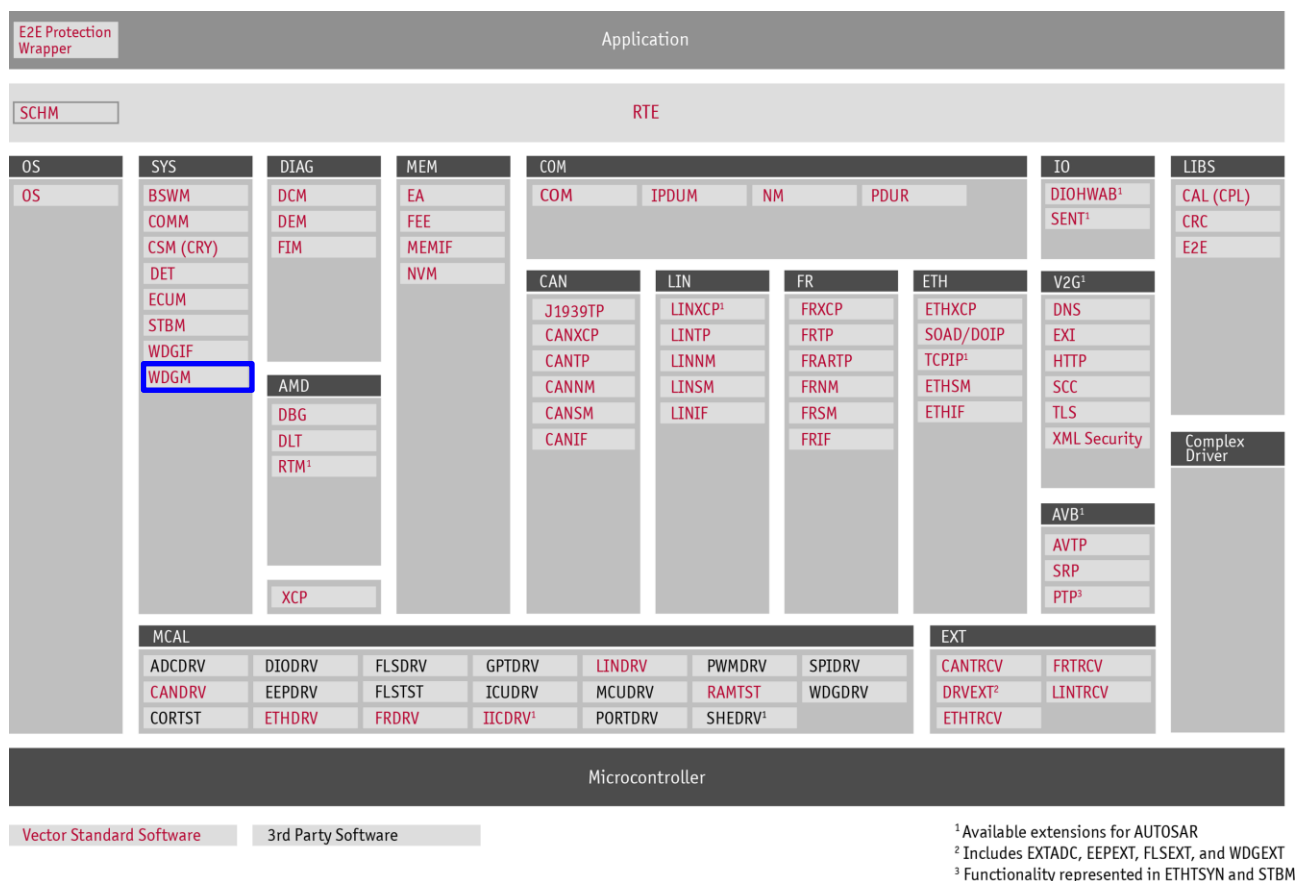
Figure 2-1   AUTOSAR 4.x Architecture Overview

The WdgM Stack consists of the hardware-independent modules Watchdog Manager (blue rectangle) and Watchdog Interface and a hardware-dependent module Watchdog Driver.

Figure 2-2 shows the WdgM Stack with its modules in an AUTOSAR environment.
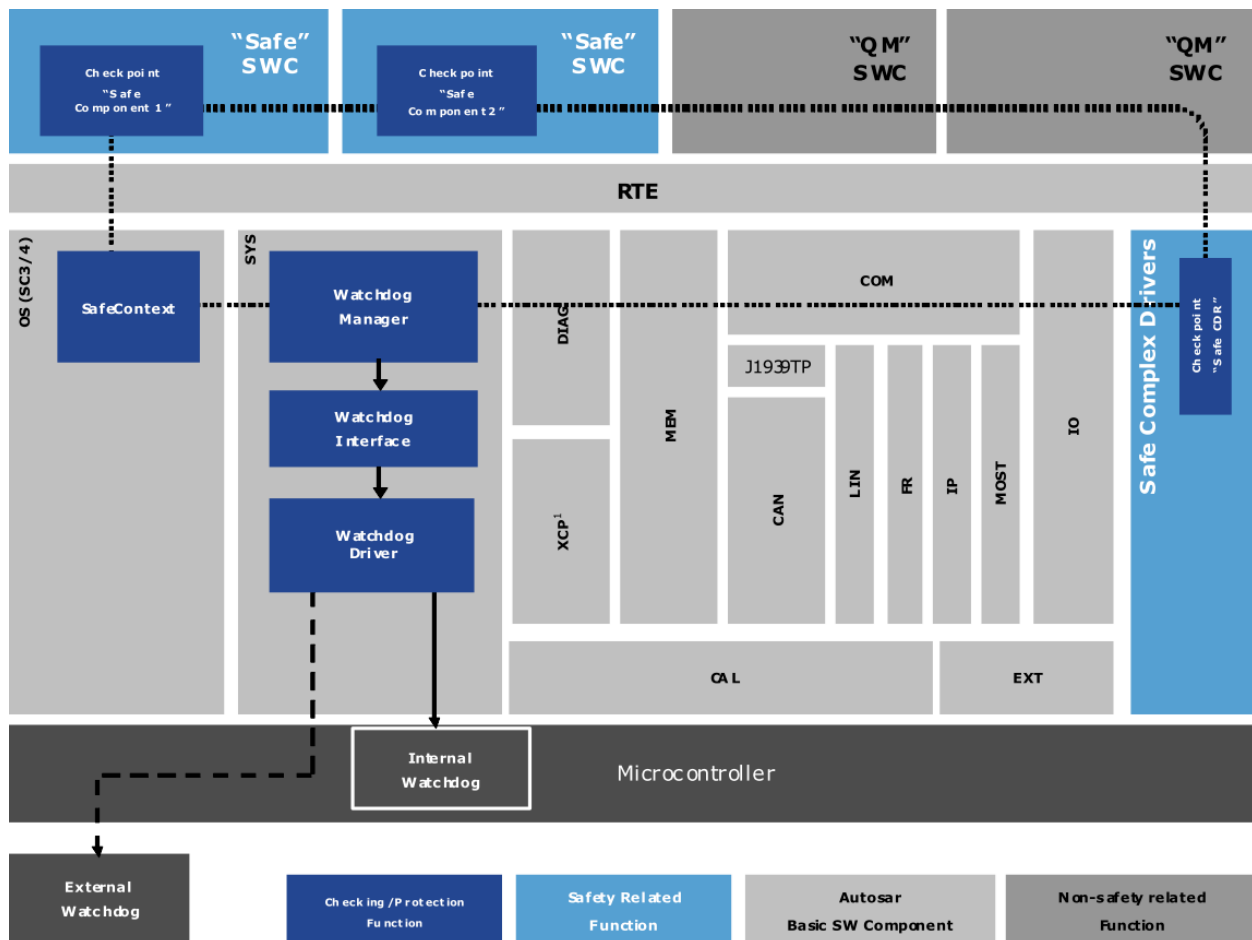
Figure 2-2   Watchdog Manager Stack in an AUTOSAR environment

The WdgM controls, through the WdgIf and the Wdg, the hardware-implemented watchdogs, which can be one or more internal or external watchdog devices.

Note
A watchdog device requires a hardware-dependent Wdg driver.

Figure 2-3 shows the layered structure of the WdgM Stack. The attached watchdog device can be internal or external.

Applications

BSW's
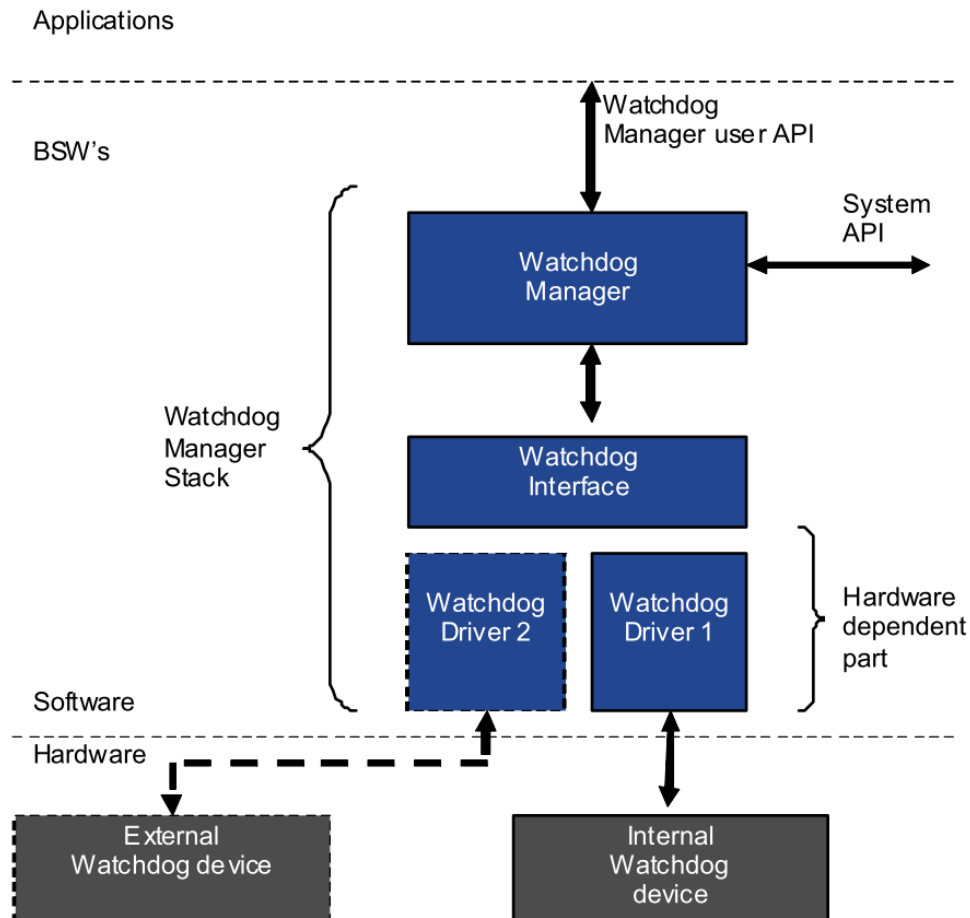
Figure 2-3   Layered structure of the Watchdog Manager

The WdgM monitors the program flow and timing constraints of so-called supervised entities (SE). The SEs are software entities (like application software) that are supervised by the WdgM. When the WdgM detects a violation of the preconfigured program flow or the timing constraints, it takes a number of configurable actions to log that violation and/or go to a safe state (for details, see Section Basic Functionality of the WdgM).

2.2   Use Cases

The WdgM monitors the user software at runtime and compares the preconfigured logical and temporal constraints with the actual logical and temporal behavior. The WdgM can monitor the following violations:

> timing violation (checked by deadline supervision and alive supervision)

> program flow violation (checked by logical monitoring)

The WdgM periodically triggers the watchdog device through its interface (WdgIf) and driver layer (Wdg). When the WdgM detects a fault in the program flow or timing then it stops the watchdog triggering, or it initiates a reset of the microcontroller immediately or after a delay, depending on the WdgM configuration.

The WdgM monitors the following software and hardware faults:

> The supervised entity is executed, but the execution was not requested.

> The supervised entity was not executed, but the execution was requested.

> The execution of the supervised entity started too early or too late.

> The execution time of a supervised entity or part of a supervised entity or many supervised entities is longer or shorter than expected.

> The program flow of a supervised entity or part of a supervised entity or many supervised entities differs from expected program flow.

The reaction of the WdgM to detected faults can be configured as follows:

> WdgM sends information about the detected fault.

> WdgM initiates a reset of the microcontroller after a watchdog timeout.

> WdgM initiates an immediate reset of the microcontroller.

2.3 Basic Functionality of the WdgM

As described in AUTOSAR [1], the WdgM is a basic software module that monitors the program flow of supervised entities (SE).

2.3.1 Supervised Entity and Program Flow Supervision

A supervised entity is a software part that is monitored by the WdgM. There is no fixed relationship between supervised entities and the architectural building blocks in AUTOSAR.

The checkpoints mark important steps during the execution of an algorithm. At the checkpoint, a supervised entity calls the function `WdgM_CheckpointReached()` directly (if no runtime environment is present) or with a wrapper function (if a runtime environment is present) being provided by the runtime environment. The checkpoints are connected by transitions. Local transitions bind Checkpoints to a closed graph. These graphs represent the program flow.

The WdgM knows which program flow is correct and decides if a supervised entity behaves as expected or violates the predefined rules.

The question of how to identify the checkpoints for an algorithm is a trade-off between performance and code block size per checkpoint:

> The more checkpoints an algorithm has, the better is the representation of the code structure. But this has an adverse effect on performance.

> However, if an algorithm has only a few checkpoints, then there are code segments and program flow branches that are not represented. In this case, performance will be better, but not everything will be monitored.

A supervised entity can represent an algorithm, a function, or – in the case of an operating system – an entire task. In the AUTOSAR definition, a supervised entity can be distributed over more than one task or application. There can be several supervised entities for the same task. However, the WdgM implementation does not support the distribution of one supervised entity over more than one task or application when they run in different contexts. The WdgM expects that at least one supervised entity and at least one checkpoint are defined.

Figure 2-4 shows the example of a simple supervised entity called `temperature_control`:

> Supervised entity `temperature_control` has six checkpoints (illustrated by oval boxes), which are connected by directed transitions (illustrated by arrows).

> As can be seen in Figure 2-4, it is possible to reach the checkpoint `temperature_needs_correction` after the checkpoint `read_temperature`.

> However, reaching the `checkpoint heater_adjusted_successfully` after the checkpoint `read_temperature` would be a violation of the program flow.
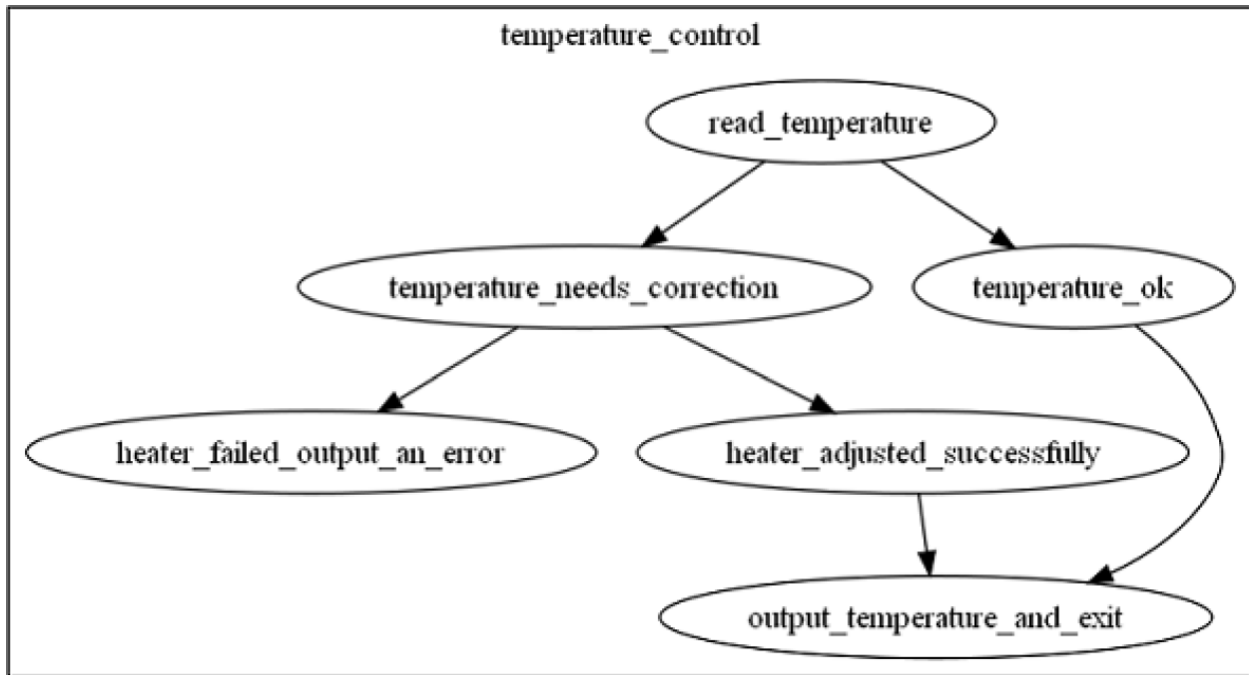
Figure 2-4   Example of a simple supervised entity with a control flow

2.3.2   Program Flow Supervision

> **Note**
> Program Flow Supervision can only be used if the WDG add-on for program flow and deadline supervision is licensed.
>
> ("WdgM_ProgramFlowAndDeadlineMonitoring")

Control (program) flow supervision is highly recommended by ISO 26262-6 (7.4.14). Apart from its main feature, which is to detect logical errors in the monitored algorithms, program flow supervision increases the probability of detecting illegal program counter jumps within the whole system.

In addition to the specification by AUTOSAR, it is possible to tolerate program flow violations within a supervised entity for a certain amount of supervision cycles. It is possible to define a program flow reference cycle (a multiple of the WdgM supervision cycle) and a tolerance, which is a number of program flow reference cycles, during which program flow violations should be tolerated for the supervised entity. If a program flow violation is detected for more program flow reference cycles than the defined tolerance, then the supervised entity changes its status from FAILED to EXPIRED.

The necessary configuration parameters to tolerate program flow violations of a supervised entity are:

> WdgMFailedProgramFlowRefCycleTol: This parameter contains the acceptable amount of program flow violations for this supervised entity.

> **WdgMProgramFlowReferenceCycle**: This parameter contains the amount of supervision cycles to be used as reference by the program flow supervisions of this supervised entity.

> **Note**
> The program flow reference cycle for a supervised entity starts with the first detected program flow violation and not with the WdgM startup. Hence, the first program flow reference cycle starts with the transition of the supervised entity from status OK to FAILED. If no program flow violation is detected for a whole program flow reference cycle within the tolerance then the supervised entity recovers and changes its status from FAILED to OK. Otherwise, if the tolerance is exhausted and the program flow violations continue, then the supervised entity changes its status to EXPIRED. It can be said that the program flow reference cycle is processed only during the status FAILED – it starts with the first detected program flow violation. The program flow reference cycle is restarted with each following transition from OK to FAILED, and it is not processed during the status OK, EXPIRED or DEACTIVATED.

2.3.3 Deadline Supervision

> **Note**
> Deadline Supervision can only be used if the WDG add-on for program flow and deadline supervision is licensed.
> ("WdgM_ProgramFlowAndDeadlineMonitoring")

The main purpose of deadline supervision is to check the temporal, dynamic behavior of the supervised entity. However, it would also strongly increase the probability of detecting random jumps or irregular updates of the timebase tick counter, which might otherwise degrade system integrity without being discovered.

The temporal behavior of the supervised entities can be monitored by assigning deadlines to transitions.

> A deadline is defined through a maximum deadline (parameter `WdgMDeadlineMax`) and a minimum deadline (parameter `WdgMDeadlineMin`). The destination checkpoint of a transition should not be reached before the minimum time or after the maximum time after which the source checkpoint of that transition was reached. Otherwise the WdgM will detect a deadline violation. Apart from a maximum deadline time it is strongly recommended to use a minimum deadline time as well, where applicable. This allows discovering timebase tick counter errors implicitly. Deadlines are good for discovering crashed tasks or infinite loops. If the destination checkpoint is never reached because the task ended with an error or is stuck in a loop, this would cause a deadline violation.

> A deadline is assigned to an already defined transition by specifying the same source and destination checkpoints as for the transition. The corresponding deadline parameters are `WdgMDeadlineStartRef` and `WdgMDeadlineStopRef`.

> For local transitions, the source and destination checkpoints belong to the same supervised entity.

> For global transitions, the source and destination checkpoints belong to different supervised entities.

An example of a supervised entity with deadlines defined for its transitions is given below. The first deadline is defined to have a minimum of 0 and a maximum of 2 (seconds). Hence, CP1 must be reached no later than 2 seconds after CP0. The second deadline implies that CP2 must be reached no earlier than 1 and no later than 3 seconds after CP1. Otherwise a deadline violation will be detected.
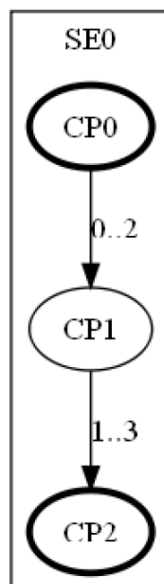
Figure 2-5   Example of a simple supervised entity with deadlines

Note
Deadline violation is detected

> when the next checkpoint is reached outside the defined deadline or

> within the `WdgM_MainFunction()` if the next checkpoint is not reached at all (or has not been reached yet) and the maximum deadline has already expired

A slightly more complex situation is when several transitions go out of the same checkpoint. In this case, deadline violations are detected in the same manner when the next checkpoint is reached outside the defined deadlines. However, if none of the next checkpoints is reached, the `WdgM_MainFunction()` detects a deadline violation only after the maximum of maximum deadlines of all outgoing transitions has elapsed, which is

seconds after reaching CP0, shown in Figure 2-6. If the program gets stuck after CP0, the deadline violation is detected within the next main function that is executed not earlier than 5 seconds after reaching CP0.
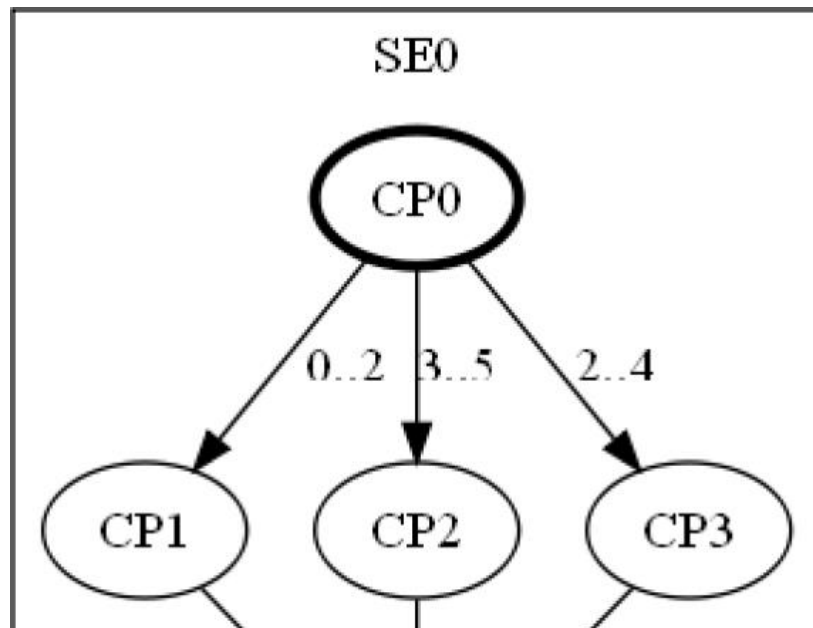
Figure 2-6   Example of multiple outgoing transitions with deadlines

A special case is a hybrid situation when some of the outgoing transitions have deadlines and others do not. In this case, the main function detects a deadline violation if none of the next checkpoints is reached within the maximum of configured deadlines in order to detect blocked supervised entities. No deadline violation will be detected after the maximum has expired, however, if the checkpoint without deadline is reached before the main function. If none of the CP1, CP2 is reached after CP0 (Figure 2-7), then the next `WdgM_MainFunction()` (executed at least 2 seconds after CP0 is reached) detects a deadline violation. If, however, CP1 is reached after 2 seconds, but before the next `WdgM_MainFunction()`, no deadline violation would be detected.

> **Note**
> To avoid this ambiguous situation it is a good practice to define deadlines for all outgoing transitions of a checkpoint (or for none of them).
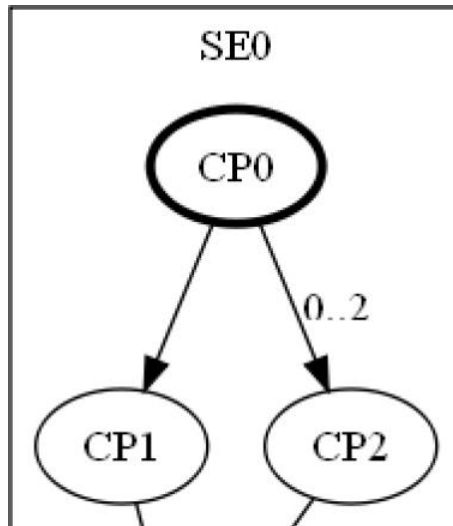
Figure 2-7   Example of a the case where only one of several outgoing transitions has a deadline

The rules for deadline violation detection also apply to global transitions or to the case of local transitions mixed with global transitions at a checkpoint.

In addition to the specification by AUTOSAR, it is possible to tolerate also deadline violations within a supervised entity for a certain amount of supervision cycles. It is possible define a deadline reference cycle (a multiple of the WdgM supervision cycle) and a tolerance, which is a number of deadline reference cycles, during which deadline violations should be tolerated for the supervised entity. If a deadline violation is detected for more deadline reference cycles than the defined tolerance, then the supervised entity changes its status from `FAILED` to `EXPIRED`.

The necessary configuration parameters to tolerate deadline violations of a supervised entity are:

> `WdgMFailedDeadlineRefCycleTol`: This parameter contains the acceptable amount of violated deadlines for this supervised entity.

> `WdgMDeadlineReferenceCycle`: This parameter contains the amount of supervision cycles to be used as reference by the deadline supervisions of this supervised entity.

Note

The deadline reference cycle for a supervised entity starts with the first detected deadline violation and not with the WdgM start up. Hence, the first deadline reference cycle starts with the transition of the supervised entity from the status `OK` to `FAILED`. If no deadline violation is detected for a whole deadline reference cycle within the tolerance, then the supervised entity recovers and changes its status from `FAILED` to `OK`. Otherwise, if the tolerance is exhausted and the deadline violations continue, then the supervised entity changes its status to `EXPIRED`. It can be said that the deadline reference cycle is processed only during the status `FAILED` – it starts with the first detected deadline violation. The deadline reference cycle is restarted with each following transition from `OK` to `FAILED`, and it is not processed during the status `OK`, `EXPIRED` or `DEACTIVATED`.

2.3.4 Alive Supervision

Aliveness monitors the frequency of hits of checkpoints. For example, the algorithm could expect a sensor to report its measurements on a regular basis, and a certain task needs to process this data periodically. If a task stops reporting (alive sign is lost or too infrequent) or starts reporting too often, then the aliveness of that task is violated.

Alive supervision is associated with a checkpoint in a supervised entity. If you need to monitor only the frequency with which a task is called, you can define a supervised entity that contains only one checkpoint with the corresponding aliveness parameters.

Note

Irregular calls of the WdgM main function or the omission of calls of `WdgM_CheckPointReached()` would most likely result in aliveness violation. When alive supervision for a checkpoint is activated, then that checkpoint must be regularly called for the entire period during which the supervised entity is active, otherwise aliveness violation will be detected. In the first supervision cycle, the alive counter evaluation can be suppressed by the parameter `WdgMFirstCycleAliveCounterReset`.

It is important to consider which aliveness parameters are better for a specific situation. The example below shows how to choose the appropriate alive supervision parameters:

> `WdgMExpectedAliveIndications`:

  Defines how many alive indications (checkpoint reached calls) are expected within one supervision reference cycle.

> `WdgMSupervisionReferenceCycle`:

  Defines the supervision reference cycle length as a number of supervision cycles (`WdgMSupervisionCycle`).

> `WdgMMinMargin`:

Defines the lower tolerance of expected alive indications.

> `WdgMMaxMargin`:

Defines the upper tolerance of expected alive indications.

> Hence, the allowed number of indications is in the range
`[WdgMExpectedAliveIndications - WdgMMinMargin,`
`WdgMExpectedAliveIndications + WdgMMaxMargin]`

---

i

Note

In contrast to the deadline and program flow reference cycle the alive supervision cycle begins with the WdgM startup. The alive supervision in the very first cycle can be influenced by the parameter `WdgMFirstCycleAliveCounterReset`. This is because each alive counter is evaluated once per supervision reference cycle. This means that the supervision reference cycle is processed from the system startup on and during the status `OK` and `FAILED` of the corresponding supervised entity. If the supervised entity is in the status `EXPIRED`, then the supervision reference cycle is not needed anymore. If the supervised entity is in the status `DEACTIVATED`, then the supervision reference cycle is frozen. It is restarted if the supervised entity is activated again.

There are several ways for monitoring the task given in the example above. Below, one variant is given:

Set

> `WdgMExpectedAliveIndications=1`

> `WdgMSupervisionReferenceCycle=1`

> `WdgMMinMargin=1`

> `WdgMMaxMargin=0`

This means the WdgM should expect 1 or 0 (`WdgMExpectedAliveIndications - WdgMMinMargin`) occurrences within one supervised reference cycle, which is fixed to 20ms (which is one WdgM supervision cycle).
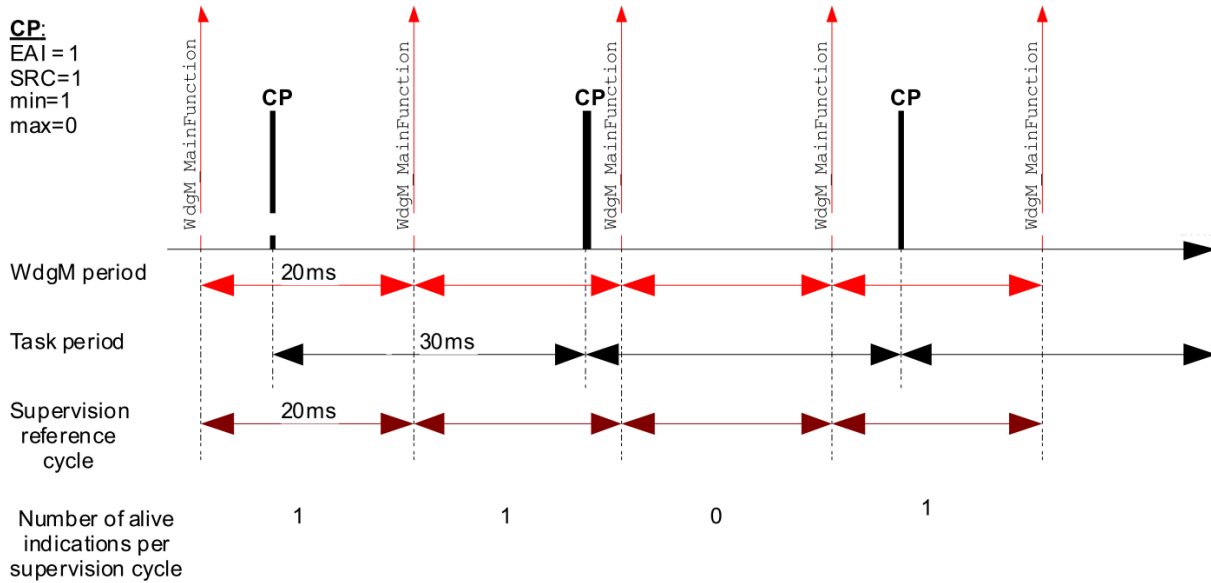
Figure 2-8   A task being monitored during one WdgM supervision cycle (20ms)

However, if the task stops being executed it will not be detected, because zero alive indications per supervised reference cycle are tolerated. Therefore, this choice of setting aliveness parameters is not very good.

Below, a second variant is given:

Set

> `WdgMExpectedAliveIndications=2`

> `WdgMSupervisionReferenceCycle=2`

> `WdgMMinMargin=1`

> `WdgMMaxMargin=0`

This means the WdgM should expect 1 or 2 alive indications within one supervised reference cycle, which is fixed to 40ms (and which is two WdgM supervision cycles).
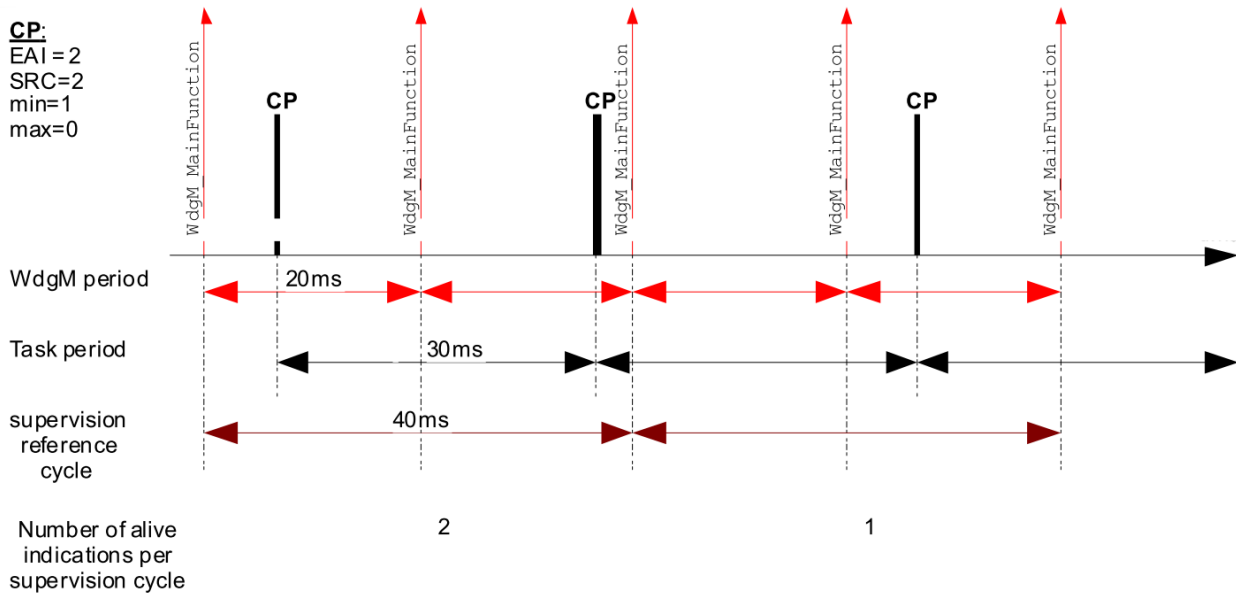
Figure 2-9 A task being monitored during two WdgM supervision cycles (40ms)

This configuration solves the problem of detecting the disappearance of the task. However, the reaction time for error detection doubles from 20 to 40ms.

A third variant would be to set the supervision reference cycle to the least common multiple of the WdgM supervision cycle and the task period. In the example given above this would be 60ms (three WdgM supervision cycles). In this case, we expect exactly 2 alive indications. Hence, the minimum and maximum margins are both 0.

Note
The task period and the WdgM supervision cycle must be synchronized and started with an offset to each other (e.g. scheduled in an operating system).

2.3.5 More Details on Checkpoints and Transitions

Every supervised entity has one initial checkpoint. The number of end checkpoints can be zero, one or more than one. If the supervised entity contains only one single checkpoint, then it should be both an initial and an end checkpoint. Local transitions are defined by their source and destination checkpoints, which must belong to the same supervised entity. Those local transitions are specified in the parameters WdgMInternalTransitionSourceRef and WdgMInternalTransitionDestRef.

After initialization of the WdgM, all supervised entities are passive.

Note
This has nothing to do with the supervised entity state WDGM_LOCAL_STATUS_DEACTIVATED.

A supervised entity becomes active when its local initial checkpoint has been called. In the example of the supervised entity `temperature_control` (see Section Supervised Entity and Program Flow Supervision and Figure 2-4), the initial checkpoint is read_temperature. Only if the supervised entity is active, its checkpoints (other than the initial checkpoint) may be reached, otherwise a program flow violation occurs. Reaching an end checkpoint, the supervised entity is set to passive state, and it can be activated again only through the initial checkpoint.

Reaching the initial checkpoint again after the supervised entity has been activated is a program flow violation.

Local reflexive transitions (from a checkpoint to itself) are allowed only if configured. The reflexive transitions cannot be defined for local initial or local end checkpoints.

Local initial checkpoints are not allowed to have local incoming transitions.

Local end checkpoints are not allowed to have local outgoing transitions.

2.3.6 Global Transitions

It is possible to represent program flow dependencies between supervised entities by using so-called global transitions. Global transitions are defined for the WdgM configuration by their source and destination checkpoints, which must belong to different supervised entities and which are specified by the parameters `WdgMExternalCheckpointInitialRef` and `WdgMExternalCheckpointFinalRef`. The end checkpoint of a supervised entity is usually connected to the initial checkpoint of another supervised entity, expressing a logical dependency between them. However, global transitions are allowed between any two checkpoints of any two supervised entities.

One must keep in mind several things when defining a global transition between two arbitrary checkpoints:

> If the source of the global transition is not a local end checkpoint, then the source entity will remain active. Program flow violation would occur if its initial checkpoint were reached again.

> If the destination checkpoint of the global transition is not a local initial checkpoint, the destination entity may not be active. Program flow violation would occur if a non-initial checkpoint of an inactive supervised entity were reached.

> Exactly one global initial checkpoint must be defined. The first global transition passed must have that checkpoint as a source.

> It is possible to define one or several global end checkpoints or none. Once the global end checkpoint served as a destination checkpoint of a global transition, no more global transitions are allowed (unless they are started with the global initial checkpoint).

Figure 2-10 shows a global transition between two supervised entities:

> The `pressure_sensor_task` gets the pressure value.

> The `control_pressure_task` calculates a reaction and reacts to the measured pressure. However, it can start only after the first task (`pressure_sensor_task`) has finished and after the pressure value has been obtained. This relation is shown by a global transition (see dotted arrow).

> Some transitions in Figure 2-10 have comments that show deadlines in milliseconds.

> Deadlines can also be defined for global transitions (see dotted arrow), where 1..5ms means that the second task (`control_pressure_task`) should start not later than 5ms, but not earlier than 1ms after the first task has finished.
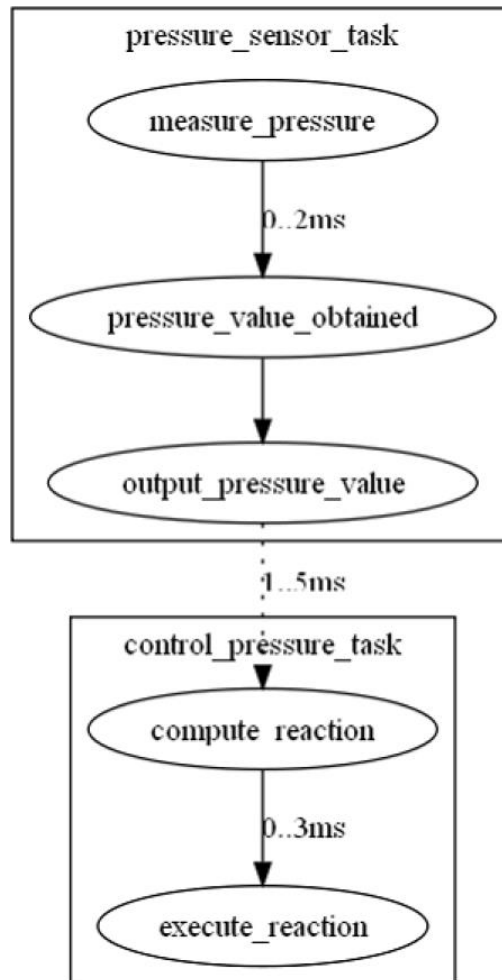
Figure 2-10   Global transition between two supervised entities

Note
Global transitions between supervised entities that are assigned to different processor cores are not supported.

2.3.7 Global Transitions and Program Flow

In general the, program flow does not differ between local and global transitions. But what seems intuitive for local transitions might not be so obvious for global transitions. This section gives examples that show the usage of local and global transitions with a focus on program flow split.

From the perspective of the WdgM, the program flow is the consecutive reaching of checkpoints. The start of each program flow must be a local initial checkpoint. The program flow propagates through local transitions within the boundaries of a supervised entity and through global transitions within the boundaries of the whole system. The program flow might eventually come to an end at a local end checkpoint, or never come to an end if a program flow loop occurs.

A very important feature is that it is not allowed to split the program flow. This means that the program flow is allowed to take only one transition at each checkpoint from which more than one local or global transition comes out.

2.3.7.1 Example of an Incorrect Global Transition Split

Figure 2-11 shows that after checkpoint cp0_1 the program flow must decide to take either the global transition cp1_0 or cp2_0. Reaching cp2_0 immediately after reaching cp1_0 would result in a program flow violation.
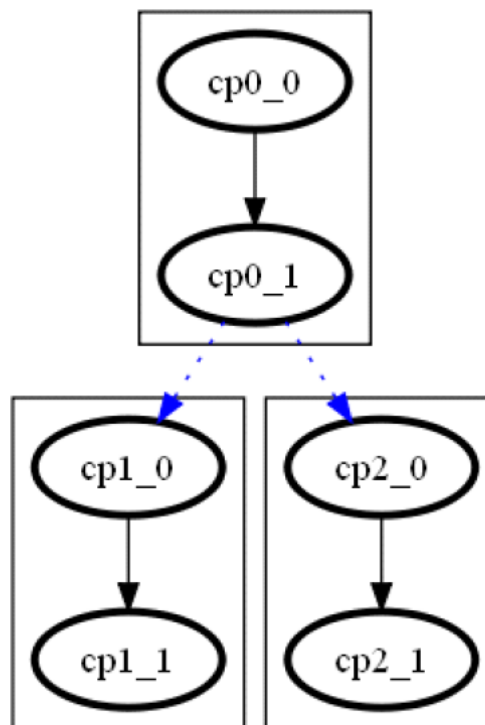
Figure 2-11   Incorrect global transition split

2.3.7.2 Example of an Incorrect Program Split in the Middle of an Entity

Figure 2-12 shows another example. Let us assume that the program flow reaches cp0_0 and then cp0_1. Afterward the program flow decides to take the global transition reaching cp1_0 instead of taking the local transition. Now, if the local transition took place afterward (by reaching cp0_2), a program flow violation would occur. However, cp0_2 can be reached via the global transition if the program flow comes from cp1_1.
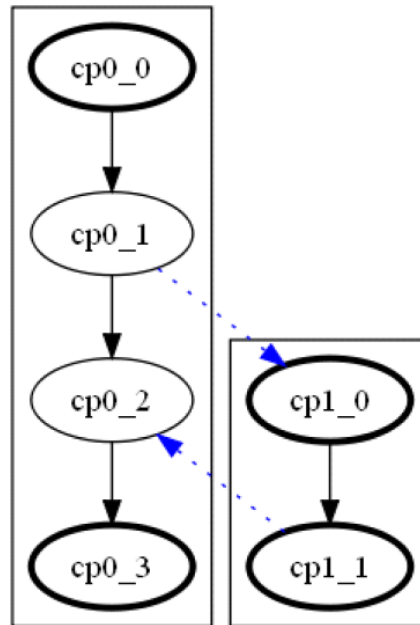
Figure 2-12   Incorrect program split in the middle of an entity

> **Note**
> It is easy to create configurations with complex global transitions that do not make much sense in a real system. For example, if "jumping out" of a supervised entity from a checkpoint that is not a local end checkpoint, one must keep in mind that this supervised entity is still active (local activity flag is still `true`), and it cannot be restarted by reaching its local initial checkpoint again. Thus, it is recommended to use global transitions carefully and let them start only at local end checkpoints of a supervised entity and end at a local initial checkpoint of some other entity. Exceptions to this must be analyzed thoroughly, with respect to the program flow and the local activity of both supervised entities.

2.3.8   WdgM Supervision Cycle

The supervision cycle is the time period in which the cyclic supervision algorithm is executed. At the end of each supervision cycle, the main function, `WdgM_MainFunction()`, is called. This function evaluates the checkpoint data gathered in the previous period and triggers the Watchdog if no violation has been detected. Function `WdgM_MainFunction()` also checks for violations depending on the reference cycle defined for the respective monitoring feature.

Example: If `WdgMProgramFlowReferenceCycle=3`, then the check for program flow violation is done in every third call of `WdgM_MainFunction()`.

The shorter this period and the reference cycles, the shorter the reaction time of the WdgM, but the more processor time is consumed.

> **Note**
> Aliveness supervision is strongly connected to this period. The expected number of alive indications for a certain checkpoint refers to the last supervision cycle (configurable for the checkpoint), which is expressed in the number of supervision cycles.

Figure 2-13 shows a time span with 3 supervision cycles. In each cycle, CP1 and CP2 are hit once. Once the WdgM main function is called, the window for the next watchdog trigger is defined by `WdgMTriggerWindowStart` and `WdgMTriggerConditionValue`.
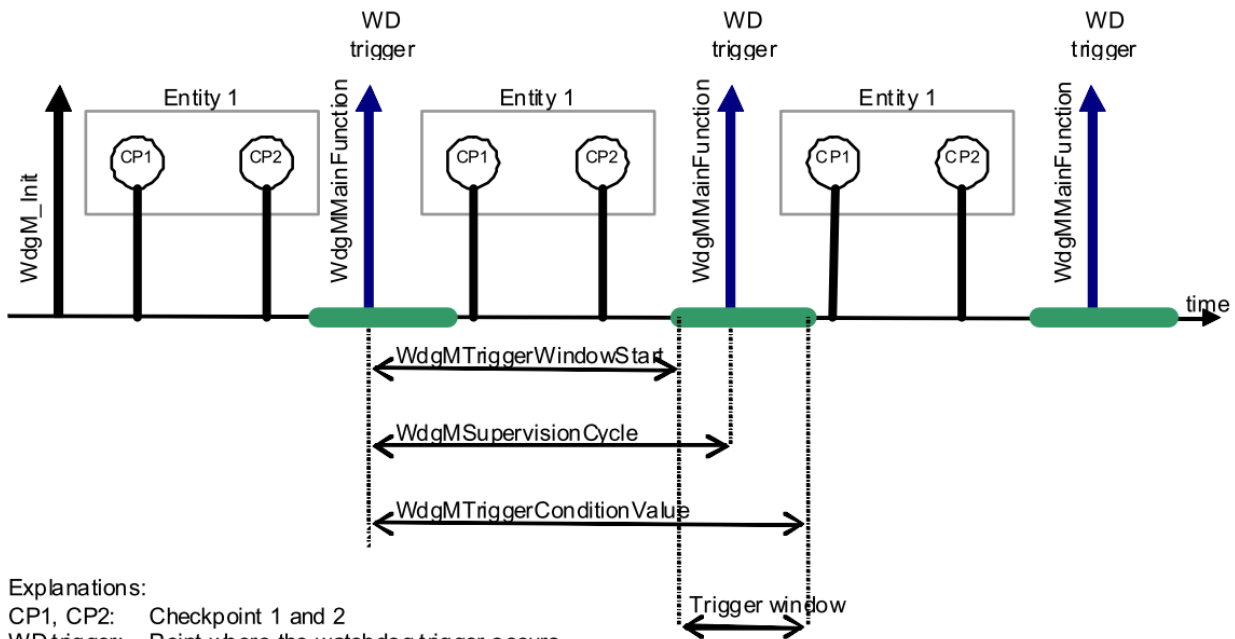
Explanations:
CP1, CP2:   Checkpoint 1 and 2
WD trigger: Point where the watchdog trigger occurs
Entity1:   Entity with two checkpoints
Green bar: Time window where re-triggering is allowed

Figure 2-13   WdgM supervision cycle

> **Note**
> For the scheduling of WdgM_MainFunction() calls, the integrator shall set
>
> > WdgMTicksPerSecond,
> >
> > WdgMSupervisionCycle,
> >
> > WdgMTriggerWindowStart (per Wdg Trigger Mode), and
> >
> > WdgMTriggerConditionValue (per Wdg Trigger Mode)
>
> according to the system requirements.

2.3.9 Fault Detection Time Evaluation

The WdgM distinguishes between the **fault detection time** and the **fault reaction time**.

> The fault detection time spans from the occurrence of an error to the point in time when that error is detected and communicated to the system (via DET or callback functions).

> The fault reaction time spans from the detection of an error to the actual system reset.

If a program flow violation or a deadline violation occurs, the source checkpoint and the destination checkpoint report to the WdgM when hit. At the end of the current supervision cycle, the WdgM main function, `WdgM_MainFunction()`, is called and the violation is detected (i.e. the configured destination checkpoint was hit too late or not at all) and communicated to the system.

If an alive counter violation occurs, it is also the main function that detects and communicates the violation at the end of the supervision reference cycle of the alive supervision.

The shortest fault detection and reaction time can be achieved by configuring an immediate reset. However, the time still depends on what occurs first in a supervision cycle, the fault or the hit of the checkpoint.

Note
The time from fault occurrence to the system's safety reaction is the sum of

> WdgM Fault Detection Time,

> WdgM Fault Reaction Time,

> WdgIf Fault Reaction Time and

> Wdg Fault Reaction Time.

The WdgM fault detection time is evaluated differently for the various monitoring features as shown in this section.

2.3.9.1 Alive Supervision Fault Detection Time

Assume that a fault occurs that leads to an alive counter violation. The WdgM fault detection time is the sum of the time spans

> from the fault to the scheduled call of the next checkpoint that is monitored and

> from this checkpoint to the next call of `WdgM_MainFunction()` with alive supervision.

Note
The multiple of supervision cycle that performs an alive supervision is defined by `WdgMSupervisionReferenceCycle`.

Note
If zero calls of a checkpoint within an alive supervision interval are allowed, then missing checkpoint calls cannot be detected. The fault detection time is then infinite.

The worst case for a given configuration happens when

> the time between the calls of `WdgM_MainFunction()` is always `WdgMTriggerConditionValue`,

> `WdgM_MainFunction()` with alive supervision is called,

> all checkpoints scheduled for the alive supervision interval are passed immediately afterwards, and

> the fault happens immediately after the last checkpoint.

Note
Then the maximum fault detection time is almost:

$$2 * \texttt{WdgMTriggerConditionValue} * \texttt{WdgMSupervisionReferenceCycle}$$

For `WdgMSupervisionReferenceCycle` = 2, the fault detection time is (almost) 4 * `WdgMTriggerConditionValue`.
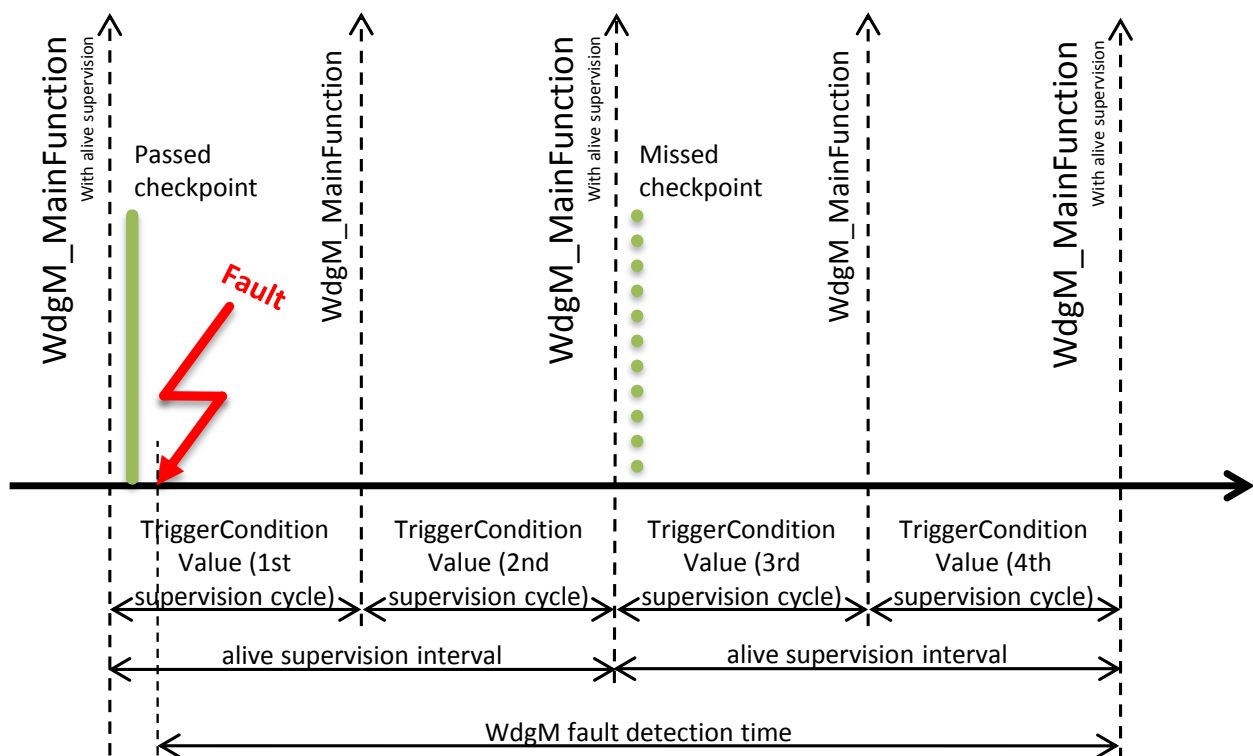
Figure 2-14   Alive supervision fault detection time

2.3.9.2   Deadline Supervision Fault Detection Time

Assume that a fault occurs that leads to a deadline violation. The WdgM fault detection time is the sum of the time spans

> from the fault to the end of the current deadline interval set by the previous checkpoint and

> from the end of the current deadline interval to the next call of `WdgM_MainFunction()` at the end of a supervision cycle.

The worst case for a given configuration happens when

> the time between the calls of `WdgM_MainFunction()` is always `WdgMTriggerConditionValue`,

> a new supervision cycle starts and

> a checkpoint is passed immediately afterwards (setting a new deadline interval for the next checkpoint), and

> the fault happens immediately after the checkpoint.

Then the fault detection time comprises

> (almost all of) the supervision cycle where the fault occurred,

> all supervision cycles up to the supervision cycle where the deadline interval ends,

> including the complete supervision cycle where the deadline interval ends, because the fault is detected at the end of this supervision cycle.

> **Note**
> Then the maximum fault detection time is almost:
>
> $$(nr + 1) * \texttt{WdgMTriggerConditionValue}$$
>
> Where the end of the deadline interval is nr supervision cycles after the successfully passed checkpoint.

For nr = 3, the Fault Detection Time is (almost) 4 * `WdgMTriggerConditionValue`.
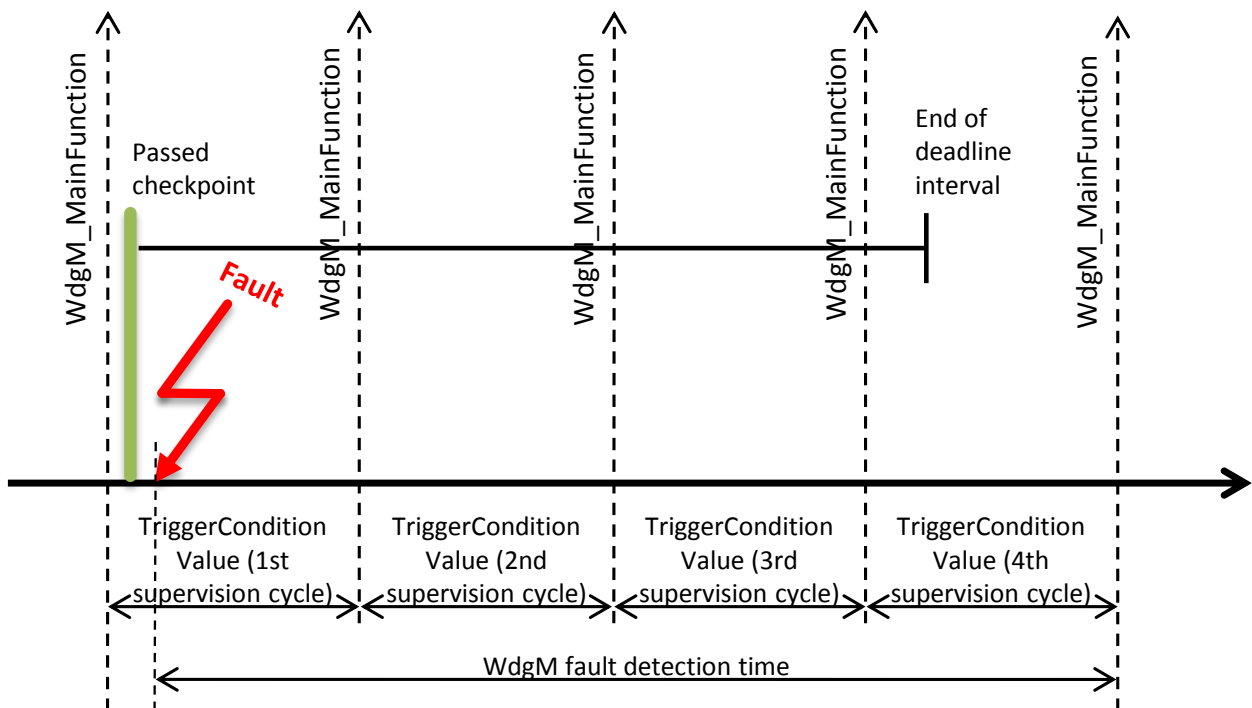
Figure 2-15   Deadline supervision fault detection time

2.3.9.3   Program Flow Supervision Fault Detection Time

Assume that a fault occurs that leads to a program flow violation. The WdgM fault detection time is the sum of the time spans

> from the fault to the call of the next but unscheduled checkpoint and

> from this checkpoint to the next call of `WdgM_MainFunction()` at the end of the current supervision cycle.

The worst case for a given configuration happens when

> the time between the calls of `WdgM_MainFunction()` is always `WdgMTriggerConditionValue,`

> a new supervision cycle starts,

> a scheduled checkpoint is passed immediately afterwards and

> the fault happens immediately afterwards.

Note
Then the maximum fault detection time is almost:
$$(sc + 1) * \texttt{WdgMTriggerConditionValue}$$
Where the unscheduled checkpoint is passed sc supervision cycles after the scheduled checkpoint.

For sc = 2, the fault detection time is (almost) 3 * `WdgMTriggerConditionValue`.
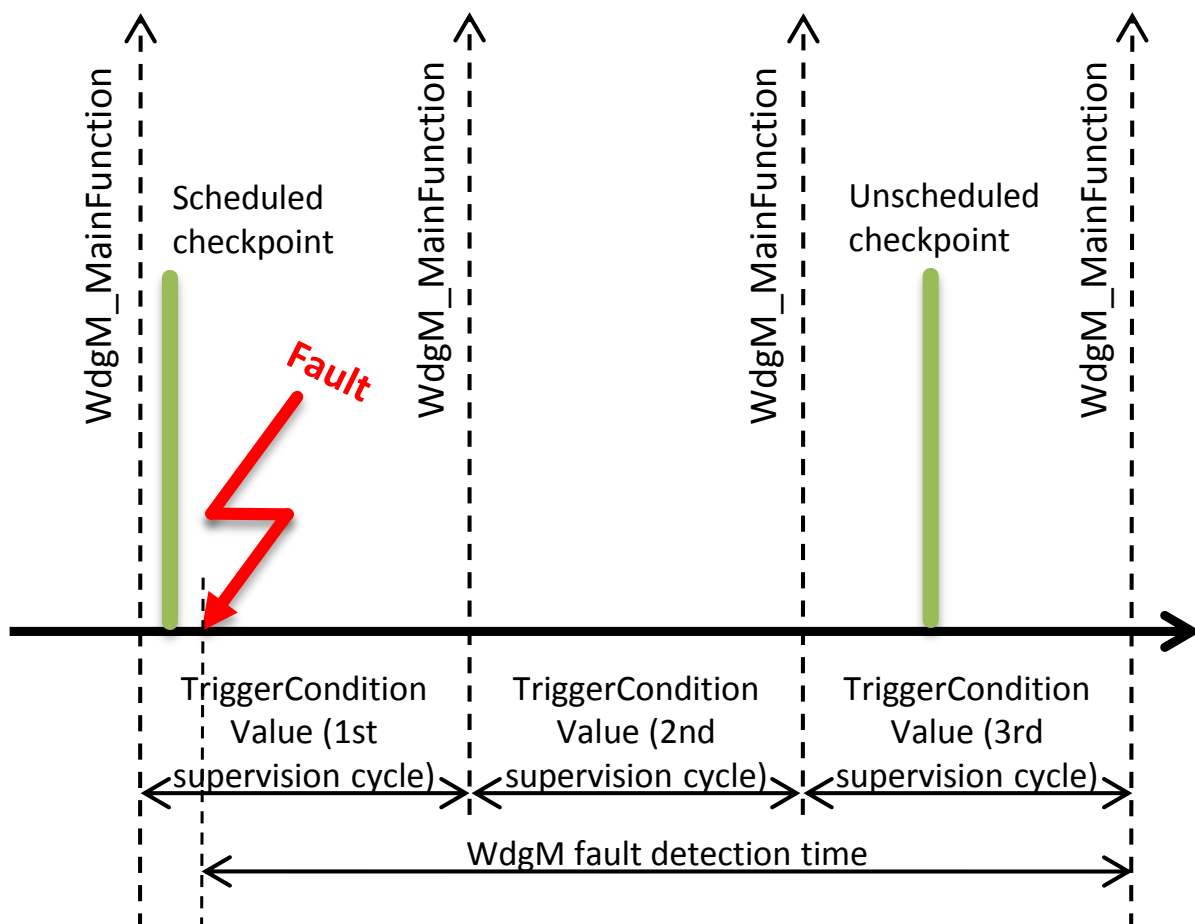
Figure 2-16   Program flow supervision fault detection time

2.3.10 Fault Reaction Time Evaluation

The WdgM fault reaction time is evaluated differently for the various monitoring features as shown in this section.

> **Note**
> This section does not discuss Wdg device resets due to a WdgM error (like DET errors). The section does also not discuss resets using `Appl_Mcu_PerformReset()`.

The WdgM fault reaction time spans

> from the end of the WdgM fault detection time

> to the error escalation to the WdgIf (excluding the processes performed in WdgIf).

The error is escalated to the WdgIf through

> a call of `WdgIf_SetTriggerWindow(DeviceIndex, 0,0)` (if `WDGM_IMMEDIATE_RESET` is `STD_ON`) or

> discontinuing of the Wdg device triggers (if `WDGM_IMMEDIATE_RESET` is `STD_OFF`).

The following assumptions take place here:

> A violation from a fault continues until the error is escalated. Discontinuing a violation before error escalation results in a recovery to OK.

> Each monitored supervised entity is active all the time. Deactivation of a supervised entity aborts the monitoring of this supervised entity. Activation of a supervised entity resumes the monitoring with OK.

The WdgM fault reaction times of the different monitoring features do not affect each other (except that the error escalation of one monitoring violation aborts all other monitoring violations).

2.3.10.1 Alive Supervision Fault Reaction Time

If a call of `WdgM_MainFunction()` ends the fault detection time and starts the fault reaction time, then

the error is escalated by `WdgM_MainFunction()` *i* supervision cycles later,

where

i = (`WdgMSupervisionReferenceCycle` * `WdgMFailedSupervisionRefCycleTol`) + `WdgMExpiredSupervisionCycleTol`.

In the worst case, every supervision cycle has the length of `WdgMTriggerConditionValue`. The fault reaction time is then `WdgMTriggerConditionValue` * *i*, where *i* is as defined above.

2.3.10.2 Deadline Supervision Fault Reaction Time

If a call of `WdgM_MainFunction()` ends the fault detection time and starts the fault reaction time, then

the error is escalated by `WdgM_MainFunction()` *i* supervision cycles later,

where

i = (`WdgMDeadlineReferenceCycle` * `WdgMFailedDeadlineRefCycleTol`) + `WdgMExpiredSupervisionCycleTol`.

In the worst case, every supervision cycle has the length of `WdgMTriggerConditionValue`. The fault reaction time is then `WdgMTriggerConditionValue` * *i*, where *i* is as defined above.

2.3.10.3 Program Flow Supervision Fault Reaction Time

If a call of `WdgM_MainFunction()` ends the fault detection time and starts the fault reaction time, then

the error is escalated by `WdgM_MainFunction()` *i* supervision cycles later,

where

i = (`WdgMProgramFlowReferenceCycle` * `WdgMFailedProgramFlowRefCycleTol`) + `WdgMExpiredSupervisionCycleTol`.

In the worst case, every supervision cycle has the length of `WdgMTriggerConditionValue`. The fault reaction time is then `WdgMTriggerConditionValue` * *i*, where *i* is as defined above.

2.3.11 Reset Path and Safe State

The safe state is entered as a result of an MCU reset. The WdgM builds its functionality on a reliable and robust reset path. The WdgM default reset path uses the Watchdog Device itself through the WdgIf. The Watchdog Device can be either an external chip or an MCU-internal controller. The system integrator can additionally set a secondary path by adding the parameter `WDGM_SECOND_RESET_PATH = STD_ON`. The secondary reset path is used when the Watchdog Interface returns an error response. This error response can be caused by communication errors to the external Watchdog device.
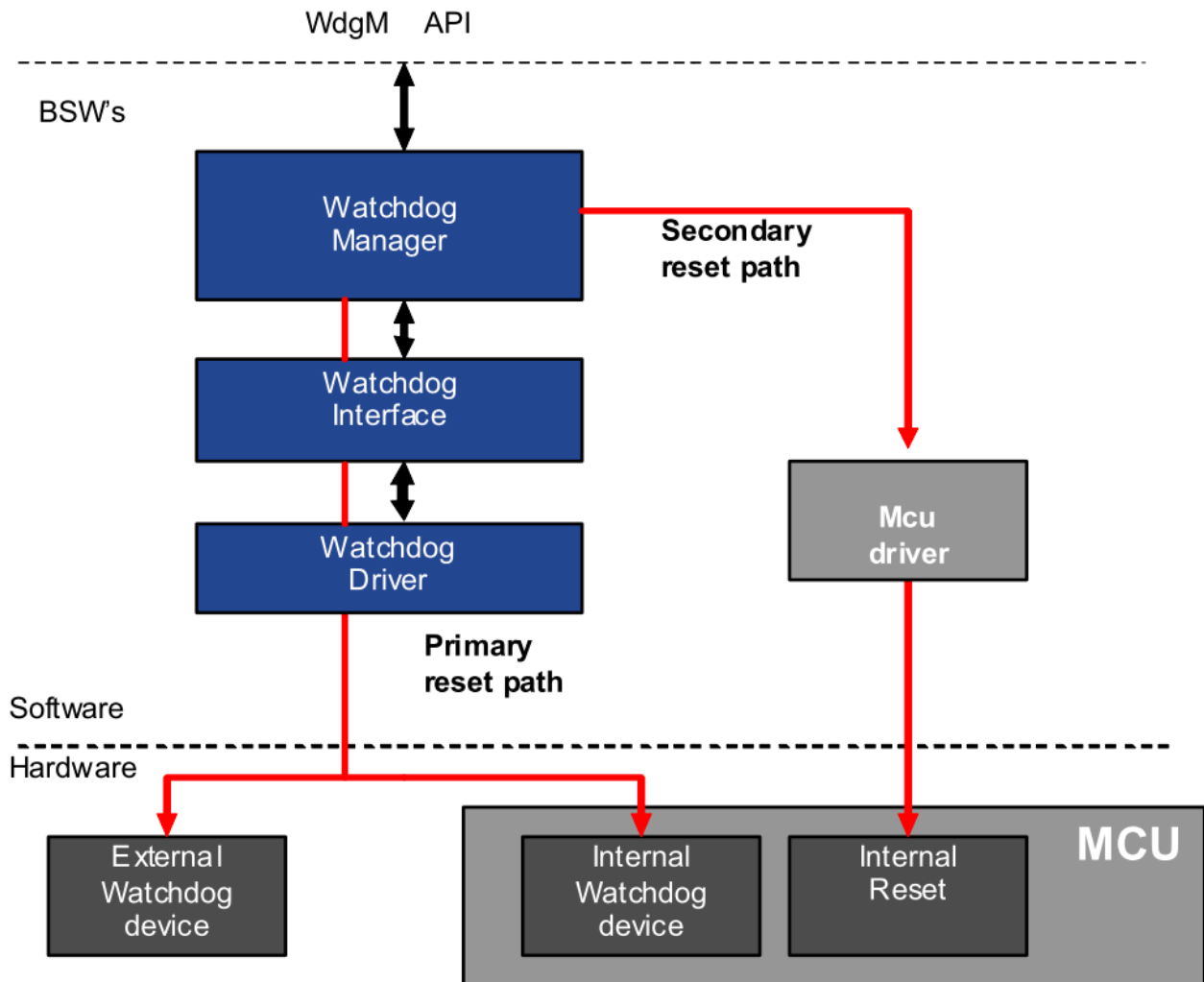
Figure 2-17   Primary and secondary reset path of the WdgM

The WdgM uses the primary reset path for a regular Watchdog-initiated reset and also for an immediate MCU reset. The primary reset path is the preferred path, because it is part of the WdgM software and thus safe. The MCU driver with the AUTOSAR function `Appl_Mcu_PerformReset()` must guarantee freedom from interference.

The secondary reset path is optional. It is used when the primary reset path signals a fault. The WdgM safe state is the MCU reset state.

> **Note**
> The WdgM safe state is not necessarily the system safe state.

The WdgM can invoke the safe state in two ways:

> MCU reset after watchdog timeout by discontinuing watchdog triggering.

> Immediate MCU reset by an immediate watchdog reset. The immediate reset can be configured.

2.3.12 WdgM Local Entity State

Every supervised entity has a local state that expresses the occurrence of detected violations:

State	Description
OK	No violation has been detected
FAILED	A violation has been detected, the reset is pending within a delay time (maybe 0 ticks) and the violation repeats.
EXPIRED	A violation has repeated throughout the delay time. A reset is inevitable.

Table 2-1   WdgM Local Entity Stats

AUTOSAR allows configuring a tolerance delay after an alive counter violation has been detected. See [1] for detailed information. AUTOSAR does not allow configuring such tolerances for program flow and deadline violations. The WdgM allows configuring such tolerances for all three monitoring features described below:

> Once a violation has been detected, the WdgM changes its state from OK to FAILED and starts a so-called tolerance time, which is configured as follows:

The tolerance time is the supervision reference cycle (according to the monitoring feature) multiplied by a supervision reference cycle tolerance value.

> As long as the violation repeats within the tolerance time at least every supervision reference cycle, the WdgM stays in the state FAILED.

> If the violation does not occur in a supervision reference cycle within the tolerance delay, the WdgM returns to the state OK as if no violation had happened. Only the status change is logged.

> If the violation has repeated to the end of the tolerance time, the WdgM enters the state EXPIRED.

d_tol   Deadline Reference Cycle Tolerance
d_rc   Deadline Reference Cycle
d_violation   at least one Deadline violation within the last **d_rc**

pf_tol   Program FlowReference Cycle Tolerance
pf_rc   Program FlowReference Cycle
pf_violation at least one Program Flow violation within the last **pf_rc**

ac_tol   Alive Counter Reference Cycle Tolerance
ac_rc   Alive Counter Reference Cycle
ac_violation at least one Alive Counter violation within the last **ac_rc**

Reference Cycle   reference cycle length given in number of WdgM cycles

Figure 2-18   Modified state machine

Note

The AUTOSAR implementation can be simulated for deadline and program flow violations with

```
reference cycle = reference cycle tolerance = 0
```

The exact names of the configuration fields for the tolerance delay are:

Monitoring	Reference Cycle	Reference Cycle Tolerance
Alive Supervision	WdgMSupervisionReferenceCycle	WdgMFailedSupervisionRefCycleTol
Program Flow Supervision	WdgMProgramFlowReferenceCycle	WdgMFailedProgramFlowRefCycleTol
Deadline Supervision	WdgMDeadlineReferenceCycle	WdgMFailedDeadlineRefCycleTol

Table 2-2   Names of configuration fields

2.3.13 WdgM Global State

The local states are periodically summarized in a WdgM global state. If all supervised entities have the state `OK`, then the global state is `OK`. When at least one supervised entity changes to the state `FAILED`, then the global state becomes `FAILED`. When at least one supervised entity changes to the state `EXPIRED`, the global state becomes `EXPIRED`. Once the global state is `EXPIRED`, the WdgM continues the delay until it enters the state `STOPPED`. This is when the WdgM stops triggering the Watchdog (or resets it). The delay is the supervision cycle multiplied by the configurable expired supervision cycle tolerance (parameter `WdgMExpiredSupervisionCycleTol`).

Once in the state `STOPPED`, the WdgM brings the system to the safe state by performing a system reset through the WdgIf module and, thus, through the watchdog device(s) in the system. If the preprocessor option `WDGM_SECOND_RESET_PATH` is set to `STD_ON` and the WdgIf reports a failure, then the system goes into a safe state through the MCU module.

> **Note**
> In a multi-core system, each Watchdog Manager instance running on a particular processor core builds a separate global state independently of the other processor cores.

2.3.14 Basic Operation of the WdgM Stack

The WdgM is the upper layer of the WdgM Stack. As described above, the WdgM is responsible for monitoring applications through preconfigured supervised entities. The result of this monitoring is usually translated into servicing one or more watchdog devices through the rest of the WdgM Stack – the Watchdog Interface (WdgIf) and one or more Watchdog Drivers (Wdg).

Figure 2-19 shows a possible WdgM Stack configured to run on a microcontroller with two watchdog devices, an internal and an external one.
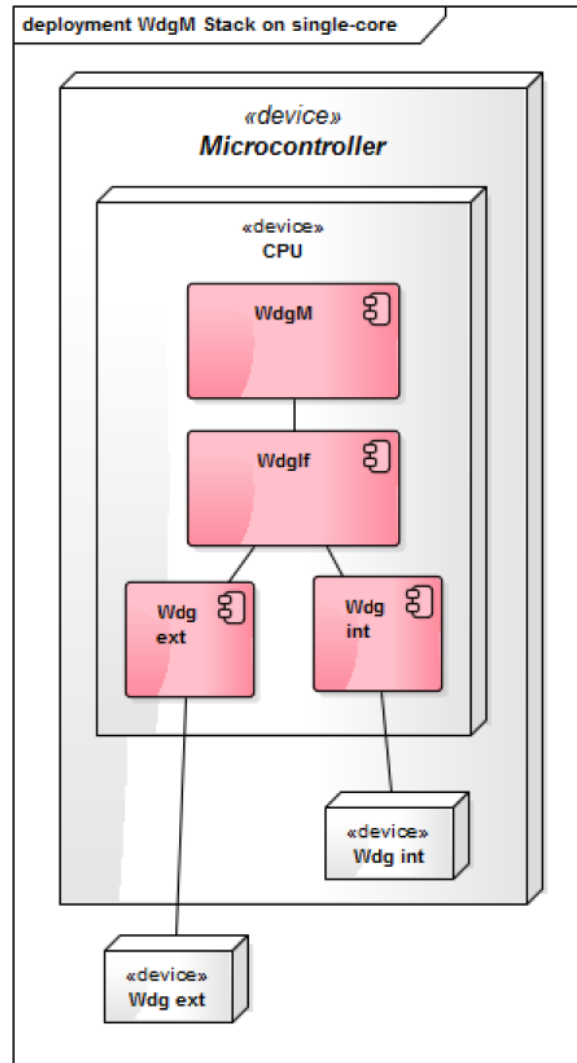
Figure 2-19   Example of an WdgM Stack configuration

Figure 2-20 shows the behavior of the WdgM Stack.

First, the Wdg drivers configured in the system are initialized. Then the WdgM is initialized, and it calls the `SetMode` functions of each configured driver during its initialization. During runtime, the monitored applications report to the WdgM by calling the function `WdgM_CheckpointReached()` or directly via RTE. During this call, the program flow and part of the deadline supervision is evaluated (see compute SE local state #1 in Figure 2-20). In each supervision cycle, `WdgM_MainFunction()` is called. It evaluates the status of each supervised entity, the rest of deadline supervision and alive supervision (see compute SE local states #2 in Figure 2-20) and, based on this, it computes the global state. Depending on the global state, the configured watchdogs are either serviced, or a reset is deliberately caused. The latter is done either by omitting the servicing or by instructing the watchdog to make a reset right away (for more information refer to parameter `WdgMImmediateReset`).
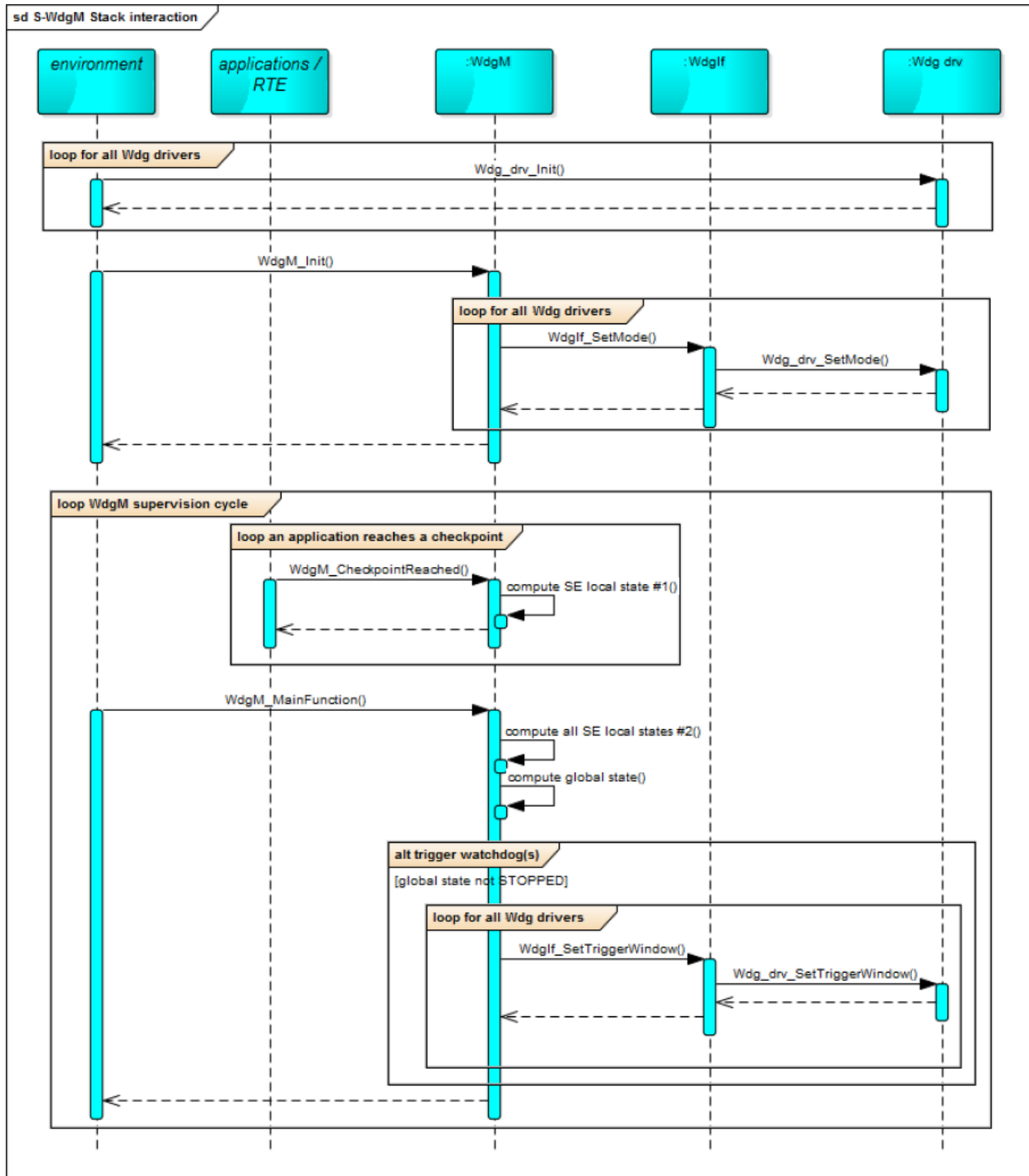
Figure 2-20   Behavior of the WdgM Stack

2.4   WdgM in Multi-Core Systems

The WdgM can be used in single-core and multi-core systems. Each processor core to be monitored by the WdgM runs a separate WdgM instance. It is not necessary that the WdgM Stack runs on each processor core. It can be configured to run on a subset of them only where monitoring of supervised entities is required.

Each WdgM instance runs independently of the others and must be initialized with its own configuration. It has its own time base and calls the `WdgM_MainFunction()` separately.

If the WdgM is configured to run on a multi-core system, then an internal preprocessor option (`WDGM_MULTICORE_USAGE`) is automatically set to `STD_ON`. Thus, the embedded code can handle several processor cores. Otherwise, this option is set to `STD_OFF`, which optimizes the code for a single-core system. The optimizations are done even if the WdgM is configured to run only on one core in a multi-core environment.

In order to configure the WdgM (ECU description file) to run on several processor cores in a multi-core system, a separate `WdgMConfigSet` container needs to be configured for each core. The WdgMConfigSet container has a WdgMMode subcontainer (note, that only one is allowed), which identifies the processor core that it is configured to run on only one core of a multi-core system.

Note, that the WdgMGeneral container which contains general configuration parameters as well as the supervised entities in the system is one for all configured cores. Each supervised entity can be used on one core only and must have a unique ID in the system.

As the WdgM instances run independently on the different processor cores, each supervised entity in the system is configured for one processor core and can be used only on that core. Global transitions between supervised entities are allowed only for supervised entities running on the same processor core.

There are 2 different concepts on how the WdgM Stack can react to detected violations, the independent and combined core reaction concept.

The independent core reaction concept says that each WdgM instance controls one or more watchdogs. It builds an independent global state and decides on triggering its watchdog(s) or causing a deliberate reset. Whether a processor core reset or system reset is issued, depends on the hardware configuration and not on the WdgM.

Figure 2-21 shows the operation of the WdgM on a multi-core system (independent core reaction).
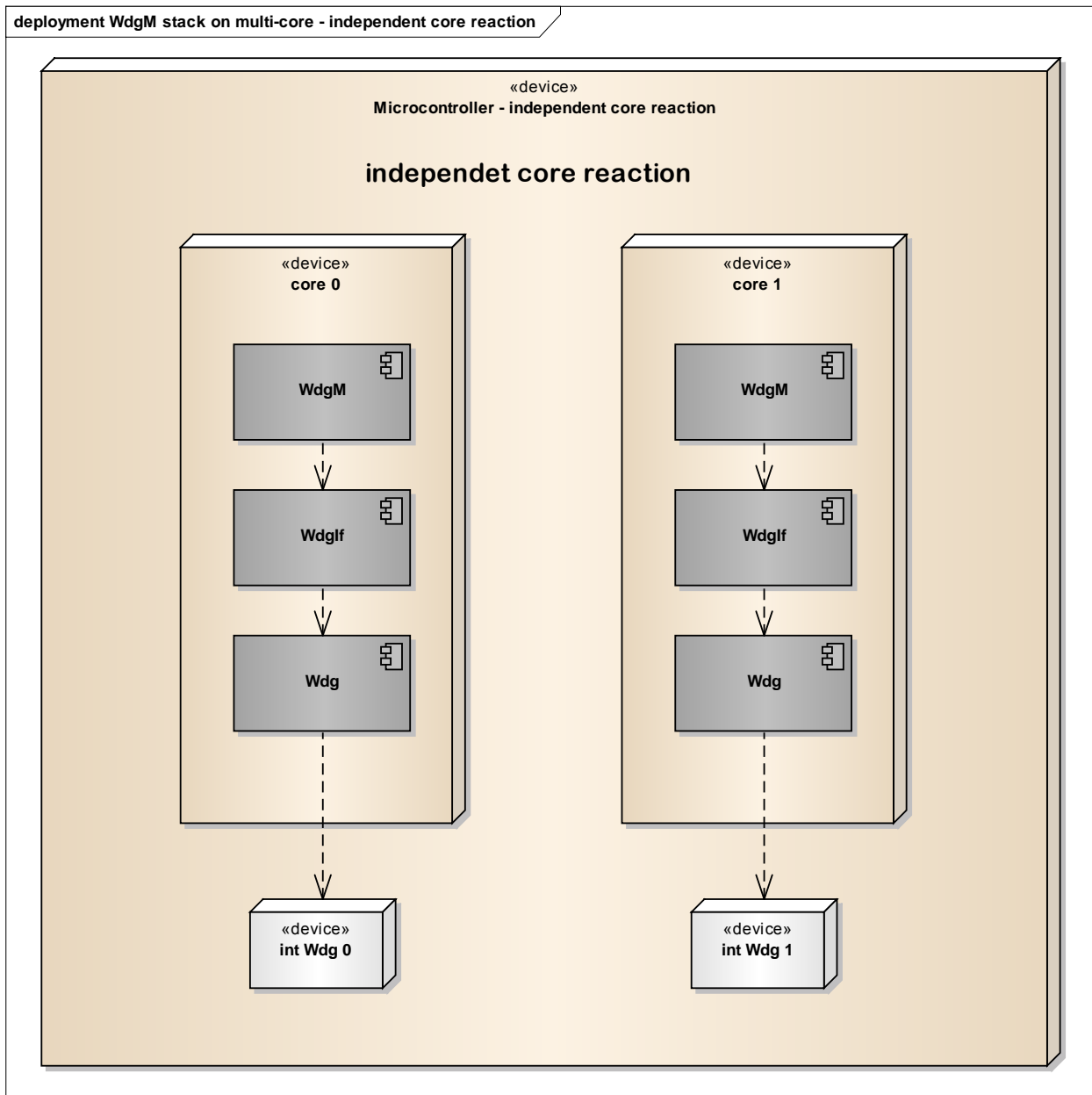
Figure 2-21   WdgM Stack on a multi-core system configured for independent core reaction

The combined core reaction concept declares, that each WdgM instance independently builds a global state of its processor core and the several global states are then combined

> either by hardware (e.g. a special hardware module in the microcontroller reading the states of the internal watchdog devices of each processor core)

> by software (e.g. a special watchdog driver that is called on each processor core and that combines the status of each core into a single reaction)

Combining the WdgM status on each core in hardware is strongly hardware- dependent, and its applicability can vary from microcontroller to microcontroller.

Combining the WdgM status on each core in software can be done with the feature State Combiner of the underlying WdgIf module.

2.4.1 State Combiner

The State Combiner is a special hardware-independent peace of software that is implemented as an additional feature of the WdgIf module. On one core, the State Combiner is configured to work in master mode in order to control the actual watchdog device. The instances of the State Combiner on the other cores are configured to work in slave mode. They do not trigger but communicate with the master State Combiner only via shared memory. The State Combiner on the master core will only allow triggering the actual watchdog device if the global status of the WdgM instances on all cores is other than STOPPED. In other words, as soon as at least one core has detected an irreparable error and requests a reset, the actual watchdog device will not be serviced anymore (or an explicit reset will be initiated depending on the ECUC description parameter WdgMImmediateReset).
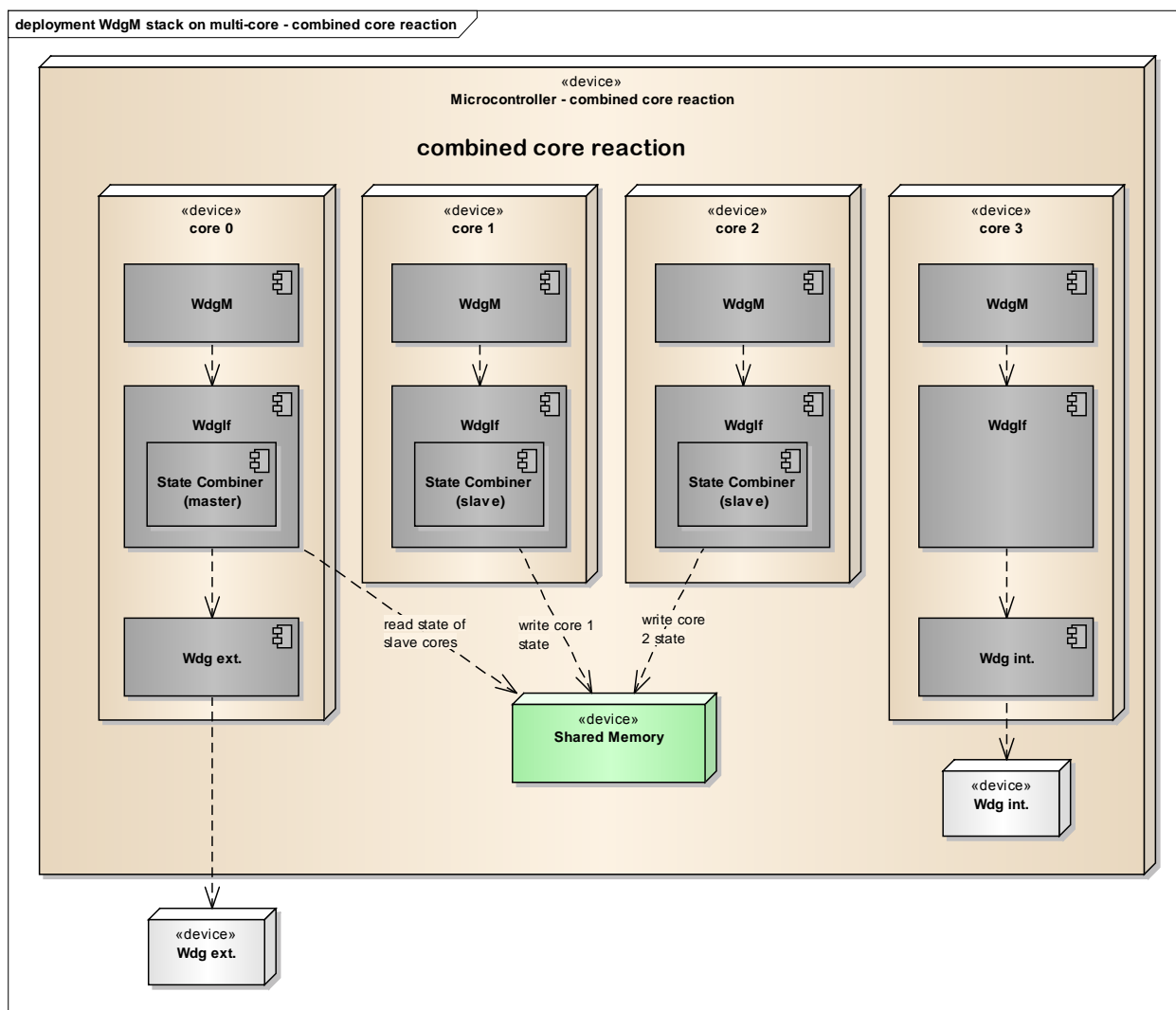
Figure 2-22   WdgM Stack on a multi-core system using the State Combiner for a combined core reaction

Figure 2-23 shows the dynamic behavior of a WdgM running on 2 cores with a State Combiner for a combined core reaction.
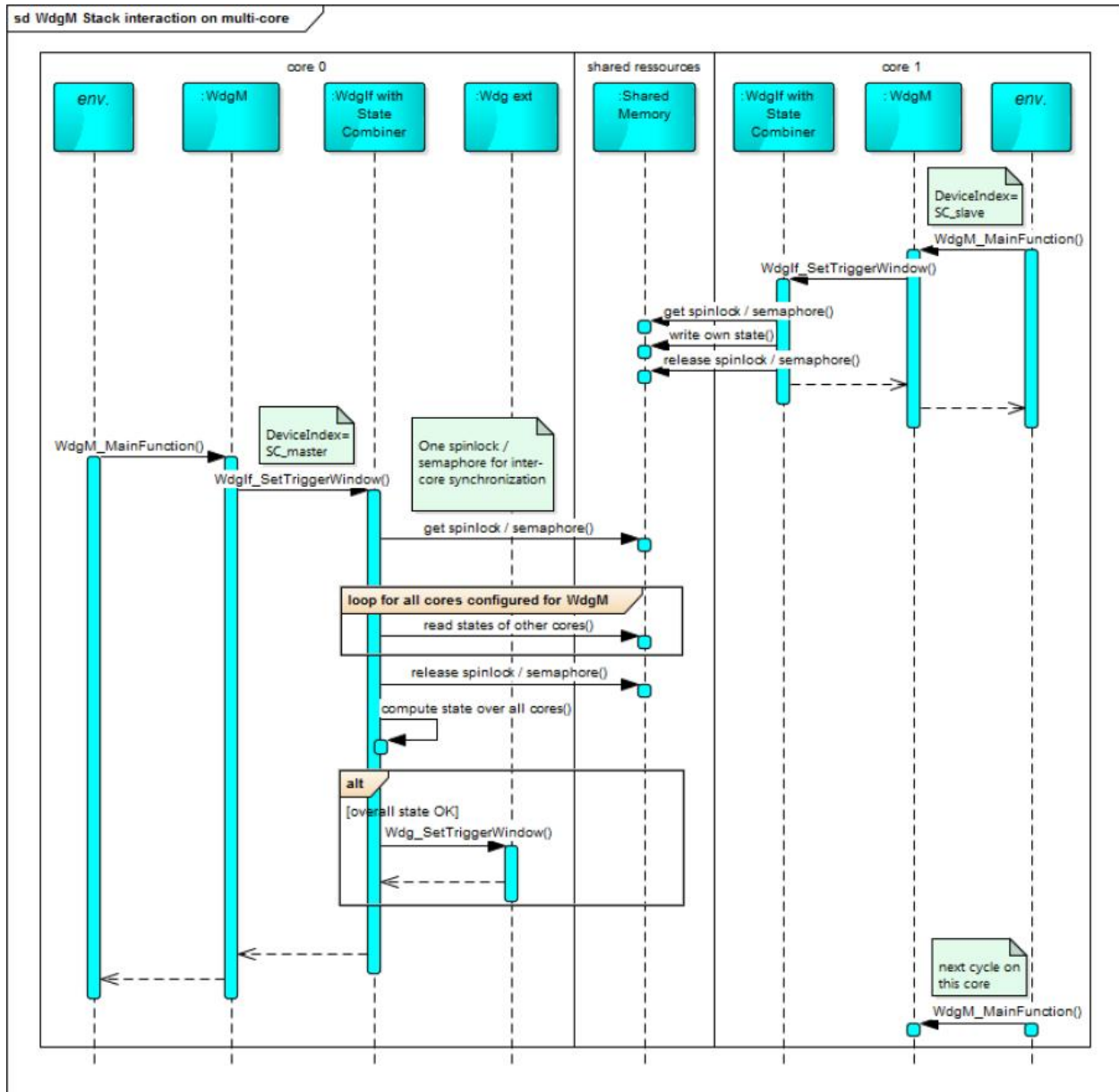
Figure 2-23   Dynamic Behavior on a multi-core system using the State Combiner for a combined core reaction

For more information on the State Combiner refer to the WdgIf User Manual [4].

2.4.2   AUTOSAR Debugging

AUTOSAR Debugging allows debugging the WdgM module by granting it access to certain module internal variables. This AUTOSAR feature is extended for the WdgM by adding special functions that make the debugging process or tracing easier, e.g. detection and types of WdgM violations.

Variables accessible for debugging:

> The local monitoring status of each supervised entity is accessible through the API `WdgM_GetLocalStatus()`

> `WdgMConfigPtr -> WdgMSupervisedEntityRef[SEID] .EntityStatusGRef -> LocalMonitoringStatus`

> The global monitoring status: accessible through the API `WdgM_GetGlobalStatus()` or `WdgMConfigPtr -> DataGRef -> GlobalMonitoringStatus`

> The alive counters of each checkpoint of a supervised entity: `WdgMConfigPtr -> WdgMSupervisedEntityRef[SEID].WdgMCheckpointRef[CPID]. WdgMAliveLRef -> AliveCounter`

> The time when the initial CP of an SE has been reached: `WdgMConfigPtr-> WdgMSupervisedEntityRef[SEID].EntityStatusLRef -> RememberedInitialCheckpointTime`

> The time when the most recent CP of an SE has been reached: `WdgMConfigPtr -> WdgMSupervisedEntityRef[SEID].EntityStatusLRef -> RememberedCheckpointTime`

> The most recently reached CP of an SE: WdgMConfigPtr -> WdgMSupervisedEntityRef[SEID].EntityStatusLRef -> RememberedCheckpointId

Note
SEID is the ID of an SE. CPID is the ID of a CP. WdgMConfigPtr is a pointer to the configuration with which the WdgM was initialized.

Additional debugging feature (not defined in AUTOSAR):

> The function WdgM_ WdgM_GetFirstExpiredSEViolation() provides a way to detect what kind of violation caused the first SE in the system to change its local status to WDGM_LOCAL_STATUS_EXPIRED: program flow, deadline, alive supervision or a combination between them.

3 Functional Description

The WdgM monitors safety-relevant applications on the ECU. The WdgM is a basic software module at the service layer of the standardized basic software architecture of AUTOSAR. The WdgM monitors the program flow of a configurable number of so-called supervised entities (SE). When the WdgM detects a violation of the preconfigured temporal or logical constraints in the program flow, it takes a number of configurable actions to log the fault and to go to a safe state after a configurable time delay. The safe state is reached by resetting the watchdog or by omitting watchdog triggering.

Every supervised entity has a defined control flow. Significant points in this control flow are represented by checkpoints (CP). This means the control flow can be modeled as a graph, with the checkpoints being the nodes and the pieces of code in between being the transitions (see Figure 2-4 for an example).

The WdgM configuration defines the allowed transitions between the checkpoints, and the timing constraints for these transitions

> within every supervised entity (local transitions)

> between checkpoints of different supervised entities (global transitions)

The supervised entities have to report to the WdgM when they have reached a checkpoint. Thus, the developer has to insert calls at the checkpoints that pass this information to the WdgM.

The WdgM functionality partially deviates from the AUTOSAR requirements. For details, refer to Section Deviations from the AUTOSAR 4.0.1 Watchdog Manager.

3.1 Features

The features listed in the following tables cover the functionality specified for the WdgM.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1   Supported AUTOSAR standard conform features

Vector Informatik provides further WdgM functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 3-2   Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
> Services to initialize the WdgM
> Functionality for:
> Alive Supervision
> Deadline Supervision

Supported AUTOSAR Standard Conform Features
> Program Flow Supervision

Table 3-1 Supported AUTOSAR standard conform features

> **Caution**
> Deadline Supervision and Program Flow Supervision can only be used if the additional feature "WdgM_ProgramFlowAndDeadlineMonitoring" is licensed.
> (See also sections Deadline Supervision and Program Flow Supervision)

3.1.1 Deviations from the AUTOSAR 4.0.1 Watchdog Manager

The WdgM is compatible with the AUTOSAR 4.0.1 Watchdog Manager, but not fully compliant. This has the following reasons:

> The AUTOSAR specification does not define functionality comprehensively and precisely enough for implementation (e.g. global transitions).

> The AUTOSAR specification does not contain certain functionality (e.g. program flow, deadline supervision recovering).

> The AUTOSAR specification defines an approach that is very complex to be handled by the user or consumes too much run time (WdgM mode switching).

> The AUTOSAR specification does not fully consider safety requirements (e.g. windowed Watchdog Trigger).

Below you can find the deviations from the AUTOSAR 4.0.1 Watchdog Manager in detail:

3.1.1.1 Entities, Checkpoints and Transitions

> For periodical watchdog triggering at least one supervised entity and one checkpoint should be defined.

> In contrast to AUTOSAR, local activity flags of the supervised entities are set back to `FALSE` every time an end checkpoint of this supervised entity is reached as specified in later versions of the WdgM. Analogously, the global activity flag is set back to `FALSE` as soon as a global end checkpoint is reached.

> Local initial checkpoints cannot have incoming local transitions, but they can have incoming global transitions.

> Local end checkpoints cannot have outgoing local transitions, but they can have outgoing global transitions.

> If global transitions are used, then there must be exactly one global initial checkpoint.

> The global initial checkpoint should be called before any other global checkpoint is invoked.

> If a non-initial checkpoint of a supervised entity is reached and this supervised entity is not active, then this is considered to be a program flow violation in this supervised entity.

> If a checkpoint is the source for a local and a global transition, then only one of the two transitions can occur. The other one is considered a program flow violation. This is because the program flow cannot split into 2 paths. If, for example, a new task is started from a CP1 (global transition to CPnew) and the original task continues (local transition to CP2), then the sequence following the sequences of checkpoint hits is not allowed:

  > CP -> CPnew -> CP2 and

  > CP -> CP2 -> CPnew.

> If a local initial checkpoint is the destination checkpoint for a global transition, then the checkpoint must be hit by following the global transition. There is a dilemma, though: If several supervised entities form a cycle of transitions, with each supervised entity entered via a global transition from the previous supervised entity, then there is no way to start the cycle, because no local initial checkpoint is allowed to be hit in a way other than via the global transition. The solution is an exception in the WdgM: A local initial checkpoint can be hit, not coming through the global transition, if it is also the global initial checkpoint.

> As in AUTOSAR, the WdgM needs a time source in order to measure transition deadlines. Whereas AUTOSAR does not define the source for ticks, the WdgM allows the user to choose between three tick sources:

  > Internal software source

  > External tick source

  > OsCounter source

  For details see Section Deadline Measurement and Tick Counter.

  The checkpoint and entity identifiers are zero-based and increase the list of integer numbers without gaps.

> Deadline supervision is bound to program flow. Only if program flow transitions are configured, it is possible to configure transition deadlines.

> The local/global end checkpoint does not need to be defined.

> Either zero or maximum one global initial checkpoint can be configured. So there can be zero or one global graph.

3.1.1.2 Watchdog and Reset

> For safety reasons, the WdgM uses the primary watchdog reset as an immediate reset (`WDGM_IMMEDIATE_RESET = STD_ON`). In contrast, the AUTOSAR Watchdog Manager uses the external function `Mcu_PerformReset()`.

  The WdgM does not support a partition reset with `BswM_WdgM_RequestPartitionReset()`.

3.1.1.3 API

> The WdgM function `WdgM_SetMode()` switches the trigger mode only. This relates to the fields

  > `WdgMTriggerConditionValue`

  > `WdgMTriggerWindowStart` (not used – shall be configured with 0)

  > `WdgMWatchdogMode`

> It does not change the set of supervised entities. This can be simulated by activating and deactivating different sets of supervised entities for different modes.

> For safety and complexity reasons, the function `WdgM_DeInit()` is not implemented.

> The status reporting mechanism is configurable.
  On one hand mode ports can be used to notify applications / SWCs etc. about status changes as specified in AUTOSAR. On the other hand the WdgM can be configured to use direct callback notification to report a local and global state change.

> The WdgM checks the configuration independently of the `WdgMDevErrorDetect` parameter. This parameter enables/disables only the DET reporting.

3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Functionality for multi-core handling
The WdgM allows tolerance delay for all three monitoring features. In AUTOSAR, this is restricted to alive supervision. Tolerance delay allows recovering from program flow and deadline violations as well as from alive counter violations.
The interpretation of the AUTOSAR parameter `WdgMExpiredSupervisionCycleTol` implements a delay of (`WdgMExpiredSupervisionCycleTol + 2`) supervision cycles. The WdgM implements a delay of `WdgMExpiredSupervisionCycleTol` supervision cycles. This allows configuring no delay, with the tolerance value set to 0.
The WdgM provides the following function in addition to the AUTOSAR Debugging features: `WdgM_GetFirstExpiredSEViolation()`.
The WdgM provides the functions `WdgM_DeactivateSupervisionEntity()` and `WdgM_ActivateSupervisionEntity()` for deactivating and activating of the SE. These functions are not AUTOSAR 4.0.1 compatible.

Table 3-2   Features provided beyond the AUTOSAR standard

3.2 Initialization

In a safety-related system, the initialization of the Watchdog device should be done as soon as possible after system start (at least before a QM task may compromise the initialization process). The Watchdog device starts the counter for the next expected trigger.

> **Caution**
> Per default, the initialization routine is mapped to init-phase two (INIT_TWO_IF).
>
> In a safety related system the init phase has to be changed. This can only be done if the function WdgM_Init within the EcuC (InitFunctions) is set to user defined. An earlier phase has to be chosen.

> **Note**
> The ways how the Watchdog device is initialized, configured, and how it reacts are platform-dependent and can be different. See the corresponding Watchdog Driver User Manual

The time between the initialization of the Wdg and the first triggering in function `WdgM_MainFunction()` (supervision cycle 0) must match the Watchdog requirements. This time can be adapted in the Wdg configuration by changing the initial Wdg trigger window to meet the operating system start time requirements (see Figure 3-1).
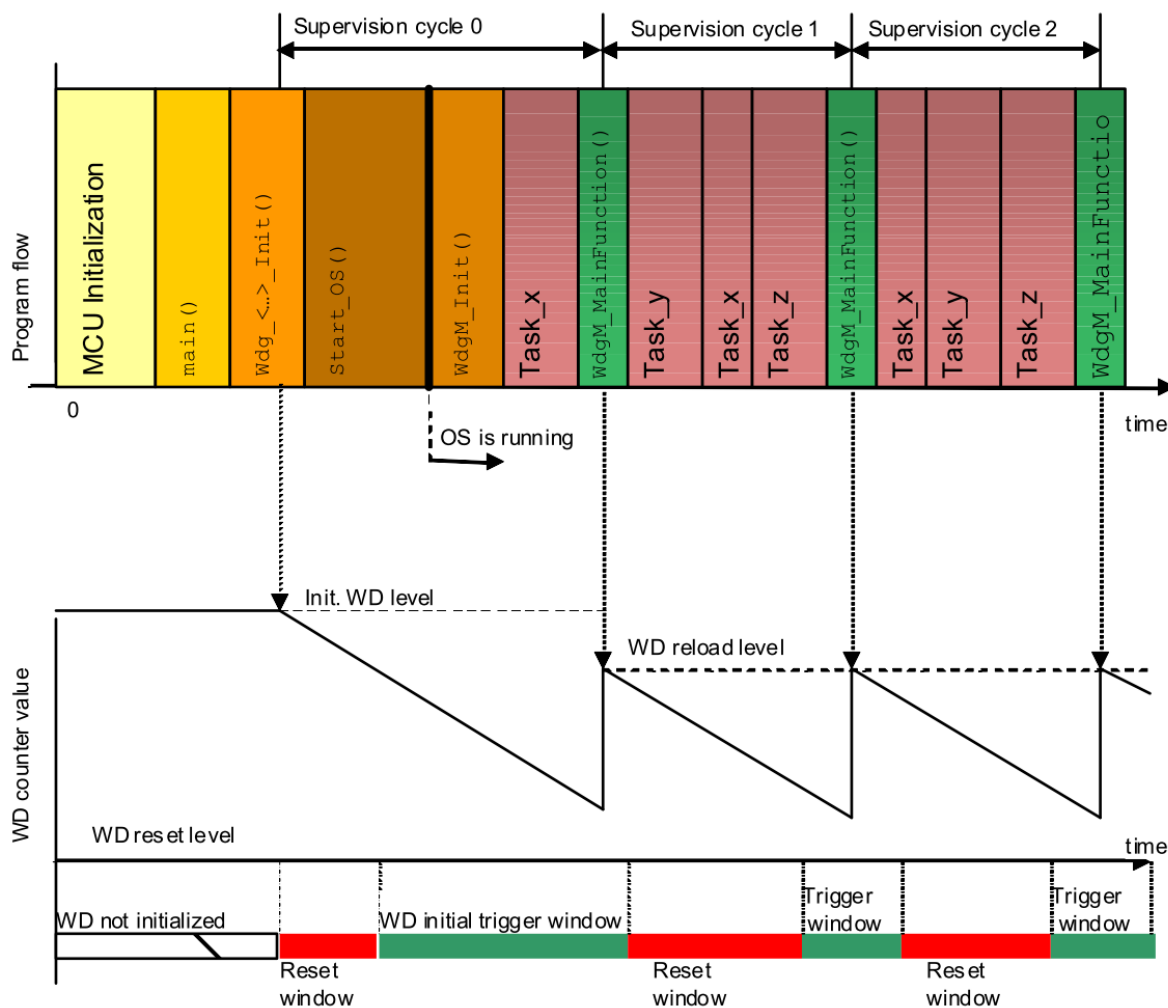
Figure 3-1   Start phase of the WdgM

The y-axis in Figure 3-1 shows the Wdg counter value, which is reset after each trigger. Then the countdown runs until the Wdg is triggered again (within the Wdg initial trigger window or Trigger window) or 0 (Wdg reset level) is reached (i.e. the window has been missed) so that a reset is performed.

> **Note**
>
> > Not all hardware platforms can configure a different trigger time for the first supervision cycle (cycle 0).
>
> > In the first supervision cycle, the alive counter evaluation can be suppressed by the parameter `WDGM_FIRSTCYCLE_ALIVECOUNTER_RESET`.
>
> > The functions `WdgM_Init()` and `WdgM_MainFunction()` functions can be placed inside a task, too.
>
> > The function `Wdg_<...>_Init()` can be placed before `main()`.
>
> > For a multi-core system the `WdgM_Init()` function must be called on each processor core once, with the valid configuration for this processor core.
>
> > The `WdgM_MainFunction()` called periodically on each processor core, on which WdgM is running, with the configured period for this processor core.

After the execution of function `WdgM_Init()` the supervision of configured entities is activated and the checkpoints can be executed (called).

3.3 Memory Sections

Memory segmentation into sections is especially important when memory protection is used in the system.

The WdgM uses three basic RAM data sections:

1. Memory sections for local data of every SE: This section contains local information about every supervised entity and, if defined, also the alive counters. These variables are used by the WdgM_CheckpointReached() function and are part of the private SWC (task, application) memory and written only in the context of this SWC.

> **Note**
> > The WdgM does not protect this memory section.
>
> > For a multi-core system, the local data section for a SE must be accessible from the core for which this SE is configured.

2. Memory sections for global data: This section contains the WdgM global data such as WdgM global status and timebase tick counter. It is a WdgM private memory.

> **Note**
> In the AUTOSAR environment, where QM and safety-related modules are used together, the WdgM global data should be placed in a so-called trusted memory section to guarantee its safety and integrity.
> > For a multi-core system, the global data section is configured per mode, i.e. separately for each processor core, on which the WdgM is running.

3. Memory sections for global shared data: This section contains data such as the last active entity. This memory must be writable for all SWCs using the `WdgM_CheckpointReached()` function and for the `WdgM_Init()` function. As this is a memory where all the QM SWCs could write, the WdgM variables are protected (stored double-inverted) by the WdgM itself. The WdgM checks the correctness of these variables with read operations. If a fault is detected, the WdgM initiates a reset

> **Note**
> For a multi-core system, where several cores are configured for the WdgM, the global shared section must be accessible by each of these cores.

S-WdgM RAM memory map
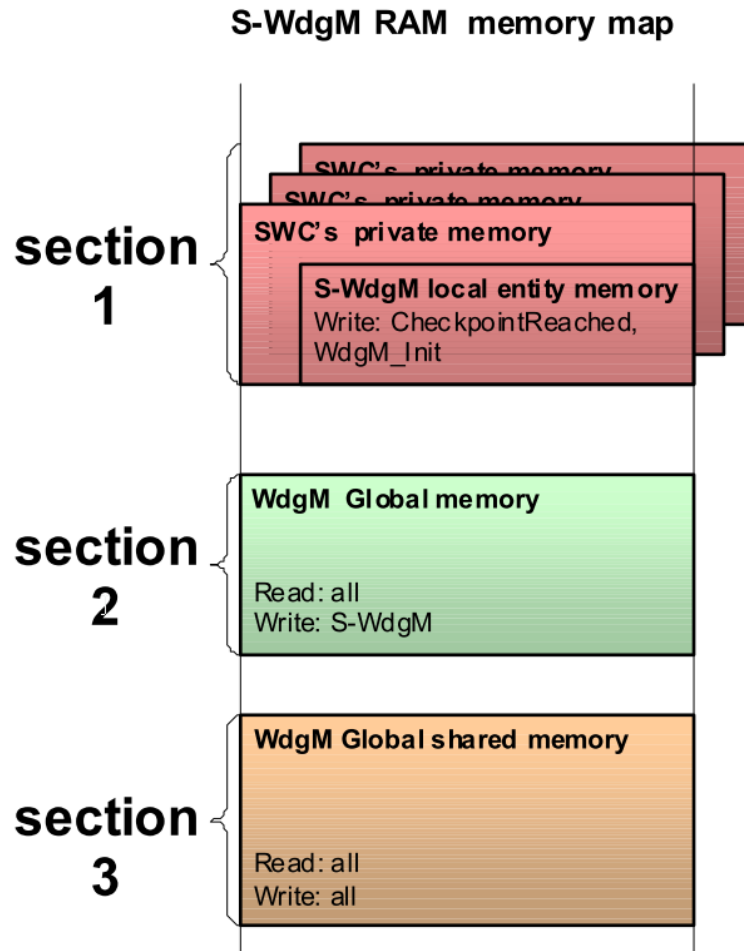
Figure 3-2   Memory usage of the WdgM

3.3.1   Memory Sections Details

Local entity memory:

Local entity data is supervised entity private data. This is the data where the function `WdgM_CheckpointReached()` writes. The WdgM configuration generator provides defines so that the status variables of every supervised entity can be placed in a separate RAM section. The declaration of every entity starts with defines `WDGM_SEi_START_SEC_VAR_*` and ends with `WDGM_SEi_STOP_SEC_VAR_*`, where `i` is the ID of the supervised entity. Theses defines are in the generated file `WdgM_OsMemMap.h`. Hence, it must be included in the file `MemMap.h`.

If the entity is linked to an OS application (through its ECU description parameter `WdgMAppTaskRef`), then the supervised entity data is placed in a section embedded in `appl_name_START_SEC_VAR_*` and `appl_name_STOP_SEC_VAR_*` or `OS_START_appl_name_VAR_*` and `OS_STOP_appl_name_VAR` (if MICROSAR OS as of version Gen7), where `appl_name` is the name of the application. In this case, the integrator must make sure to include the file `Os_MemMap.h` in file `MemMap.h` after the file `WdgM_OSMemMap.h`.

Global memory: Global data are private WdgM variables. The memory mapping defines are `WDGM_GLOBAL_START_SEC_VAR_*` and `WDGM_GLOBAL_STOP_SEC_VAR_*`.

This section can be mapped to an OS application (through its ECU description parameter `WdgMGlobalMemoryAppTaskRef`). For this mapping, defines `appl_name_START_SEC_VAR_*` and `appl_name_STOP_SEC_VAR_*` or `OS_START_appl_name_VAR_*` and `OS_STOP_appl_name_VAR` (if MICROSAR OS as of version Gen7) are used, where `appl_name` is the name of the application. In this case, the integrator must make sure to include the file `Os_MemMap.h` in file `MemMap.h` after the `WdgM_OSMemMap.h`.

As this section is internally not protected by the WdgM, it should be in a memory area where it cannot be corrupted.

Global shared memory: Global shared data should be placed in a RAM section where all tasks can read and write to that data. For a multi-core system, the global shared data section must be accessible by each processor core.

The memory mapping defines are `WDGM_GLOBAL_SHARED_START_SEC_VAR_*` and `WDGM_GLOBAL_SHARED_STOP_SEC_VAR_*`. These variables are internally protected by the WdgM. For this mapping, defines `GlobalShared_START_SEC_VAR_NOINIT_UNSPECIFIED` and `GlobalShared_STOP_SEC_VAR_NOINIT_UNSPECIFIED` or `OS OS_START_SEC_GLOBALSHARED_VAR_NOINIT_UNSPECIFIED` and `OS_STOP_SEC_GLOBALSHARED_VAR_NOINIT_UNSPECIFIED` (if MICROSAR OS as of version Gen7) are used.

3.3.2 Code and Constants

Following memory sections need to be set up for WdgM's code:

Section	Description
`WDGM_START_SEC_CODE` / `WDGM_STOP_SEC_CODE`	Set up manually, e.g. in `MemMap.h`.

Table 3-3   Code and Constants

Following memory sections need to be set up for WdgM's constants:

Section	Description
`WDGM_START_SEC_CONST_32BIT` / `WDGM_STOP_SEC_CONST_32BIT` `WDGM_START_SEC_CONST_UNSPECIFIED` / `WDGM_STOP_SEC_CONST_UNSPECIFIED`	Set up manually, e.g. in `MemMap.h`.

Table 3-4   WdgM constants

3.3.3 Module Variables

3.3.3.1 Module Variables with MICROSAR Os Gen6 / AUTOSAR Os version 4.0

Following memory sections need to be set up for WdgM's module variables:

Section	Description
`WDGM_GLOBAL_START_SEC_VAR_32BIT_COREn_PRIVATE` /	If the ECU description parameter

Section	Description
`WDGM_GLOBAL_STOP_SEC_VAR_32BIT_ COREn_PRIVATE` `WDGM_GLOBAL_START_SEC_VAR_NOINIT_ UNSPECIFIED_COREn_PRIVATE` / `WDGM_GLOBAL_STOP_SEC_VAR_NOINIT_ UNSPECIFIED_COREn_PRIVATE`	`WdgMGlobalMemoryAppTaskRef` is set, then these sections are renamed according to the configured OS application (prefix "`WDGM_GLOBAL_`" is converted to "`<OSApp>_`", where `<OSApp>` is the name of the OS application; suffix "`_COREn_PRIVATE`" is removed) and generated as part of `WdgM_OsMemMap.h`. Otherwise they need to be set manually, e.g. in `MemMap.h`. The suffix "`_COREn_PRIVATE`" corresponds to the processor core for which the OS application is configured.
`WDGM_GLOBAL_SHARED_START_SEC_VAR_ NOINIT_UNSPECIFIED` / `WDGM_GLOBAL_SHARED_STOP_SEC_VAR_ NOINIT_UNSPECIFIED`	These sections are always assigned in the generated file `WdgM_OsMemMap.h` to OS sections and renamed to: `GlobalShared_START_SEC_VAR_UNSPECI FIED` / `GlobalShared_STOP_SEC_VAR_UNSPECIF IED`
`WDGM_START_SEC_VAR_NOINIT_16BIT` / `WDGM_STOP_SEC_VAR_NOINIT_16BIT` `WDGM_START_SEC_VAR_NOINIT_8BIT` / `WDGM_STOP_SEC_VAR_NOINIT_8BIT`	Set up manually, e.g. in `MemMap.h`. Used only for AUTOSAR Debugging. Must be read/write accessible from the WdgM main functions executed on each processor core.

Table 3-5   Module variables with MICROSAR Os Gen6 / AUTOSAR Os version 4.0

3.3.3.2   Module Variables with MICROSAR Os Gen7 / AUTOSAR Os version 4.2

Following memory sections need to be set up for WdgM's module variables:

Section	Description
`WDGM_GLOBAL_START_SEC_VAR_32BIT_ COREn_PRIVATE` / `WDGM_GLOBAL_STOP_SEC_VAR_32BIT_ COREn_PRIVATE` `WDGM_GLOBAL_START_SEC_VAR_NOINIT_ UNSPECIFIED_COREn_PRIVATE` / `WDGM_GLOBAL_STOP_SEC_VAR_NOINIT_ UNSPECIFIED_COREn_PRIVATE`	If the ECU description parameter `WdgMGlobalMemoryAppTaskRef` is set, then these sections are renamed according to the configured OS application. "`WDGM_GLOBAL_START_SEC_` / `WDGM_GLOBAL_STOP_SEC_`" is converted to "`OS_START_SEC_`"`<OSApp>_` / `OS_STOP_SEC_`"`<OSApp>_`", where `<OSApp>` is the name of the OS application; suffix "`_COREn_PRIVATE`" is removed. This is generated as part of `WdgM_OsMemMap.h`.
`WDGM_GLOBAL_SHARED_START_SEC_VAR_ NOINIT_UNSPECIFIED` / `WDGM_GLOBAL_SHARED_STOP_SEC_VAR_ NOINIT_UNSPECIFIED`	These sections are always assigned in the generated file `WdgM_OsMemMap.h` to OS sections and renamed to: `OS_START_SEC_GLOBALSHARED_VAR_NOIN IT_UNSPECIFIED` / `OS_STOP_SEC_GLOBALSHARED_VAR_NOINI T_UNSPECIFIED`

Section	Description
`WDGM_START_SEC_VAR_NOINIT_16BIT` / `WDGM_STOP_SEC_VAR_NOINIT_16BIT` `WDGM_START_SEC_VAR_NOINIT_8BIT` / `WDGM_STOP_SEC_VAR_NOINIT_8BIT`	Set up manually, e.g. in `MemMap.h`. Used only for AUTOSAR Debugging. Must be read/write accessible from the WdgM main functions executed on each processor core.

Table 3-6   Module variables MICROSAR Os Gen7 / AUTOSAR Os version 4.2

3.3.4   Supervised Entity Variables

3.3.4.1   Supervised Entity Variables with MICROSAR Os Gen6 / AUTOSAR Os version 4.0

Following memory sections need to be set up for WdgM's supervised entity variables:

Section	Description
`WDGM_SEi_START_SEC_VAR_NOINIT_32BIT_COREn_PRIVATE` / `WDGM_SEi_STOP_SEC_VAR_NOINIT_32BIT_COREn_PRIVATE` `WDGM_SEi_START_SEC_VAR_NOINIT_UNSPECIFIED_COREn_PRIVATE` / `WDGM_SEi_STOP_SEC_VAR_NOINIT_UNSPECIFIED_COREn_PRIVATE`	If the ECU description parameter `WdgMAppTaskRef` corresponding to supervised entity with ID "`i`" configured for core ID "`n`" is set, then these sections are renamed according to the configured OS application (prefix "`WDGM_SEi_`" is converted to "`<OSApp>_`", where `<OSApp>` is the name of the OS application; suffix "`_COREn_PRIVATE`" is removed) and generated as part of `WdgM_OsMemMap.h`.

Table 3-7   Supervised Entity Variables MICROSAR Os Gen6 / AUTOSAR Os version 4.0

3.3.4.2   Supervised Entity Variables with MICROSAR Os Gen7 / AUTOSAR Os version 4.2

Following memory sections need to be set up for WdgM's supervised entity variables:

Section	Description
`WDGM_SEi_START_SEC_VAR_NOINIT_32BIT_COREn_PRIVATE` / `WDGM_SEi_STOP_SEC_VAR_NOINIT_32BIT_COREn_PRIVATE` `WDGM_SEi_START_SEC_VAR_NOINIT_UNSPECIFIED_COREn_PRIVATE` / `WDGM_SEi_STOP_SEC_VAR_NOINIT_UNSPECIFIED_COREn_PRIVATE`	If the ECU description parameter `WdgMAppTaskRef` corresponding to supervised entity with ID "`i`" configured for core ID "`n`" is set, then these sections are renamed according to the configured OS application. "`WDGM_SEi_START_SEC_` / `WDGM_SEi_STOP_SEC_`" is converted to "`OS_START_SEC_`"`<OSApp>_` / `OS_STOP_SEC_`"`<OSApp>_`", where `<OSApp>` is the name of the OS application; suffix "`_COREn_PRIVATE`" is removed. This is generated as part of `WdgM_OsMemMap.h`.

Table 3-8   Supervised Entity Variables MICROSAR Os Gen7 / AUTOSAR Os version 4.2

3.4   Timing Setup

This chapter describes the WdgM timing configuration parameters. The timing of the WdgM is defined by

> the calling period of function `WdgM_MainFunction()`

> the count period of the WdgM tick counter (for deadline supervision)

Every time when the function `WdgM_MainFunction()` is invoked

> the alive counters are evaluated

> running deadlines are checked for violations

> checkpoint fault indications are evaluated and, finally

> the WdgM global status of all supervised entities is calculated

Note
The time period during which the function `WdgM_MainFunction()` is called, is the WdgM supervision cycle. This cycle time is also used for the periodic setting of the trigger condition of the Watchdog device. The period of this cycle determines the shortest WdgM reaction time. For example: If the WdgM reaction time should be not more than 10 ms, the supervision cycle time should be set to 10 ms or shorter.

Note
For a multi-core system, the calling period and the count period might be configured differently for the WdgM instance running on each core. For reasons of simplicity, this section covers the case for one processor core only. The WdgM instances in a multi-core core setup act independently of each other.

Figure 3-3 shows the WdgM timing configuration parameters. The parameters can be set by a Configuration Tool.
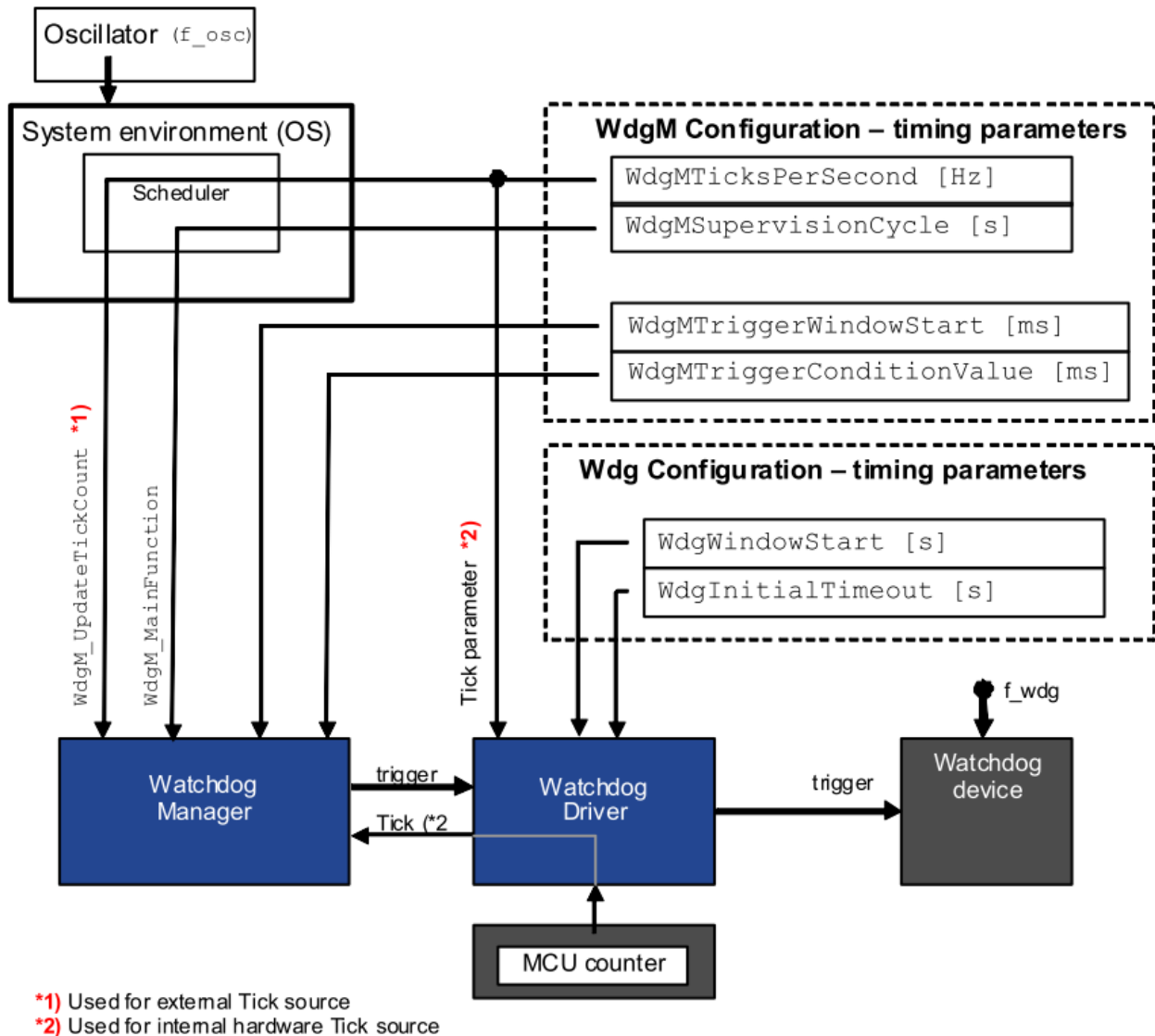
Figure 3-3   Time base of WdgM

Two configuration parameters shown in Figure 3-3 are used by the System Environment only. The Scheduler uses these parameters and periodically calls

> function `WdgM_MainFunction()` and

> if defined, also function `WdgM_UpdateTickCounter()`

All the other parameters are used by the WdgM and Wdg.

Configuration Parameter	Description
WdgMSupervisionCycle	This parameter defines the time period in which the WdgM performs cyclic supervision. This is the time period in which function WdgM_MainFunction() is called. The user of this parameter is the system environment that periodically calls function WdgM_MainFunction().
WdgMTicksPerSecond	This parameter defines the frequency by which the WdgM tick counter is incremented.

Configuration Parameter	Description
	> If the external tick counter is selected, the user of this parameter is the system environment that periodically calls function WdgM_UpdateTickCount().
	> If the OsCounter is selected, the user has to configure an OsCounter and reference the OsCounter from the WdgMOsCounterRef. The parameter WdgMTicksPerSecond will be configured automatically according to the OsCounter configuration.
	> If the internal software timebase is selected, the user only has to configure the WdgMSupervisionCylce. The parameter WdgMTicksPerSecond will be configured automatically according to the supervision cylce configuration.
	> The parameter WdgMTicksPerSecond must not be zero.
WdgMTriggerWindowStart	This parameter is actually not used and should be set to 0.
WdgMTriggerConditionValue	This parameter defines, for all supervision cycles (except for the first), the upper limit of the Watchdog trigger window. If the Watchdog is not triggered in time, a reset is caused. This parameter is in milliseconds. The user is the WdgM.

Table 3-9   Configuration Parameters

3.4.1   Deadline Measurement and Tick Counter

The transition time between two checkpoints is measured in ticks. The tick counter delivers a time base for deadline supervision. The tick counter is the smallest deadline time unit for the WdgM. There are three possible tick sources (see Figure 3-4   WdgM   Tick   source selection for deadline supervision):

> Internal software tick source: The tick source is software-based where the internal counter is incremented every time the WdgM main function (WdgM_MainFunction()) is called. If the internal software tick source is selected, the frequency (WdgMTicksPerSecond) is the same as WdgM_MainFunction() is called.

> External tick source: The tick must be counted externally by calling the WdgM function WdgM_UpdateTickCount(). If the external tick source is selected, the system integrator is responsible for calling this function on a regular basis. The WdgM internally checks if the number of ticks corresponds with the supervision cycle.

> OsCounter: The Os is responsible for the counter. If the OsCounter is selected as source, the system integrator is responsible for configuring the OsCounter properly. The WdgM internally checks if the number of ticks corresponds with the supervision cycle.

> **Note**
> The tick source can be selected by setting the parameter `WdgMTimebaseSource`. The default parameter value is `WDGM_INTERNAL_SOFTWARE_TICK`.

> **Note**
> When you configure a multi-core system, it is possible to select only one tick source for all the processor cores. However, ticks per second can be different.
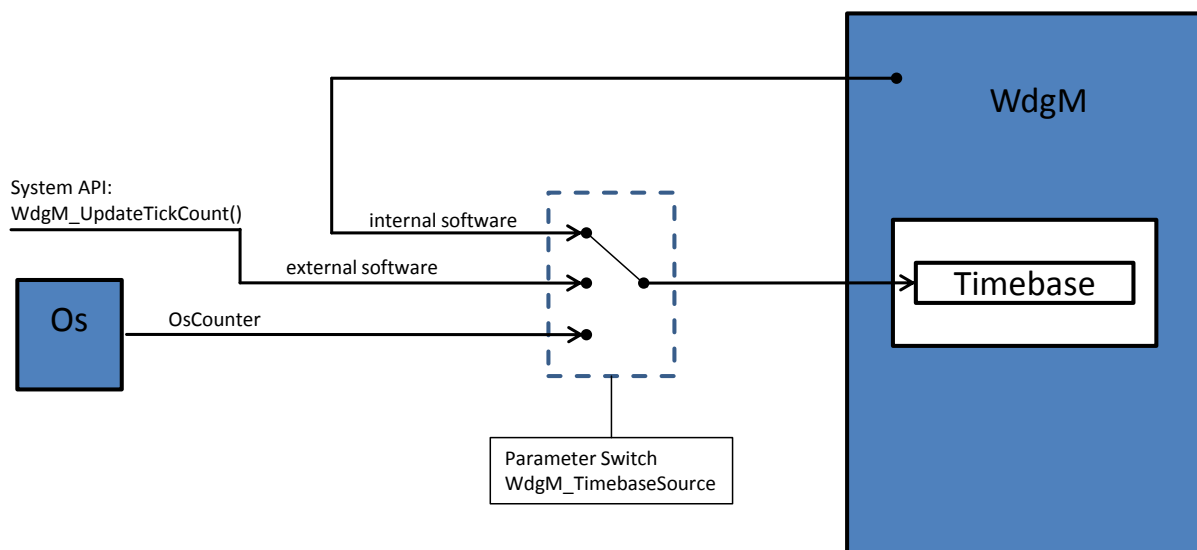
Figure 3-4   WdgM Tick source selection for deadline supervision

The ticks per second must be configured for the WdgM to translate the monitored deadlines from seconds (as stored in the AUTOSAR ECU description files) to WdgM ticks. This conversion is done during configuration generation for the WdgM, with the deadlines being stored in the generated configuration as WdgM ticks.

> **Note**
>
> > Non-integer ticks are not allowed. If a deadline cannot be converted into an integer number of WdgM ticks, the WdgM configuration generator will report an error.
>
> > For an internal software tick source and an external tick source the internal tick counter is initialized to 1.
>
> Examples
>
> > Let a WdgM tick be 2 ms. If a deadline is 3 ms, it cannot be converted to WdgM ticks without loss of accuracy. It will be between 1 and 2 WdgM ticks.
>
> > Let a WdgM tick be 1 ms (i.e. the parameter `WdgMTicksPerSecond` is set to `1000`). A deadline of `0.002s=2ms` is then translated to 2 WdgM ticks. But a deadline of `0.0005s=0.5ms` cannot be translated to an integer number of WdgM ticks.

> **Note**
> There is a trade-off between the WdgM tick resolution and performance. The shorter the tick length, the finer the deadlines that can be monitored. However, the performance gets worse due to more frequent calls to the `WdgM_UpdateTickCount()` function.

3.5 Using Checkpoints in Interrupts

Generally, the call of the function `WdgM_CheckpointReached()` is not restricted to a specific context. However, if it is called from an interrupt, the system designer must be aware of the following:

> All checkpoints of the supervised entity which runs in the interrupt context must be called from the same interrupt and never outside of it. This is because the function `WdgM_CheckpointReached()` is allowed to interrupt itself only if called for different supervised entities.

> The runtime of the function `WdgM_CheckpointReached()` must be considered. Note that the runtime can vary depending on the platform and the complexity of the referenced supervised entity.

> The function `WdgM_CheckpointReached()` requests to disable/enable interrupts (by calling e.g. `SchM_Exit_WdgM()`/`SchM_Enter_WdgM()`) – the usage of disable/enable interrupt routines must be allowed out of the interrupt context.

> The interrupt context must have read/write access to the global shared memory (memory mapping defines `WDGM_GLOBAL_SHARED_START_SEC_VAR_NOINIT_*`).

> The interrupt context must have read/write access to referenced supervised entity local memory (memory mapping defines `WDGM_SEn_START_SEC_VAR_NOINIT_*`, where n is the supervised entity ID provided to the function

`WdgM_CheckpointReached()`). The same rules apply to this SE local memory – it might be write accessible from contexts which have the same quality level as the interrupt context or higher, but it must be protected from all other contexts.

3.6 Integration into a Multi-Core System

The WdgM can be used on a single core and on multiple cores simultaneously. In order to achieve this task in a more generic and hardware-independent way inter-core communication is avoided. Each processor core on which the WdgM needs to do its monitoring runs a separate WdgM instance. Each WdgM instance controls one or more watchdogs. It builds an independent global state and decides on triggering its watchdogs or causing a deliberate reset. Everything that is valid for single-core integration is valid for multi-core usage as well. However, each core must be handled as a separate processor.

The integration specifics for a multi-core system are as follows:

> Each processor core runs the `WdgM_Init()` function separately with its own configuration.

> The configuration for each processor core (which contains only its settings, supervised entities, etc.) is generated in a separate configuration structure. However, the preprocessor options are common for all cores.

> Each processor core executes the `WdgM_MainFunction()` separately and periodically. The period for each processor core might be different and depends on the configuration.

> The global memory data is configured separately for each processor core and must be accessible from this core and the application that is responsible for running the `WdgM_MainFunction()`.

> The global shared memory section must be accessible by all processor cores.

3.7 States

See WdgM Local Entity State (2.3.12) and WdgM Global State (2.3.13).

3.8 Main Functions

See WdgM Supervision Cycle (2.3.8).

3.9 Error Handling

3.9.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Appl_Det_ReportError()` as specified in [2], if development error reporting is enabled (i.e. pre-compile parameter `WdgM_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 `Appl_Det_ReportError ()`.

The reported WdgM ID is 13.

The reported service IDs identify the services which are described in 5.2. The following table presents the service IDs and the related services:

Service ID	Service
0x00	WdgM_Init
0x02	WdgM_GetVersionInfo
0x03	WdgM_SetMode
0x05	WdgM_ActivateSupervisionEntity
0x06	WdgM_DeactivateSupervisionEntity
0x08	WdgM_MainFunction
0x0B	WdgM_GetMode
0x0C	WdgM_GetLocalStatus
0x0D	WdgM_GetGlobalStatus
0x0E	WdgM_CheckpointReached
0x0F	WdgM_PerformReset
0x10	WdgM_GetFirstExpiredSEID
0x12	WdgM_UpdateTickCount
0x13	WdgM_GetFirstExpiredSEViolation

Table 3-10   Service IDs

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

Error Code	Description
(0x10u)	API service called if WdgM uninitialized
(0x11u)	API service WdgM_Init() called with wrong parameter
(0x12u)	API service WdgM_SetMode() called with wrong parameter
(0x13u)	API service WdgM_Init() called and no supervised entity is configured API service called with wrong supervised entity id
(0x14u)	API service called with NULL_PTR as parameter
(0x15u)	API service WdgM_Init() called and a trigger mode is erroneously configured to be OFF and OFF mode is not allowed
(0x16u)	API service WdgM_Init() called and on checkpoint is configured in a supervised entity API service WdgM_CheckpointReached() called with wrong checkpoint id
(0x17u)	Not used
(0x28u)	API service WdgM_MainFunction() detected 'stuck-in' or 'negative jump' of timebase tick counter or timebase tick counter is out of configured range

Error Code	Description
(0x29u)	API services WdgM_MainFunction() or WdgM_CheckpointReached is called and local / global status is undefined
(0x2Au)	API services of WdgIf called and return value is E_NOT_OK
(0x2Bu)	API service WdgM_MainFunction() detected memory corruption
(0x2Cu)	API service WdgM_MainFunction() called while already invoked
(0x2Du)	Supervised entity shall be deactivate while supervised entity is active
(0x2Eu)	API service and invalid processor core id is determined within the service

Table 3-11   Errors reported to DET

3.9.2   Production Code Error Reporting

By default, production code related errors are reported to the DEM using the service `Appl_Dem_ReportErrorStatus()` as specified in [3], if production error reporting is enabled (i.e. pre-compile parameter `WDGM_DEM_REPORT==STD_ON`).

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

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

Error Code	Description
WDGM_E_IMPROPER_CALLER	Service WdgM Set Mode called with invalid caller id.
WDGM_E_MONITORING	Monitoring has failed (a watchdog reset will occur).

Table 3-12   Errors reported to DEM

4 Integration

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

4.1 Scope of Delivery

The delivery of the WdgM contains the files which are described in the chapters 4.1.1 and 4.1.2.

4.1.1 Static Files

File Name	Description
WdgM.c	Implementation of the WdgM, defines the API for the Service Layer of the BSW-Layer.
WdgM_Checkpoint.c	Implementation of the WdgM, defines the API for the Application Layer.
WdgM.h	Header file of the WdgM, provides API function declarations.
WdgM_Cfg.h	Provides defines and declarations for the WdgM configuration identifiers.

Table 4-1　Static files

4.1.2 Dynamic Files

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

File Name	Description
WdgM_PBcfg.c	This file contains the main configuration structure with the default name WdgMConfig_Mode0. This configuration name should be used by the initialization function, i.e. by call WdgM_Init(&WdgMConfig_Mode0). If necessary, the non-standard AUTOSAR name WdgMConfig_Mode0 can be renamed to WdgMConfigSet in the Configuration Tool (e.g., DaVinci).
WdgM_PBcfg.h	The file contains the declaration of the WdgM configuration.
WdgM_OSMemMap.h	The file contains defines of all used / necessary memory sections.
WdgM_Cfg_Features.h	The file contains WdgM precompile directives.

Table 4-2　Generated files

4.2   Critical Sections

The WdgM implements the following critical section:

> `WDGM_EXCLUSIVE_AREA_0`: This critical section is used to protect all uninterruptable sequences. It shall lock all interrupt sources and task switches.

5 API Description

The WdgM software module is the top level layer of the Watchdog Manager Stack. The WdgM software module contains the core functionality with supervised entity state machines and calculation of the WdgM global state. The WdgM communicates on one side through its user API with the Application Layer (optionally using RTE) and through its system API with the Basic Software Components (BSW) and, on the other side, with the WdgIf layer.

5.1 Type Definitions

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

Type Name	C-Type	Description	Value Range
WdgM_ConfigType	struct	This is the type for the WdgM configuration structure. This structure is generated by the WdgM configuration generator.	N/A
WdgM_SupervisedEntityIdType	uint16	This is the type for an individual supervised entity for the Watchdog Manager. **Note:** If configuration parameter WDGM_USE_RTE is set to STD_ON, then this type is imported, otherwise it is generated.	0...65534
WdgM_CheckpointIdType	uint16	This is the type for a checkpoint in the context of a supervised entity for the WdgM. **Note:** If configuration parameter WDGM_USE_RTE is set to STD_ON, then this type is imported, otherwise it is generated.	0...65534
WdgM_ModeType	uint8	This is the type for the ID of a trigger mode that was configured for the WdgM. The current trigger mode can be retrieved with WdgM_GetMode(). **Note:** If configuration parameter WDGM_USE_RTE is set to STD_ON, then this type is imported, otherwise it is generated.	0...255
WdgM_LocalStatusType	uint8	This is the type for the local monitoring state of a supervised entity. The current local state of a supervised entity can be retrieved with WdgM_GetLocalStatus(). **Note:** If configuration parameter WDGM_USE_RTE is set to STD_ON, then this type is imported, otherwise it is generated.	WDGM_LOCAL_STATUS_OK = 0 WDGM_LOCAL_STATUS_FAILED = 1 WDGM_LOCAL_STATUS_EXPIRED = 2 WDGM_LOCAL_STATUS_DEACTIVATED = 4
WdgM_GlobalStatusType	uint8	This is the type for the global monitoring state. It summarizes the local states of all supervised entities. The current global state can be retrieved with WdgM_GetGlobalStatus().	WDGM_GLOBAL_STATUS_OK = 0, WDGM_GLOBAL_STATUS_FAILED = 1, WDGM_GLOBAL_STATU

Type Name	C-Type	Description	Value Range
		Note: If configuration parameter WDGM_USE_RTE is set to STD_ON, then this type is imported, otherwise it is generated.	S_EXPIRED = 2, WDGM_GLOBAL_STATUS_STOPPED = 3, WDGM_GLOBAL_STATUS_DEACTIVATED = 4
Std_VersionInfo Type	struct	This is the parameter type of function WdgM_GetVersionInfo()	N/A
WdgM_Violation Type	uint8	Used with AUTOSAR Debugging (parameter WdgMAutosarDebugging). This parameter is the parameter type of function WdgM_GetFirstExpiredSEViolation()	WDGM_VIOLATION_NONE: No violations WDGM_VIOLATION_PF: Program flow violation WDGM_VIOLATION_DM: Deadline supervision violation WDGM_VIOLATION_AS: Alive supervision violation WDGM_VIOLATION_PF_DM: Program flow and deadline supervision violations WDGM_VIOLATION_PF_AS: Program flow and alive supervision violations WDGM_VIOLATION_DM_AS: Deadline supervision and alive supervision violations WDGM_VIOLATION_PF_DM_AS: Program flow, deadline supervision and alive supervision violationsmonitoring and alive supervision violations

Table 5-1   Type definitions

5.2   Services provided by WdgM

5.2.1   WdgM_Init

Prototype	
void **WdgM_Init**(const WdgM_ConfigType* WdgMConfigPtr)	
Parameter	
WdgMConfigPtr	Pointer to post-build configuration data
Return code	
void	

Functional Description
The `WdgM_Init()` function initializes the WdgM. After the execution of this function, monitoring is activated according to the configuration of `ConfigPtr`. This function can be used during monitoring, too, but note that all pending violations are lost.
Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is non-reentrant.
Expected Caller Context
> This service is expected to be called in application context.

Table 5-2   WdgM_Init

5.2.2   WdgM_GetVersionInfo

Prototype
void **WdgM_GetVersionInfo** (Std_VersionInfoType* VersionInfo)
Parameter
VersionInfo	Pointer to where to store the version information of the WdgM module.
Return code
void	
Functional Description
The WdgM_GetVersionInfo() function returns information about the version of this module. This includes the module ID, the vendor ID, and the vendor-specific version number.
Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is reentrant.

Table 5-3   WdgM_GetVersionInfo

5.2.3   WdgM_SetMode

Prototype
Std_ReturnType **WdgM_SetMode** (WdgM_ModeType Mode, uint16 CallerID)
Parameter
Mode	The ID of the Trigger Mode to which the WdgM must be set.
CallerID	ID of the caller allowed to call the function `WdgM_SetMode()`. The allowed caller is defined in the configuration. The caller ID is checked if `WdgMDefensiveBehavior` is true.

Return code	
Std_ReturnType	E_OK: The new Trigger Mode has been successfully set.
	E_NOT_OK: The setting of the new Trigger Mode failed.

Functional Description
This functions sets the Trigger Mode of the WdgM. The WdgM Trigger Mode is a set of Watchdog trigger times and Watchdog mode. The WdgM can have one or more Trigger Modes for every watchdog. In contrast to AUTOSAR, where the Mode represents a set of entities with all entity-specific parameters, the WdgM Trigger Mode only sets the following parameters: **>** WdgMTriggerConditionValue **>** WdgMTriggerWindowStart **>** WdgMWatchdogMode **Note:** A change to trigger mode with ID Mode sets all configured watchdogs to the trigger mode with ID Mode. As a consequence, all watchdogs must have configured the same number of Trigger Modes. This function can be used to increase the WdgM supervision cycle in an MCU sleep mode.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1) **>** This function is asynchronous. **>** This function is reentrant.

Table 5-4   WdgM_SetMode

5.2.4   WdgM_ActivateSupervisionEntity

Prototype	
Std_ReturnType **WdgM_ActivateSupervisionEntity** (WdgM_SupervisedEntityIdType SEID)	
Parameter	
SEID	Supervised entity identifier.
Return code	
Std_ReturnType	E_OK: Marking the supervised entity for activation was successful.
	E_NOT_OK: Marking the supervised entity for activation failed.

Functional Description
The function marks an entity for activation. An entity can only be activated when its local state is WDGM_LOCAL_STATUS_DEACTIVATED. The activation itself happens at the end of the supervision cycle inside the `WdgM_MainFunction()`.

Note:

> This function can degrade system safety. The activation of entity supervision in safety-related products needs special attention to avoid unintended supervised entity deactivation.

> In the same call of `WdgM_MainFunction()`, first the local states of all supervised entities and the global state are set, then the supervised entity is activated.

> After SE activation the function `WdgM_GetLocalStatus()` can be used to check the SE local state.

> This function is only available if the preprocessor switch `WdgMEntityDeactivationEnabled` is set to true and if the entity option `WdgMEnableEntityDeactivation` is set to true.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is asynchronous.
> This function is reentrant (for different `SEID`).
> This function is an extension of the AUTOSAR specification

Table 5-5   WdgM_ActivateSupervisionEntity

5.2.5   WdgM_DeactivateSupervisionEntity

Prototype
Std_ReturnType **WdgM_DeactivateSupervisionEntity** (WdgM_SupervisedEntityIdType SEID)

Parameter	
SEID	ID of the supervised entity to be deactivated. Range `[0...N]`

Return code	
Std_ReturnType	E_OK: Marking the supervised entity for deactivation was successful.
	E_NOT_OK: Marking the supervised entity for deactivation failed.

Functional Description

The function marks an entity for deactivation. An entity can only be deactivated when its local state is `WDGM_LOCAL_STATUS_OK` or `WDGM_LOCAL_STATUS_FAILED`. The deactivation itself happens at the end of the supervision cycle inside the `WdgM_MainFunction()`. When an entity is deactivated then its checkpoints are not evaluated anymore and the entity local state is `WDGM_LOCAL_STATUS_DEACTIVATED`.

Note:

> When an entity is deactivated, the global transitions to this entity are not evaluated.

> Using this function can degrade system safety. The deactivation of entity supervision in safety-related products needs special attention to avoid unintended supervised entity deactivation.

> The function `WdgM_DeactivateSupervisionEntity()` can deactivate a supervised entity only before its initial checkpoint was passed or after its end checkpoint was passed. The focus here is on entities that are spread over more than one supervision cycle.

> Note: The local program flow of a supervised entity may span over more than one supervision cycle. Those active entities cannot be deactivated while running. Deactivating active SEs leads to a DEM error report.

> In the same call of `WdgM_MainFunction()`, first the supervised entity is deactivated, then the local states of all supervised entities and the global state are set.

> After SE deactivation the function `WdgM_GetLocalStatus()` can be used to check the SE local state.

> This function is only available if the preprocessor switch `WdgMEntityDeactivationEnabled` is set to true and if the entity option `WdgMEnableEntityDeactivation` is set to `true`.

Particularities and Limitations

> Service ID: see table 'Service IDs' (chapter 3.9.1)

> This function is asynchronous.

> This function is reentrant (for different `SEID`).

> This function is an extension of the AUTOSAR specification

Table 5-6   WdgM_DeactivateSupervisionEntity

5.2.6   WdgM_MainFunction

Prototype
void **WdgM_MainFunction**(void)
Parameter
void	
Return code
void	

Functional Description

This function evaluates monitoring data gathered from the hit checkpoints in all supervised entities during the supervision cycle. Depending on the violation found (if there is any), the

> local state of the supervised entities and

> the WdgM global state

are determined again.

Depending on the resulting global state:

> the Wdg is triggered, or

> the Wdg trigger discontinues (safe state), or

> the Wdg is reset (safe state).

The function must run at the end of every supervision cycle. It may be called by the Basic Software Scheduler or a task with a fixed period time.

The `WdgM_MainFunction()` function is not reentrant. To prevent data inconsistency when it is interrupted by itself (e.g. due to schedule overload), the function checks if it is executed concurrently. If this function is started before its last instance has finished, it raises a development error.

Note:

> Alive counter violations are detected at the end of every alive supervision reference cycle,

> Program flow violations are detected at the end of every supervision cycle,

> Continued program flow violations are detected at the end of every program flow supervision cycle.

> Deadline violations are detected at the end of every supervision cycle,

> Continued of deadline violations are detected at the end of every deadline supervision cycle.

Particularities and Limitations

> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is non-reentrant.
> This service is always available.

Table 5-7   WdgM_MainFunction

5.2.7   WdgM_GetMode

Prototype	
`Std_ReturnType` **`WdgM_GetMode`**`(WdgM_ModeType* Mode)`	
Parameter	
`Mode`	Pointer to the current Trigger Mode ID of the Watchdog Manager
Return code	
`Std_ReturnType`	`E_OK`: Current Trigger Mode successfully returned.
	`E_NOT_OK`: Returning current Trigger Mode failed.

Functional Description
Returns the current Trigger Mode of the WdgM. The WdgM Trigger Mode represents one Watchdog trigger time and mode setting.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is reentrant.

Table 5-8   WdgM_GetMode

5.2.8   WdgM_GetLocalStatus

Prototype	
Std_ReturnType **WdgM_GetLocalStatus** (WdgM_SupervisedEntityIdType SEID, WdgM_LocalStatusType* Status)	
Parameter	
SEID	Identifier of the supervised entity whose monitoring state is returned.
Status	Pointer to the local monitoring state of the given supervised entity.
Return code	
Std_ReturnType	E_OK: Current monitoring state successfully returned.
	E_NOT_OK: Returning the current monitoring state failed.
Functional Description	
Returns the monitoring state of the given supervised entity. **Note:** The WdgM updates the state inside the WdgM_MainFunction() every supervision cycle.	
Particularities and Limitations	
> Service ID: see table 'Service IDs' (chapter 3.9.1)	
> This function is synchronous.	
> This function is reentrant.	

Table 5-9   WdgM_GetLocalStatus

5.2.9   WdgM_GetGlobalStatus

Prototype	
Std_ReturnType **WdgM_GetGlobalStatus** (WdgM_GlobalStatusType* Status)	
Parameter	
Status	Pointer to global monitoring state of the WdgM.
Return code	
Std_ReturnType	E_OK: Current global monitoring state successfully returned.
	E_NOT_OK: Returning the current global monitoring state failed.

Functional Description
Returns the global monitoring state of the WdgM.
Note: The WdgM updates the state inside the `WdgM_MainFunction()` every supervision cycle.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is reentrant.

Table 5-10   WdgM_GetGlobalStatus

5.2.10 WdgM_CheckpointReached

Prototype	
Std_ReturnType **WdgM_CheckpointReached** (WdgM_SupervisedEntityIdType SEID, WdgM_CheckpointIdType CheckpointID)	
Parameter	
SEID	Identifier of the supervised entity that reports a checkpoint.
CheckpointID	Identifier of the checkpoint within a supervised entity that has been reached.
Return code	
Std_ReturnType	E_OK: Checkpoint monitoring successful.
	E_NOT_OK: Checkpoint monitoring fault. Returned in the following cases
	> WDGM_E_NO_INIT: Uninitialized WdgM (DET code 0x10)
	> WDGM_E_PARAM_SEID: Wrong Id number of the supervised entity (DET code 0x13)
	> WDGM_E_CPID: Invalid checkpoint ID number (DET code 0x16)
	> WDGM_E_PARAM_STATE: Invalid WdgM state. Reset will be invoked (DET code 0x29).

Functional Description
Indicates to the WdgM that a checkpoint within a supervised entity has been reached.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is reentrant (in the context of a different supervised entity).

Table 5-11   WdgM_CheckpointReached

5.2.11 WdgM_PerformReset

Prototype	
Std_ReturnType **WdgM_PerformReset**(void)	
Parameter	
void	

Return code	
Std_ReturnType	E_OK: This value will not be returned because the reset is activated, and the routine does not return. E_NOT_OK: The function has failed.

Functional Description
Instructs the WdgM to cause an immediate watchdog reset. **Note:** This function is hardware-dependent. Some watchdogs do not support an immediate reset. Check the Wdg Driver documentation. This function can require direct access to hardware registers. Access to hardware registers can be dependent on hardware platforms and software architectures. Hence, the application that calls WdgM_PerformReset() must have the corresponding access rights.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1) > This function is synchronous. > This function is non-reentrant. > Other particularities, limitations, post-conditions, pre-conditions

Table 5-12   WdgM_PerformReset

5.2.12 WdgM_GetFirstExpiredSEID

Prototype
Std_ReturnType **WdgM_GetFirstExpiredSEID** (WdgM_SupervisedEntityIdType* SEID)

Parameter	
SEID	A pointer to a variable that stores the ID of the first SE which has made a transition to the state WDGM_LOCAL_STATUS_EXPIRED or 0 if the function did not execute correctly.

Return code	
Std_ReturnType	E_OK: The function could extract the record for the first expired supervised entity successfully. E_NOT_OK: An error was detected (input parameter or memory corruption of the record)

Functional Description
This function returns the ID of the first SE that reached the expired state and, thus, is potentially responsible for a system reset. It must be executed after at least one SE reached the expired state, e.g. after a reset, otherwise the returned result might not be correct. **Note:** The record for the first expired SE is stored double inverse (so that memory corruption can be detected) and in a variable section that is not initialized (to preserve the data after a reset, but this also means that there is initially no valid entry).

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1) > This function is synchronous. > This function is non-reentrant.

Table 5-13   WdgM_GetFirstExpiredSEID

5.2.13 WdgM_GetFirstExpiredSEViolation

Prototype
`Std_ReturnType` **`WdgM_GetFirstExpiredSEViolation`** `(WdgM_ViolationType* ViolationType)`

Parameter	
`ViolationType`	A pointer to a variable that stores the violation type that caused the first SE to make a transition to state `WDGM_LOCAL_STATUS_EXPIRED` or `0` if the function did not execute correctly. This parameter shows if the violation was a program flow violation, a deadline supervision violation, an alive counter violation, or a combination between them.

Return code	
`Std_ReturnType`	`E_OK`: The function was able to successfully extract the record for the first violation type.
	`E_NOT_OK`: An error was detected (input parameter or memory corruption of the record).

Functional Description
This function returns the violation type of the first supervised entity which reached the expired state – and thus is potentially responsible for a system reset. It must be executed after at least one supervised entity reached the expired state, e.g. after a reset, otherwise the returned result might not be correct. Note, that the record for the violation type is stored double inverse (so that memory corruption can be detected) and in a variable section which is not initialized (to preserve the data after a reset, but this also means that initially there is no valid entry).
This function is enabled with the configuration option `WdgMAutosarDebugging`.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is non-reentrant.

Expected Caller Context
> Specify if it might be called from interrupt context

Table 5-14   WdgM_GetFirstExpiredSEViolation

5.2.14 WdgM_UpdateTickCount

Prototype
`void` **`WdgM_UpdateTickCount`**`(void)`

Parameter	
`void`	

Return code	
`void`	

Functional Description
This function increments the WdgM timebase tick counter by one. When the precompile configuration parameter `WdgMTimebaseSource` is set to `WDGM_EXTERNAL_TICK`, then this function needs to be called periodically from outside the WdgM.
The timebase tick counter delivers the time base for deadline supervision. In the AUTOSAR environment.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is non-reentrant.
> This function can be called, for example, from a task with fixed time period and high priority.

Expected Caller Context
> Specify if it might be called from interrupt context

Table 5-15   WdgM_UpdateTickCount

5.3   Services used by WdgM

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

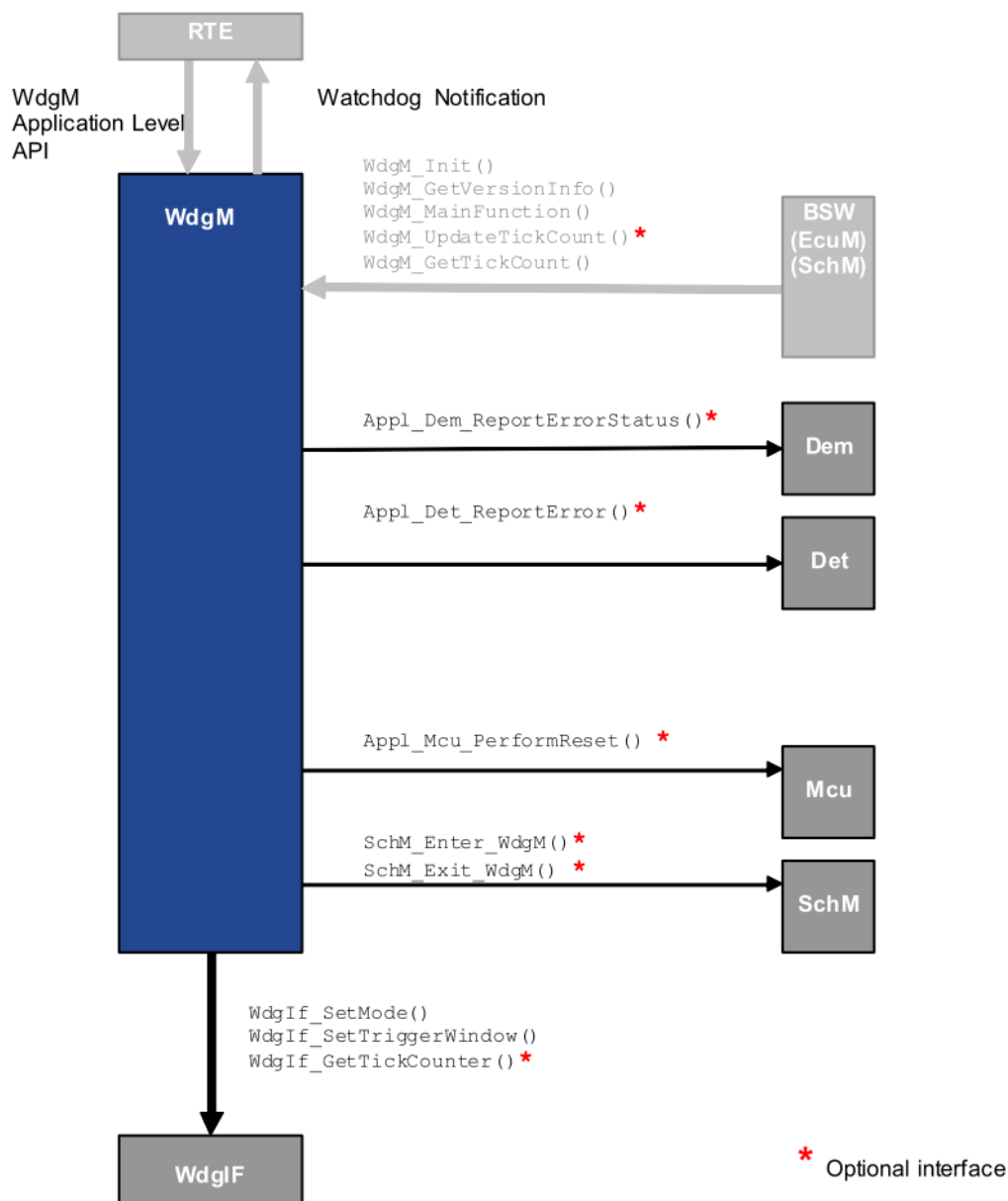| Component	API
Det	Det_ReportError()
Dem	Dem_ReportErrorStatus()
Mcu	Mcu_PerformReset()
Os	GetCoreID()
SchM	> SchM_Enter_WdgM_WDGM_EXCLUSIVE_AREA_0 > SchM_Exit_WdgM_WDGM_EXCLUSIVE_AREA_0
WdgIf	> WdgIf_SetMode() > WdgIf_SetTriggerCondition()

Table 5-16   Services used by the WdgM

Figure 5-1   Expected interfaces to external modules

	Note
i	If the precompile switches

> `WdgMDevErrorDetect`

> `WdgMDemReport`

> `WdgMUseOsSuspendInterrupt`

> `WdgMImmediateReset`

> `WDGM_SECOND_RESET_PATH`

are set to FALSE, the WdgM module does not call the corresponding function(s).

	Note
i	The functions listed in the table above may not meet the required quality level and, thus, must be wrapped in order to ensure freedom from interference with the WdgM. The integrator must implement the Appl_...() functions according to his safety requirements.

	Note
i	The system integrator must revise the necessity of the expected interfaces. A called external function may degrade the quality level of the WdgM below the required quality level.

5.4 Configurable Interfaces

5.4.1 Notifications

At its configurable interfaces the WdgM defines notifications that can be mapped to callback functions provided by other modules. The mapping is not statically defined by the WdgM but can be performed at configuration time. The function prototypes that can be used for the configuration have to match the appropriate function prototype signatures, which are described in the following sub-chapters.

5.4.1.1 Global state callback

Prototype	
void **WdgM_GlobalStateChangeCbk** (WdgM_GlobalStatusType new_state);	
Parameter	
new_state	Contains the global state after the global state change.
	Note: In a multi-core system, the global state callback function can be set up for each processor core separately.
Return code	
void	

Functional Description
If `WDGM_STATE_CHANGE_NOTIFICATION == STD_ON` and the WdgM global state changes, then the callback routine defined by the parameter `WdgMGlobalStateChangeCbk` is called. The name of the function can be arbitrary.

Particularities and Limitations
> Service ID: see table 'Service IDs' (chapter 3.9.1)
> This function is synchronous.
> This function is non-reentrant.

Expected Caller Context
> May be called from task level.

Table 5-17   Global state callback

5.4.1.2 Local state change notification

Prototype	
void **WdgM_LocalStateChangeCbk** (WdgM_LocalStatusType new_state);	
Parameter	
new_state	Contains the local state after the local state change.
Return code	
void	
Functional Description	
If WDGM_STATE_CHANGE_NOTIFICATION == STD_ON and the local state of a supervised entity changes, then the callback routine defined by the parameter WdgMLocalStateChangeCbk is called. The name of the function can be arbitrary (but of course different for each supervised entity).	
Particularities and Limitations	
> Service ID: see table 'Service IDs' (chapter 3.9.1) > This function is synchronous. > This function is non-reentrant.	
Call context	
> May be called from task level.	

Table 5-18   Local state change notification

5.5 Service Ports

A single SWC description file (WdgM_swc.arxml) is generated by the WdgM configuration generator. For each referenced OsApplication (referenced by `WdgMGlobalMemoryAppTaskRef` and `WdgMAppTaskRef`) a separate component type element is generated (named WdgM_<OsApplication>) within. If no OsApplication is referenced at all, only one component type is generated (named WdgM).

5.5.1 Client Server Interface

A client server interface is related to a Provide Port at the server side and a Require Port at client side.

The following client server interfaces with corresponding operations are available:

> WdgM_AliveSupervision

> WdgM_LocalStatus

> WdgM_General

If status reporting mechanism is configured to `WDGM_USE_MODE_SWITCH_PORTS`:

> WdgM_IndividualMode

> WdgM_GlobalMode

If status reporting mechanism is configured to `WDGM_USE_NOTIFICATIONS`:

> WdgM_LocalStatusCallbackInterface

> WdgM_GlobalStatusCallbackInterface

5.5.1.1 Provide Ports on WdgM Side

At the Provide Ports of the WdgM the API functions described in 5.2 are available as Runnable Entities. The Runnable Entities are invoked via operations. The mapping from a SWC client call to an operation is performed by the RTE. In this mapping the RTE adds port defined argument values to the client call of the SWC, if configured.

The following sub-chapters present the Provide Ports defined for the WdgM and the operations defined for the Provide Ports, the API functions related to the operations and the port defined argument values to be added by the RTE.

5.5.1.1.1 Port Prototype for WdgM_AliveSupervision

There are two possibilities for creation of a client server port prototype:

> For each checkpoint (if parameter `WdgMGenerateCPIdAsPortDefinedArgument` is set to `STD_ON`)

alive_<WdgMSupervisedEntityShortname>_<WdgMCheckpointShortname>

With this client server port prototype the following operation can be invoked:

Operation	API Function	Port Defined Argument Values
CheckpointReached	WdgM_CheckpointReached	WdgM_SupervisedEntityIdType SEID, WdgM_CheckpointIdType CPID

Table 5-19 alive_<WdgMSupervisedEntityShortname>_<WdgMCheckpointShortname>

> **>** For each supervised entity (if parameter `WdgMGenerateCPIdAsPortDefinedArgument` is set to `STD_OFF`)

alive_<WdgMSupervisedEntityShortname>

With this client server port prototype the following operation can be invoked:

Operation	API Function	Port Defined Argument Values
CheckpointReached	WdgM_CheckpointReached	WdgM_SupervisedEntityIdType SEID

Table 5-20   alive_<WdgMSupervisedEntityShortname>

5.5.1.1.2   Port Prototype for WdgM_LocalStatus

For each supervised entity a client server port prototypes is created:

localStatus_<WdgMSupervisedEntityShortname>

With this client server port prototype the following operation can be invoked:

Operation	API Function	Port Defined Argument Values
GetLocalStatus	WdgM_GetLocalStatus	WdgM_SupervisedEntityIdType SEID

Table 5-21   individual_<WdgMSupervisedEntityShortname>

5.5.1.1.3   Port Prototype for WdgM_General

This client server port prototype is created only once per core.

If an OsApplication is referenced by `WdgMGlobalMemoryAppTaskRef`, the following port prototype is created:

general_Core< WdgMModeCoreAssignment >

If no OsApplication is referenced, the following port prototype is created

general

The related client server interface is WdgM_GlobalMode. No port defined argument values are added. With this client server port prototype the following operations can be invoked:

Operation	API Function	Condition
GetMode	WdgM_GetMode	-
GetGlobalStatus	WdgM_GetGlobalStatus	-
GetLocalStatus	WdgM_GetLocalStatus	-
PerformReset	WdgM_PerformReset	-
SetMode	WdgM_SetMode	
GetFirstExpiredSEID	WdgM_GetFirstExpiredSEID	-
GetFirstExpiredSEViolation	WdgM_GetFirstExpiredSEViolation	AUTOSAR Debugging enabled
ActivateSupervisionEntity	WdgM_ActivateSupervisionEntity	EntityDeactivation enabled

Operation	API Function	Condition
DeactivateSupervisionEntity	WdgM_DeactivateSupervisionEntity	
UpdateTickCount	WdgM_UpdateTickCount	Timebase is EXTERNAL_TICK

Table 5-22   global_<WdgMGlobalMemoryAppTaskRefShortname> / global_WdgM

5.5.1.2   Require Ports on WdgM Side

At its Require Ports the WdgM calls operations. These operations have to be provided by the SWCs by means of Runnable Entities. These runnable entities implement the callback functions expected by the WdgM.

The following sub-chapter present the Require Ports defined for the WdgM, the operations that are called from the WdgM and the related notifications, which are described in chapter 5.4.

5.5.1.2.1   Port Prototype for WdgM_LocalStatusCallbackInterface

If a callback function is configured for a supervised entity (`WdgMLocalStateChangeCbk`), for each of those supervised entities a client server port prototypes is created:

localStateChangeCbk_<WdgMSupervisedEntityShortname>

With this client server port prototype the following operation shall be invoked:

Operation	Notification
LocalStatusCallback	Local state change notification

Table 5-23   localStateChangeCbk_<WdgMSupervisedEntityShortname>

5.5.1.2.2   Port Prototype for WdgM_GlobalStatusCallbackInterface

If a callback function is configured for a mode (`WdgMGlobalMemoryAppTaskRef`), for each of those modes (/ cores) a client server port prototypes is created:

globalStateChangeCbk_Core<WdgMModeCoreAssignment>

With this client server port prototype the following operation shall be invoked:

Operation	Notification
GlobalStatusCallback	Global state change notification

Table 5-24   localStateChangeCbk_<WdgMSupervisedEntityShortname>

5.5.1.3   Mode Ports on WdgM for Status Reporting

If the WdgM has Mode Ports configured, the WdgM informs applications, SWCs, etc. via these Mode Ports about status changes.

For each supervised entity a mode port prototypes is created:

mode_<WdgMSupervisedEntityShortname>

For each ConfigSet / Core a mode prototype is created:

globalmode_Core<WdgMModeCoreAssignment>

6 Configuration

6.1 Configuration Variants

The WdgM supports the configuration variants

> `VARIANT-PRE-COMPILE`

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

The WdgM can be configured using the following tool:

> DaVinci Configurator 5 (AUTOSAR 4 packages only). Parameters are explained within the tool.

The outputs of the configuration and generation process are the configuration source files.

6.2 WdgM Configuration Verification

The WdgM Verifier is a tool for the verification of the generated WdgM configuration. The WdgM Verifier is delivered as a DLL (wdgm_verifier.dll) that must be compiled with the configuration files produced by the generator and the files produced by the XSLT Processor. The compilation result is a Windows Verifier.exe program. Running the Verifier generates a report file (verifier_report.txt) that contains the result of the verification.

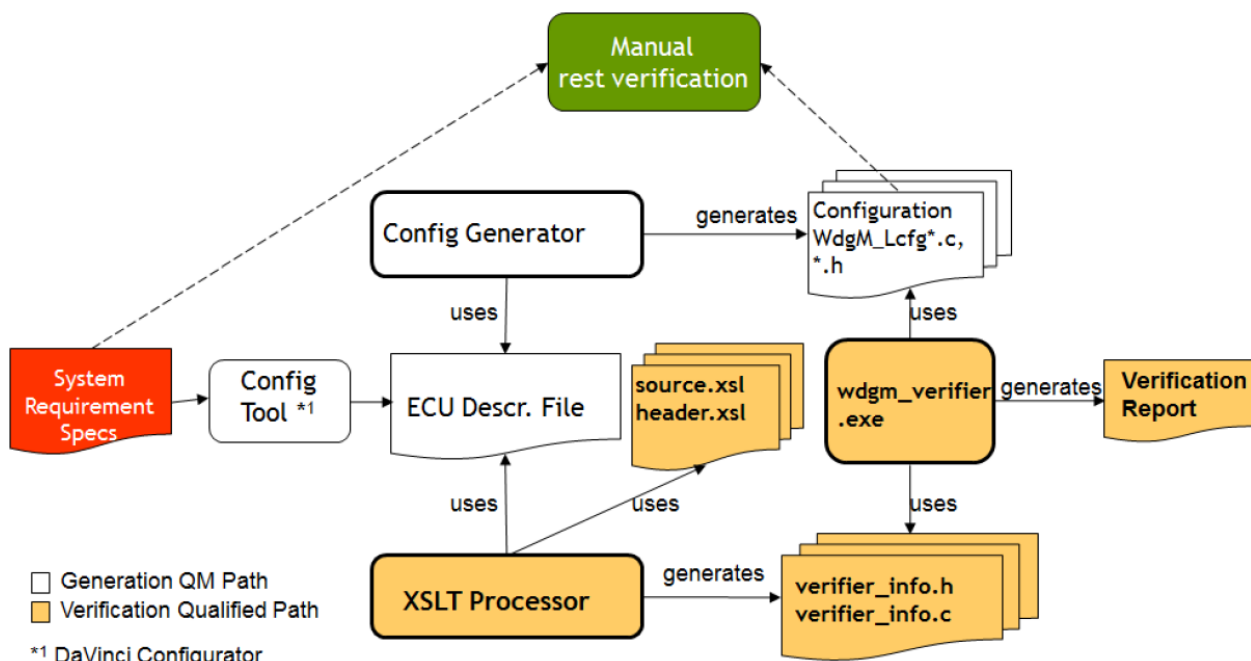Figure 6-1 shows the workflow of the WdgM Verifier build.

Figure 6-1   Workflow of the WdgM Configuration Verifier build

Note
The WdgM generator is **not** ASIL D, therefore its output cannot be trusted, hence additional checks are required by use of the WdgM Verifier.

Note
The Verifier is only content of the delivery if the WdgM is ordered in safe context.

Practical Procedure
The verification process consists of the following steps, which are explained in detail in the following sections:

> creation of WdgM Info files out of the ECU Description file (for the Verifier build),

> build (compilation) of the Verifier,

> Verifier run and manual check of the Verifier report,

> manual checks (which cannot be performed by the Verifier) and

> check of system specifications against the generated WdgM files.

6.2.1.1　Installing the WdgM Verifier

To run the WdgM Verifier an XSLT Processor and a working gcc environment are required.

The XSLT Processor is part of the delivery and located at "external\Misc\Wdg\xsltproc" and contains of following files:

> `iconv.dll,`

> `libexslt.dll,`

> `libxml2.dll,`

> `libxslt.dll,`

> `zlib1.dll,`

> `xsltproc.exe.`

The recommended way to install gcc is to install the MinGW environment with the provided installer program (`MinGW-5.1.6.exe` – located at "external\Misc\Wdg\MinGW") for Windows 7. To install gcc proceed as follows:

1. Start the installer program, accept the license terms and click "Next" until you are prompted to select a configuration.

2. When prompted, select Minimal configuration. There is no need to select any check boxes.

3. Complete the installation process after accepting the default settings.

4. Having installed gcc, add the `c:\MinGW\bin` directory to your search path by entering the command `set PATH=%PATH%;c:\MinGW\bin` in a command prompt window. Alternatively you can edit Environment Variables in the System Properties dialog (Start > Control Panel > System).

To verify that gcc is working, open a new command prompt window and enter `gcc --version` to let gcc show its version number.

6.2.1.2　Creation of WdgM Info Files

This section describes how to extract the ECU description information for the verification. The extraction results are the files

> wdgm_verifier_info.h and

> wdgm_verifier_info.c.

They contain the ECU description information. For extracting the ECU description information, the integrator shall use the XSLT processor named "xsltproc.exe" (included in the delivery). Further the following XSL stylesheets shall be applied for the information extraction:

> verify_wdgm_header.xsl and

> verify_wdgm_source.xsl

The XSL stylesheets use XSLT 1.0 features only.

The integrator shall extract the ECU description information as follows:

> apply verify_wdgm_header.xsl to the ECU description file and store the output to wdgm_verifier_info.h.

> apply verify_wdgm_source.xsl to the ECU description file and store the output to the file wdgm_verifier_info.c.

In case of xsltproc.exe, the syntax is:

> xsltproc.exe verify_wdgm_header.xsl ECU-description-file >wdgm_verifier_info.h

> xsltproc.exe verify_wdgm_source.xsl ECU-description-file >wdgm_verifier_info.c

Note
The verifier tool and all necessary files are located at "external\Generators\Wdg\Wdgm_Verifier".

6.2.1.3 Verifier Compilation

The WdgM Verifier executable Verifier.exe is created as follows.

The integrator shall use a compiler/linker that fulfills the requirements in [ISO26262], part 8, clause 11.4. Gcc 3.4.5 was tested, which fulfills the ISO26262 requirements.

The gcc compiler is part of the delivery if the WdgM was ordered in safe context. It is highly recommended to use the delivered compiler.

For the compilation process, the following files must be compiled and linked:

> Generated C file: WdgM_PBcfg.c

> Generated WdgM "Info file" (XSLT result): wdgm_verifier_info.c

> Files from the WdgM verifier package:

> > wdgm_verifier.dll

> > libwdgm_verifierdll.a

The compiled files include the following files (more files may be required for compilation depending on the environment and configuration options):

> WdgM header files:

> > WdgM.h

> > WdgM_Cfg.h

> WdgIf header file WdgIf_Types.h

> Created WdgM "Info file" (XSLT result): wdgm_verifier_info.h

> Generated WdgM header files:

> > WdgM_Cfg_Features.h

> > WdgM_OSMemMap.h

- > WdgM_PBcfg.h
- > Files from the WdgM Stack package:
  - > wdgm_verifier.h
  - > wdgm_verifier_types.h
  - > wdgm_verifier_version.h
- > List of platform specific files:
  - > Compiler.h
  - > Compiler_Cfg.h
  - > MemMap.h
  - > Os.h
  - > Os_MemMap.h
  - > Platform_Types.h
  - > Std_Types.h
  - > Rte_Compiler_Cfg.h (if RTE is used)
  - > Rte_MemMap.h (if RTE is used)
  - > Rte_Type.h (if RTE is used)

The set of include commands (-I path) for all include paths to these files is referred to as verify-includes.

Expert Knowledge
The syntax for the compilation call is:

gcc -Wall wdgm_verifier_info.c WdgM_PBcfg.c verify-includes –L dll-path –l wdgm_verifier -o Verifier.exe

where

- > *verify-includes* is a placeholder for the path(s) of include files as described above and
- > *dll-path is a placeholder for the path where wdgm_verifier.dll and libwdgm_verifierdll.a are located.*

In case of an error free application of the compiler/linker the output is a WdgM Verifier executable named Verifier.exe.

6.2.1.4 Verifier Run

After the WdgM Verifier executable has been built, it has to be executed. The WdgM Verifier writes a verification report to standard output 'stdout'. This report must be reviewed as stated in this section and manual verification check hast to be performed as described in the Safety Manual [5].

The integrator shall run the WdgM Verifier executable as follows:

> Verifier.exe >verifier_report.txt.

Caution
All other steps listed in section 6.2 are described in the Safety Manual [5].

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
Alive Indications	An indication provided by a supervised entity alive counter to signal its aliveness to the WdgM.
Alive Supervision	A kind of WdgM monitoring (supervision) that checks if a Supervised Entity is executed sufficiently often and not too often.
Checkpoint	A point in the control flow of a supervised entity where the activity is reported to the WdgM.
Closed Graph	A closed graph is a directed graph where every checkpoint is reachable, starting from the local initial Checkpoint.
Configuration Tool	A tool used for creating a WdgM configuration, e.g., DaVinci Configurator Pro.
Container	Refers to the AUTOSAR term "container". Represents a structure with different parameters.
Deadline Supervision	Kind of WdgM monitoring (supervision) that checks if the execution time between two Checkpoints is lower or higher as the configured limits.
Destination Checkpoint	End point of a transition.
End Checkpoint	The last checkpoint that is monitored for a supervised entity. After passing the End Checkpoint, the WdgM expects that the entity is not monitored. To start the monitoring again the Initial checkpoint must be passed first. A supervised entity can have zero or more End Checkpoints.
Error	Discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition.
Failure	Termination of the ability of an element, to perform a function as required.
Fault	Abnormal condition that can cause an element or an item to fail.
Fault Detection Time	See. WdgM Fault Detection Time.
Fault Reaction Time	The Fault Reaction Time is the WdgM Fault Reaction Time plus the Wdg Fault Reaction Time.
Global Monitoring Status	Status that summarizes the Local Monitoring Status of all supervised entities.
Global Transition	A global transition is a transition between two checkpoints in the logical program flow (i.e. source and destination checkpoint), where the checkpoints belong to different supervised entities.
Initial Checkpoint	The first checkpoint that is monitored in the supervised entity. The monitoring of a supervised entity must start at this Checkpoint. A supervised entity has exactly one Initial Checkpoint.
Local Monitoring Status	Status that represents the current result of supervision of a single supervised entity.
Local Transition	A Local Transition is the transition between two checkpoints (i.e. source

	and destination checkpoint) in the logical program flow in the same supervised entity.
Program Flow Monitoring	Kind of WdgM monitoring (supervision) that checks if the inspected software is executed in a predefined sequence. This sequence is defined by the user and collected in the WdgM configuration.
WdgM Fault Detection Time	The time-span from the occurrence of a fault to the detection of the fault by the WdgM. The detection of a fault is indicated by a change of the state WDGM_LOCAL_STATE_OK or WDGM_GLOBAL_STATE_OK to a different state.
WdgM Tick (Counter)	Tick counter is used for deadline supervision time measurement. Depending on the parameter WdgMTimebaseSource the tick counter is incremented by 1 for each supervision cycle or, for higher precision, with the API function WdgM_UpdateTickCounter() or with a hardware counter.
Safe State	The Safe State is the operating mode of an item without an unreasonable level of risk [6], part1).
Watchdog Manager Stack	The software module consisting of Watchdog Manager, Watchdog Interface and Watchdog Driver.
Watchdog Manager (WdgM)	The hardware-independent upper software layer of the Watchdog Manager Stack.
Watchdog Interface (WdgIf)	The hardware-independent middle software layer of the Watchdog Manager Stack.
Watchdog Driver (Wdg)	The hardware-dependent lowest layer of the Watchdog Manager Stack. Controls the Watchdog device.
Source Checkpoint	Start point of a transition.
Supervised Entity	A software entity that is monitored by the WdgM. Each supervised entity has exactly one identifier. A supervised entity denotes a collection of checkpoints within a software component or basic software module. There may be zero, one or more supervised entities in a software component or basic software module. Each entity has a state that is based on the states reported from all its checkpoints. All checkpoints of one entity belong to the same memory context.
Supervision Cycle	The time period of the WdgM in which the cyclic supervision algorithm is performed.
Supervision Reference Cycle	The number of supervision cycles used as a reference by Alive, Deadline and Program Flow Supervision for periodic supervision. Every kind of supervision has its own reference cycle.
Timebase Tick	The WdgM measures the deadline of a Transition in timebase ticks (In the context of this document also referred to as WdgM tick).

	Note The timebase tick can be provided by the following sources: > WdgM itself (main function) > Component OS > External source
Trigger Mode	The WdgM Trigger Mode is a set of Watchdog trigger times and Watchdog mode. One Trigger Mode is a group of the following three parameters: > `WdgMTriggerWindowStart` > `WdgMTriggerConditionValue` > `WdgMWatchdogMode` Each Watchdog device can have one or more Trigger Modes.
Watchdog Device	The Watchdog Device is the hardware part which represents the watchdog functionality. It can be an internal watchdog integrated on the MCU chip, or it can be an external watchdog device outside the MCU.

Table 7-1   Glossary

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
BswM	Basic Software Module
CP	Checkpoint
CPID	Checkpoint Id
DEM	Diagnostic Event Manager
DET	Development Error Tracer
EAD	Embedded Architecture Designer
ECU	Electronic Control Unit
EDF	ECU Description File
HIS	Hersteller Initiative Software
ISO	International Organization for Standardization
MCU	Microcontroller Unit
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
QM	Quality Managed Software (software development process)
RTE	Runtime Environment
SCHM	Schedule Manager module (according AUTOSAR 4.0.1)
SE	Supervised entity
SEID	Supervised Entity Identifier
SW-C, SWC	Software Component
Wdg	Watchdog Driver
WdgIf	Watchdog Interface
WdgM	Watchdog Manager
SWS	Software Specification
Wdg	Watchdog

Table 7-2   Abbreviations

8 Contact

Visit our website for more information on

> News

> Products

> Demo software

> Support

> Training data

> Addresses

www.vector.com