MICROSAR OS

Technical Reference

Version 5.17

Authors	Dipl.-Ing. Winfried Janz, Dipl.-Inform. Joachim Stehle, Dipl.-Ing. Rainer Künnemeyer, Dipl.-Inform. Ulf Vatter, Dipl.-Ing. Torsten Schmidt, MSc (ETH) Inf.-Ing. Raphael Mack, M.Sc. Francisco Gonzalez
Status	Released
Document ID	OS01.0007

Document Information

History

Author	Date	Version	Remarks
To	2008-05-13	5.00	From AUTOSAR 2.1 to AUTOSAR 3.0
To	2008-06-20	5.01	Initial release
To	2008-10-16	5.02	New error code for GetResource API introduced: osdErrGRISRNoAccessRights. Chapter limitations added.
Se	2009-01-27	5.03	Updated description of error codes Added description of limited error checking for CounterTrigger Added description of NMI behavior for SC2-SC4
Rk	2009-03-18	5.04	Changed name of this document and changed a remark in chapter timing protection.
Rmk	2009-04-02	5.05	Replaced genTime example by DriverlessVehicle (Chapter 10.2)
Rk	2009-06-03	5.06	Added error code osdErrRRISRNoAccessRights
Rk	2009-07-02	5.07	Fixed ESOS00002435: Description of API service StartScheduleTableSynchron, further improvements of synchronization description
Bf	2010-02-01	5.08	Conversion to process document TechnicalReference_MICROSAR.doc Adding list of complete API
Vr	2010-02-11	5.09	Changes to documentation of feature Internal Trace
Jz	2010-02-19	5.10	Finalization of API list Edited open issues resulting from change of format (see v5.08)
Vr/RK	2010-03-15	5.11	Added feature CallISRHooks Added feature TimingMeasurement Added feature StackUsageMeasurement (Chapter 3.2.2.5) Updated error codes
Rmk	2010-04-20	5.12	Description of the restrictions of hook

			functions in systems with timing protection (Chapter 3.2.4.3) Added feature SingleStack (Chapter 3.2.2.6)
Rmk	2010-05-03	5.13	> Added description of conditional generation (Chapter 8.1.3) > Improved description of SingleStack (Chapter 3.2.2.6) > Updated description of CounterTrigger<CounterName> API (Chapter 3.2.1.4.1.3) > Added attribute PreAlarmHook (Chapter 7.3.1.5) > Added restrictions of COM in Scalability Classes 3 and 4.
Fgz	2010-10-14	5.14	> Corrected page numbering > Added `HiResSystemTimer` (Chapter 3.2.1.4) > Added High Resolution time macros (Chapter 3.2.5.1.6) > Updated to new High Resolution configuration (Chapter 3.2.5.2.1) > Deleted restrictions chapter of HRST (Chapter 3.2.5.2) > Deleted Error Numbers related to HRST restrictions (Chapters 3.3.3.6, 3.3.3.7 and 3.3.3.10) > Description of both installation styles (Chapters 4.2, 4.3, 4.5) > Added example for UseSpecialFunctionName (Chapter 7.3.7.1) > Added restriction to INITIALVALUE (Chapter 7.3.8.1) > Added description of file backup mechanism (Chapter 8.1.4)
Fgz	2010-11-08	5.15	> Added remark about example file (Chapter 4.5)
Fgz	2011-03-21	5.16	> Changed name of chapter 3.2.1.4.1.3 > Corrected description of `osdErrATAlarmMultipleActivation` (Chapter 3.3.3.1) > Added note for UseSpecialFunctionName (Chapter 7.3.7.1) > Deleted SyncSchT example (Chapter 10)
Rk/Fgz	2011-03-30	5.17	> Improved description of error code osdErrSETaskSuspended (Chapter 3.3.3.4).
Rk	2011-05-17		> Support for ORTI 2.2

			> Improved description of trusted function stub generation (Chapter 3.2.7.1) > Improved description of static alarms (Chapter 3.2.1.3) > Added description of cyclical expiry point actions (Chapter 3.2.5.3) > Improved description of stack monitoring (Chapter 3.2.2.3)

Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_OS.pdf AUTOSAR OS specification; This document is available in PDF-format on the internet a the AUTOSAR homepage (http://www.autosar.org)	3.1.0
[2]	AUTOSAR_BasicSoftwareModules.pdf	1.0.0
[3]	OSEK/VDX Operating System Specification This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	2.2.3
[4]	TechnicalReference_MicrosarOS_xxxx.pdf Technical reference of Vector Microsar OS; Hardware specific part	--
[5]	OSEK/VDX Communication This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	3.02
[6]	OIL: OSEK Implementation Language This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)	2.32
[7]	Tutorial_osCAN.pdf Tutorial for the Microsar OS OSEK/AUTOSAR Realtime Operating System	1.00
[8]	autosar.xsd AUTOSAR XML schema	3.0.1

Scope of the Document

MICROSAR OS is an operating system, compliant with the AUTOSAR OS and OSEK standards. The general aspects of all Scalability Classes, SC1 – SC4, are described in this document. For each implementation, the hardware specific part is described in a separate document [4].

The implementation is based on the AUTOSAR OS specification [1].

It is also based on the OSEK OS specification 2.2 described in the document [3].

This documentation assumes that the reader is familiar with both the OSEK OS specification and the AUTOSAR OS specification.

This documentation describes only the operating system and the code generation tool.

OSEK is a registered trademark of Continental Automotive GmbH (until 2007: Siemens AG).

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

Contents

Illustrations

Tables

1 Component History

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

This chapter is included in every Microsar component documentation For the OS the history on the intended level is not relevant as Microsar OS implements all mandatory requirements of Autosar OS.

2 Introduction

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

Supported AUTOSAR Release:	3.1.0	
Supported Configuration Variants:	pre-compile	
Vendor ID:	OS_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	OS_MODULE_ID	1 decimal (according to ref. [2])

2.1 Architecture Overview

The following figure shows where the OS is located in the AUTOSAR architecture.

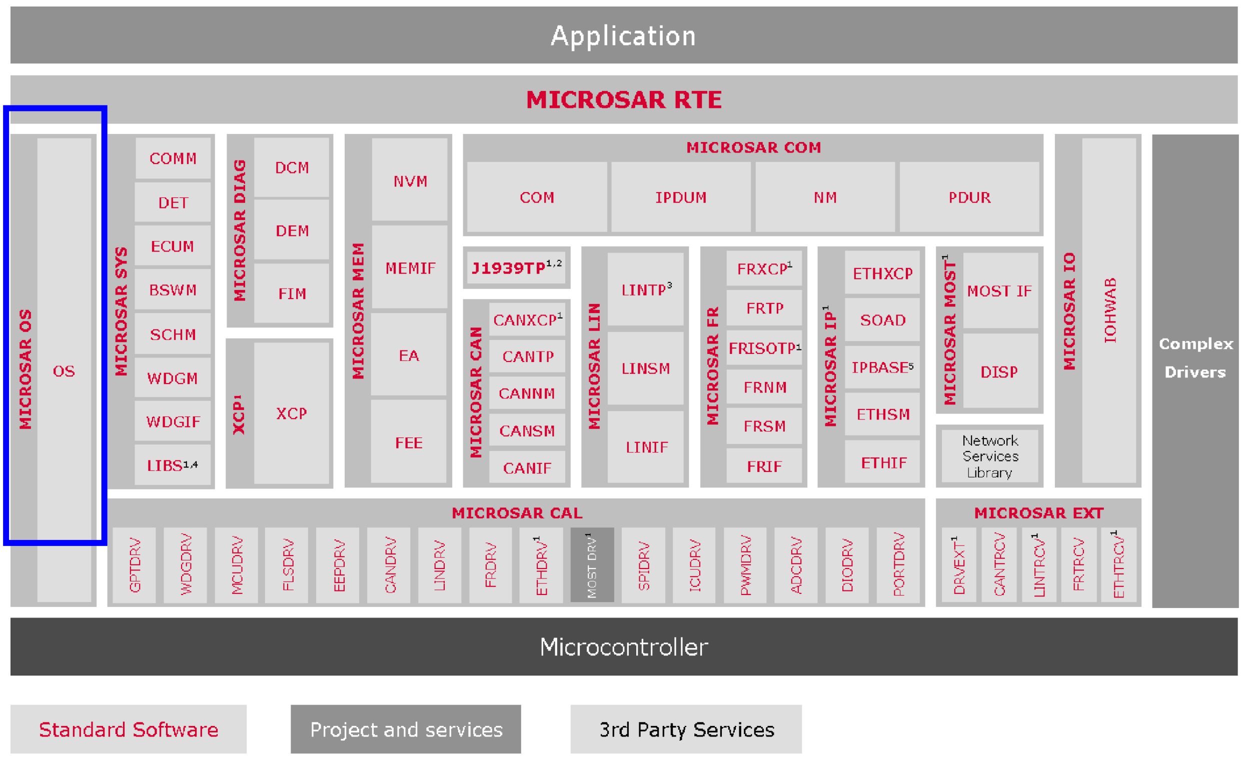

Figure 2-1 AUTOSAR architecture

The MICROSAR OS is an operating system based on the AUTOSAR OS standard 3.0.0 (ref. [1]) and on OSEK OS standard 2.2.3 (ref. [3]).

This MICROSAR OS operating system is a real time operating system which was specified for the usage in electronic control units on a range of small to large microprocessors. MICROSAR OS has attributes which differ from commonly known operating systems and which allow a very efficient implementation even on systems with low resources of RAM and ROM.

As a requirement, there is no dynamic creation of new tasks at runtime; all tasks have to be defined before compilation. The operating system has no dynamic memory management and there is no shell for the control of tasks by hand.

The operating system and the application are compiled and linked together to one file which is loaded into an emulator or is burned into an EPROM or Flash EEPROM.

3 Functional Description

3.1 Features

The features listed in this chapter cover the complete functionality specified in [1].

The "supported" and "not supported" features are presented in the following two tables. For further information of not supported features also see chapter 9.

The following features described in [1] are supported:

Supported Feature
The Vector MICROSAR OS implementation implements all mandatory features specified in [1].

Table 3-1 Supported SWS features

The following features described in [1] are not supported:

Not Supported Feature
--

Table 3-2 Not supported SWS features

The OSEK / AUTOSAR OS specifications leave many points open on implementation. Every OSEK / AUTOSAR OS implementation for a specific microcontroller has to define the open points to achieve an optimal solution for the processor. The operating system has to fit the target microprocessor and the C compiler. The programming model of the C-compiler is as important as the hardware of the processor.

3.2 Main Functions

The operating system is started by the application. The startup module (which is not part of the operating system) calls the function main. In the main function the user has to call the API function `StartOS`. `StartOS` will initialize the operating system, install the interrupt-routine for the alarm-handling, and then call the scheduler. `StartOS` will never return to the main function.

The function of the scheduler is to evaluate the task with the highest priority in the `READY` state and call this task. If the task was previously pre-empted by another higher priority task, the scheduler resumes the task.

The operating system is controlled by external events. External events can be events from interrupt routines, from the alarm management, or from schedule tables (Alarms and schedule tables are also driven by interrupt service routines). Therefore any external event will result in the change of task states.

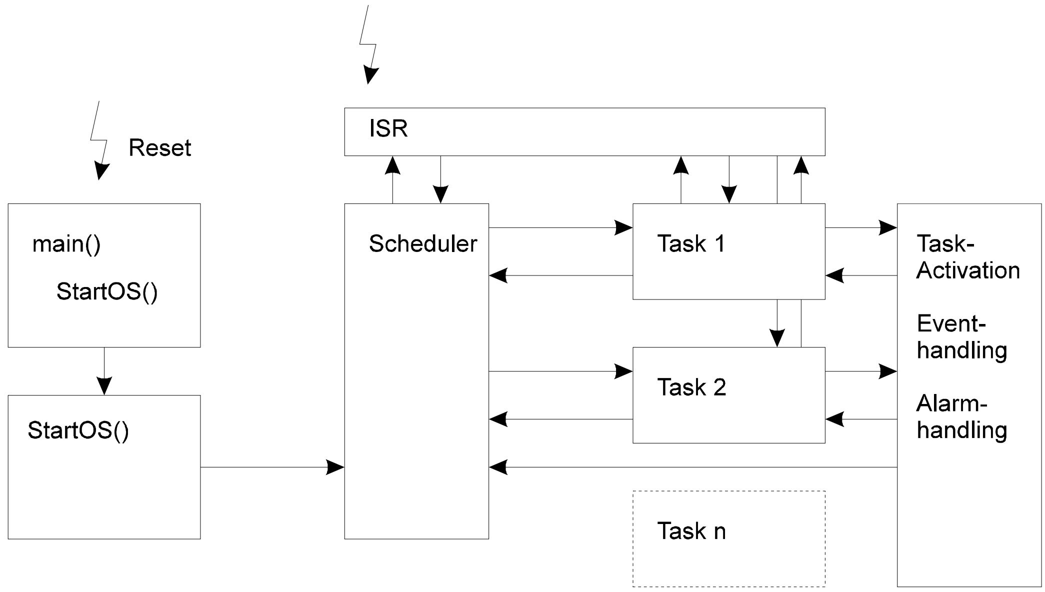

Figure 3-1 Functional parts

Interrupt routines are under the control of the application programmer. An OSEK operating system allows a fast and efficient interrupt handling, so interrupts have a short latency time. It is possible to call certain system functions from interrupt routines. It is necessary that the operating system has knowledge of any existing interrupt routines.

3.2.1 Timer and Alarms

All time based actions are performed in OSEK using counters, alarms, and schedule tables. Counters are part of the kernel and are incremented by a specific hardware resource or by means of the system service `IncrementCounter`. In case of a time based counter, the counter is incremented periodically. Alarms and schedule tables have fixed references to counters.

An alarm, if activated, has a certain value. If the referenced counter reaches the given value a defined action is performed. The action to each alarm is defined by the OIL Configurator and is compiled into the ROM. The alarm value is passed as a parameter to the functions `SetRelAlarm` or `SetAbsAlarm`.

A schedule table, if activated, has a certain starting value. If the referred counter reaches the given value, the first defined action is performed. The further actions are defined relative to this first action. These relative starting times are defined together with the action by the OIL Configurator and compiled into ROM. The starting value is passed as a parameter to the functions `StartScheduleTableRel` or `StartScheduleTableAbs`.

3.2.1.1 Time Base

Usually the time base of the operating system is configurable. The tick duration can be set in the OIL Configurator with the attribute `TickTime` in the OS object. The time base unit is µs. All time values in MICROSAR OS, which are using the time conversion macros nsec, usec, msec or sec, are based on the attribute `TickTime`. For this reason, they should only be used for alarms, using `SystemTimer` for their time basis or with counters, which are ticked with the same frequency as the `SystemTimer`.

In some implementation it is also possible to select the hardware timer which is used as system timer by means of the OIL attribute `SystemTimer`. Details about this and the alarm handling are described in the hardware specific manual [4].

3.2.1.1.1 User defined SystemTimer

In most implementations it is possible to configure `UserDefined` for the `SystemTimer`. If `UserDefined` is selected the counter API (as described in chapter 3.2.1.4.1) will be generated for the system timer and the system timer can then be triggered by any cyclic interrupt.

Example

Timer7 should be used for SystemTimer but it is necessary to use Timer7 also for the application.

OIL Configurator:

> Select `UserDefined` as SystemTimer
> Insert an ISR of category 2 for Timer7
> Select `UseGeneratedFastAlarm`
> Select the `TickTime` (e.g.1000)

The following functions are generated by the Code generator:

> `TickType GetCounterValueSystemTimer(void);`
> `StatusType InitCounterSystemTimer(TickType ticks);`

Insert the following code in the application:

```
void StartupHook(void)
{
   /* my initialization code in the StartupHook */
   ...
   /* initialize Timer7 */
   MyTimer7Init();
   /*call generated function to reset the counter of the SystemTimer*/
   InitCounterSystemTimer(0);
}
ISR(MyTimer7ISR)
{
   /* ISR is called every 500 microseconds by Timer7 */
   static uint8 n=0;
   n++;
   /* my application code in the ISR */
   ...
```

```
    /* TickTime is 1000 micro seconds, but ISR is called every 500 */
    if (n==2)
    {
       /* call generated function every 1000 micro seconds */
       IncrementCounter(SystemTimer);
       n=0;
    }
}
```

Caution
The configured `TickTime` in the OIL Configurator must be the same as the cyclic call time of the generated function Increment`Counter(SystemTimer).`

3.2.1.1.2 Timer Macros

To support the portability of OSEK, application alarm related functions, for example `SetRelAlarm`, should be called using macros for the calculation of ticks based on millisecond or seconds:

```
SetRelAlarm (Alarm1, USEC(1200), USEC(1200));  /* microseconds */
SetRelAlarm (Alarm1, MSEC(10), MSEC(10));      /* milliseconds */
SetRelAlarm (Alarm2, SEC(12), SEC(12));        /* seconds      */
```

The macros `USEC`, `MSEC` and `SEC` calculate the number of ticks necessary to get the assigned time. These macros are generated by the OIL code generator.

3.2.1.1.3 Range of Alarms

The range of alarms depends on the tick time which is configurable. It also depends on OSEK constants `OSMAXALLOWEDVALUE`. This constant depends on the OSEK data type `TickType` and the alarm management.

The hardware specific details are described in ref. [4].

3.2.1.2 Timer Interrupt Routine

The timer interrupt routine is a category 2 ISR and is part of the operating system. The configuration is done automatically by the OS using information which has to be defined in the OIL Configurator (e.g. `CpuFrequency`, `TickTime`).

3.2.1.3 Static Alarms

The MICROSAR OS operating system supports static alarms. The usage of static alarms saves RAM, because the static `CycleTime` is stored in the ROM. It is possible to start the static alarm automatically on system start if the attribute `StaticAlarm` is selected in the OIL Configurator (sub-attribute of `AUTOSTART`). If the static alarm is activated in the application, the symbols `OS_STATIC_ALARM_TIME` and `OS_STATIC_CYCLE_TIME` should be used.

To define an alarm as static in a XML configuration can be done in two ways:

1. The `OsAlarmStaticAlarm` inside the `OsAlarmAutostart` container is set to `TRUE`. The alarm will be a static alarm which has autostart. Alarm time and cycle time are configured in `OsAlarmAlarmTime` and `OsAlarmCycleTime`.
2. If no container `OsAlarmAutostart` exists (alarm has no autostart) the alarm can be made static by using the container `OsAlarmStaticAlarm`. Alarm time and cycle time are configured in `OsAlarmAlarmTime` and `OsAlarmCycleTime`.

   Please note that the container `OsAlarmStaticAlarm` will be ignored if the container `OsAlarmAutostart` exists.

Example

```
SetRelAlarm(Alarm1,OS_STATIC_ALARM_TIME,OS_STATIC_CYCLE_TIME);
```

The `AlarmTime`, the `CycleTime` and the `AlarmUnit` have to be defined in the OIL Configurator.

Caution

The implementation of static alarms is an extended feature of the MICROSAR OS implementation. The usage saves RAM, but the application code might not be portable between different AUTOSAR OS or OSEK implementations.

Caution

If only static alarms are used in an application **and** the option `USEPARAMETERACCESS` in the OIL Configurator is checked (for enhanced error management), the access macros `OSError_SetRelAlarm_increment()`, `OSError_SetRelAlarm_cycle()`, `OSError_SetAbsAlarm_start()` and `OSError_SetAbsAlarm_cycle()` return an undefined value.

3.2.1.4 Additional Counters

Per default MICROSAR OS supports one counter – the `SystemTimer` (and optionally also the `HiResSystemTimer`). If additional counters are required, e.g. for angle management, they can be added in the OIL Configurator. Note that multiple counters are supported in conjunction with the OIL attribute `UseGeneratedFastAlarms` only.

While the `SystemTimer` is controlled by the kernel, other counters have to be initialized and triggered by the user by means of an API. The required API functions are generated automatically.

Note that the timer macros `USEC()`, `MSEC()` and `SEC()` are based on the `SystemTimer` and work correctly if used with alarms assigned to the `SystemTimer`; the timer macros `HRUSEC()`, `HRMSEC()` and `HRSEC()` are based on the `HiResSystemTimer` and work correctly if used with alarms assigned to the `HiResSystemTimer`. This also applies to static alarms. Please also refer to chapter 3.2.5.1.6 for a description of the macros `OS_TICKS2xx_<CounterName>()` and `OS_xx2TICKS_<CounterName>()` .

Microsar OS supports up to 256 counters are supported.

3.2.1.4.1 Counter API

To obtain the counter specific limits (e.g. `maxallowedvalue`) the function `GetAlarmBase` can be used.

3.2.1.4.1.1 Initialization

```
StatusType InitCounter<CounterName>(TickType ticks);
```

Return value: `E_OS_VALUE` if ticks are greater than the maximum allowed.

Called by the user to set the counter to a specific value.

Caution
It is not allowed to call the `InitCounter` functions with interrupts disabled by `DisableAllInterrupts`, `SuspendAllInterrupts` or `SuspendOSInterrupts`. Violations will not be detected and might cause wrong system behavior.

It is not allowed to call the `InitCounter` functions if alarms are active for that counter. Violations are not detected by the OS.

In Scalability Classes 3 and 4 this function may only be called from trusted applications or system hooks. It is suggested to use the `InitCounter` functions only in the system wide `StartupHook`.

3.2.1.4.1.2 Read Counter

```
TickType GetCounterValue<CounterName>(void);
```

Return value: current counter value.

Called by the user to read the current counter value.

Caution
It is not allowed to call the `GetCounterValue` functions with interrupts disabled by `DisableAllInterrupts`, `SuspendAllInterrupts` or `SuspendOSInterrupts`. Violations will not be detected and might cause wrong system behavior.

3.2.1.4.1.3 Increment Counter

```
StatusType IncrementCounter(CounterType CounterName);
```

This function is the AUTOSAR OS standardized function to trigger a counter. This function does more error checking, but needs more runtime to figure out which counter shall be triggered. AUTOSAR OS does not specify functions to initialize and read a counter, so the counter always starts counting at zero and cannot be read if only AUTOSAR OS compliant functions shall be used.

Example

The ISR which triggers the counter must be of category 2.

```
ISR(MyCounterISR)
{
    IncrementCounter(MyCounter);
}

void StartupHook(void)
{
    InitCounterMyCounter(180);
}
```

Former versions of osCAN and MicrosarOS provided the API function

```
void CounterTrigger<CounterName>(void);
```

This function is obsolete and mapped to a call of the AUTOSAR API function IncrementCounter. In new systems, the API functions IncrementCounter should be used in favor of CounterTrigger<CounterName>.

3.2.2 Stack Handling

3.2.2.1 Task Stack

Each task has its own stack. The task stack holds all local data and return addresses of the task. In addition, the register context of the task is saved onto the stack if the task is preempted. If the task is transferred to the running state again the register context is removed from the task stack to restore the previously saved registers.

3.2.2.2 Interrupt Stack

The implementation of interrupt stacks depends on the hardware and is described in ref. [4].

3.2.2.3 Stack Monitoring

To use this feature, set `STACKMONITORING = TRUE` in the configuration.

If enabled, the system initializes the last useable element of each stack with an indicator value.

MICROSAR OS checks this last usable element with each task switch or ISR exit. A change of the element value indicates a stack overflow by the task or ISR. In this case, the system calls the `ErrorHook` (if configured), the `ShutdownHook` (if configured) and enters the shutdown state.

Info
It is highly recommended to use this feature during development

Note
MICROSAR OS checks the indicator value only at task switches and on a return from an ISR. Therefore, stack overflows are not detected immediately. Detection might be delayed arbitrary in case of a stack overflow in a hook routine. Some implementations of MICROSAR OS implement additional checks of the indicator value, see [4]. If memory protection is configured, stack overflows in tasks and ISRs of non trusted applications are found immediately by the memory protection.

3.2.2.4 Stack Sharing

MICROSAR OS makes the assumption that a task always starts on an empty stack. Afterwards, the stack stays valid until the task terminates. Even when the task is preempted for a long time, the stack must not be altered. With this assumption a group of different basic tasks can use the same memory region for their stack region, when only one of them can start and the rest cannot.

The following subchapters present the configuration settings detected by MICROSAR OS to enable stack sharing for a group of tasks. Please note that stack sharing is never possible for extended tasks, as the major feature of this type of task is the usage of the API function `WaitEvent`. While the extended task waits for an event to occur, any other task might become active, so the basic assumption presented above is not valid.

In the case a group of tasks shares a memory region for their stacks, the user can still set the TASK attribute `StackSize` for each of the tasks. The biggest setting for `StackSize` in the group of tasks determines the size of the memory region that is reserved by MICROSAR OS.

Info
In Scalability Class 3 and 4 these stack sharing mechanisms are only used for tasks which are assigned to the same Application.

3.2.2.4.1 Basic Tasks on the same Priority

Basic tasks cannot start as long as another basic task on the same priority level has started and not yet finished. Therefore these tasks can share one memory region for their stacks. This is automatically detected by MICROSAR OS, and whenever two or more basic tasks share the same priority level, they use the same memory region for their stack

3.2.2.4.2 Basic non-preemptive Tasks not calling SCHEDULE

Basic non-preemptive tasks can only be preempted if they call the system service `Schedule`. If they do not call this system service, they can share the same memory region for their stacks. Unfortunately, it is not possible for MICROSAR OS to detect automatically before compilation if `Schedule` is called or not. For this reason, the task object in the OIL Configurator provides the attribute `NotUsingSchedule`. By selecting this attribute (setting it to `TRUE`), the user guarantees that the task does not call `Schedule`. So a group of non-preemptive basic tasks that all have the attribute `NotUsingSchedule` set to `TRUE` share the same memory region for their stacks.

When the OS attribute `STATUS` is set to `EXTENDED` and the OS attribute `OSInternalChecks` is selected (set to `TRUE`), MICROSAR OS checks with each call of the system service `Schedule` if the attribute `NotUsingSchedule` was selected for the calling task. In the case that `NotUsingSchedule` was selected but the respective task still calls `Schedule`, an error message is produced.

3.2.2.4.3 Basic Tasks, sharing an internal Resource and not calling SCHEDULE

Basic tasks using an internal resource can only be preempted by other tasks sharing the same resource if they call the system service `Schedule`. So if they do not call this system service, they can share the same memory region for their stacks. Unfortunately, it is not possible for MICROSAR OS to detect automatically before compilation if `Schedule` is called or not. For this reason, the task object in the OIL Configurator provides the attribute `NotUsingSchedule`. By selecting this attribute (setting it to `TRUE`), the user guarantees that the task does not call `Schedule`. So a group of basic tasks that share the same internal resource and all have the attribute `NotUsingSchedule` set to `TRUE` share the same memory region for their stacks.

When the OS attribute `STATUS` is set to `EXTENDED` and the OS attribute `OSInternalChecks` is selected (set to `TRUE`), MICROSAR OS checks with each call of the system service `Schedule` if the attribute `NotUsingSchedule` was selected for the calling task. In the case that `NotUsingSchedule` was selected but the respective task still calls `Schedule`, an error message is produced.

3.2.2.5 Stack Usage

If StackUsageMeasurement is set to `TRUE`, the OS fills all available stacks with the indicator value 0xAA during StartOS (startup times will be slower). This allows measuring the amount of stack used since StartOS by counting the amount of bytes that have not been overwritten yet.

The following function is available to determine the amount of used stack:

```
osuint16 osGetStackUsage(TaskType taskId)
```

- Argument: Task number
- Return value: Maximum stack usage (bytes) by task since call of `StartOS()`

Additional implementations specific functions may be available. Please see the hardware specific part of the documentation of this implementation [4].

3.2.2.6 Single Stack Models

MICROSAR OS provides the possibility to execute all basic Tasks on one single stack if Scalability Class 1 is selected. Whether ISRs are executed on this single stack is up to the implementation.

Cross reference
Refer to the hardware specific manual for details about the stack of ISRs and a list of the stack models which are actually supported. If the hardware specific manual does not mention single stack at all, the only supported stack model is STANDARD.

3.2.2.6.1 SingleStackOsek

Full support of the OSEK/AUTOSAR specification, without any limitation. Extended Tasks still use dedicated stacks.

3.2.2.6.2 SingleStackOptimized

Support only Conformance classes BCC1 and BCC2 with the following restrictions. Functions `TerminateTask` and `ChainTask` may only be called from the body of a TASK function (implemented using the TASK macro). `TerminateTask` and `ChainTask` do not return a status code i. e. the signature of this API functions differs from the OSEK standard and are `void TerminateTask(void)` and `void ChainTask(TaskType <TaskID>)`.

The stack measurement API for Tasks `osGetStackUsage` is not available in this stack model. The implementation specific function to measure the stack usage of the system stack may be used.

3.2.2.6.3 SingleStackOptimizedCS

As SingleStackOptimized, but no stack is created by the OS at all. The OS uses the stack which was setup at the time of calling `StartOS`. This has the additional limitation that StackMonitoring and StackMeasurement is not supported

3.2.2.6.4 STANDARD

Tasks and ISRs use dedicated stacks (stack sharing is considered as described in Chapter 3.2.2.4).

3.2.3 Interrupt Handling

Implementation specific details about interrupt handling are described in the hardware specific part of this implementation [4].

Caution
Knowledge about the interrupt handling is very important. If interrupt routines are used it is essential to read this chapter.

3.2.3.1 Interrupt Categories

The OSEK OS specification defines two groups of interrupts.

3.2.3.1.1 Category 1:

Interrupts of category 1 are in general not allowed to use API functions; as such, these routines can be programmed without restrictions and are totally independent from the kernel. The programming conventions depend on the utilized compiler and assembler.

Category 1 interrupts can be enabled before call of `StartOS()`. If interrupts of category 1 and 2 cannot be disabled separately, all interrupts must be disabled.

Interrupts of category 1 are allowed to call the interrupt API as an exception to the rule presented above. If the interrupt API is used and the category 1 interrupts are enabled before the call of `StartOS`, the user has to take care about variable initialization of the interrupt API, as described in chapter 3.2.3.2.

3.2.3.1.1.1 Exceptions and SC2, SC3, SC4

According to the AUTOSAR-Standard, category 1 interrupts should be avoided with SC2, SC3 and SC4. MICROSAR OS does not allow category 1 interrupts with `TimingProtection`. Because non maskable interrupts need to be configured to category 1, some MICROSAR OS implementations allow exceptions even with timing protection.

The user may write "normal" interrupt code in an exception routine which returns to the application. Please note that this sort of exception routines will cause the exception handler to add runtime to the account of the interrupted task or ISR.

3.2.3.1.2 Category 2:

Interrupts of category 2 may use certain restricted API functions. Interrupts of category 2 can be programmed as normal C functions using the macro `ISR(name)`. The C function is called by the operating system. The necessary preparation for the interrupt routine is done automatically by a generated function.

```
ISR (AnInterruptRoutine)
{

   /* code with API calls */

}
```

Caution

Category 2 interrupts must be disabled until call of `StartOS()`! This also applies for the system timer interrupt, i.e. this interrupt must be stopped by the user at a software reset.

To ensure data consistency, the operating system needs to disable category 2 interrupts during critical sections of code. Therefore applications must not use non-maskable interrupts as category 2 interrupts.

3.2.3.2 Usage of the Interrupt API before StartOS

The usage of the interrupt API functions is in general allowed before the operating system is started. The affected functions are:

> `DisableAllInterupts`, `EnableAllInterrupts`

> `SuspendAllInterrupts`, `ResumeAllInterrupts`

> `SuspendOSInterrupts`, `ResumeOSInterrupts`

However, these functions use some internal variables that have to be initialized to zero before the first call of the interrupt API. Typically this initialization is performed by the startup code (which might be delivered with the compiler). In case no startup code is used, the function `osInitialize()` needs to be called. `osInitialize` initializes the variables which are used in the interrupt API.

3.2.4 Timing Protection

The timing protection is implemented in the Scalability Classes 2 and 4. This chapter provides some hints on the functionality according to the Autosar OS standard [1] and describes additional functionality provided by Vector MICROSAR OS. To enable the timing protection of a task or ISR, the OIL-attribute `TIMING_PROTECTION` of the respective task or ISR needs to be configured, as described in chapters 7.3.2.2 and 7.3.7.2 . (Autosar XML: OsTaskTimingProtection/OsIsrTimingProtection).

Caution
The runtime of all tasks and ISRs is observed, however parts of the time for task switch and interrupt entry/exit cannot be monitored by the timing protection. Therefore, some extra time for task switches and interrupt entry/exit needs to be considered in the configuration of the timing protection.

Caution
Timing Protection is implemented with interrupts. If the application manually disables interrupts anywhere, the timing protection cannot work as expected. In order to enable and disable interrupts, the application **must** use the following API functions:

> `DisableAllInterupts, EnableAllInterrupts`

> `SuspendAllInterrupts, ResumeAllInterrupts`

> `SuspendOSInterrupts, ResumeOSInterrupts`

Caution
The timing protection works very precise. However, if an OS API function is called by a task/ISR, the OS may enter a critical section; if a timing protection violation is detected while the system is in a critical section, the call of the protection hook may be delayed until the end of the critical section. Note, that critical sections in Autosar OS are very short.

Note
API functions SendMessage and ReceiveMessage disable interrupts during the access to the message buffers. The detection of a timing violation is delayed until the copying of the message buffer finished.

3.2.4.1 Reaction on Protection Failure

To provide a more efficient version of the OS, it is possible to select in OIL which reactions to the ProtectionHook are possible.

If only Shutdown is selected, SC2 runs with the best performance.

3.2.4.2 Timing Measurement

MICROSAR OS is not only able to provide timing protection but allows using the same functionality for timing measurement. If timing measurement is performed for a specific task or ISR, the OS measures the following times for that task or ISR:

> the maximum run time since StartOS

> the maximum locking times for resources and interrupts since StartOS

> the minimum time distance between two arrivals since StartOS

The debugger can read the result of the timing measurement via ORTI. Alternatively, the application may use the timing measurement API as described in chapter 6.4.

The OS attribute `TimingMeasurement` and the task/ISR attribute `TIMING_PROTECTION` are provided to setup timing protection and measurement.

The hardware timers and internal data structures to store measured times are limited in size. When this limit is exceeded by the measured time of a resource- or interruptlocktime, the ErrorHook function is called and the system goes into shutdown state.

3.2.4.2.1 Timing measurement configuration for a specific task/ISR

Timing measurement can be configured individually for each task and ISR. As timing protection requires the OS to measure the timing values, timing measurement is performed for all tasks and ISRs that have timing protection configured by means of the attribute `TIMING_PROTECTION`. By selecting the subattribute OnlyMeasure, the OS disables the timing protection but still measures the timing values. Please note that this configuration might be overridden by means of the global configuration, described in the next chapter.

3.2.4.2.2 Global configuration of timing measurement

In order to save configuration time, the timing measurement can be configured globally for all tasks and ISRs. MICROSAR OS provides the OIL-attribute `TimingMeasurement` (Autosar XML: `OsOSTimingMeasurement`) for that purpose. That attribute provides the possibilities to:

> Disable the timing measurement globally. This is an optimization to save memory and runtime of the timing measurement. Please set the attribute `TimingMeasurement` to `FALSE` (deselect it) for this configuration.

> Collect timing data for all tasks and ISRS. The collected timing values can be used to perform schedulability analysis and to set up the timing protection later on. For this configuration, please set the attribute `TimingMeasurement` to `TRUE` (select it) and choose `OnlyMeasureAll` for the value of the subattribute GlobalConfig.

> Perform timing measurement as configured for the task or ISR. Set the attribute `TimingMeasurement` to `TRUE` (select it) and select `AsSelected` for the value of the subattribute GlobalConfig to achieve this.

> Ignore the task/ISR attribute OnlyMeasure (perform timing measurement and protection as if this attribute was set to `FALSE`). For this configuration, please set the attribute `TimingMeasurement` to `TRUE` (select the attribute) and select `ProtectAndMeasureAll` for the value of the subattribute GlobalConfig.

The chapter 7.3.2.2 provides a description of the attribute `TIMING_PROTECTION` for tasks while chapter 7.3.7.2 provides the respective description for ISRs. Chapter 7.3.1.4 provides a description of the attribute `TimingMeasurement`. The table below documents the interdependence between the OS-attribute TimingMasurement and the task/ISR-Attribute `TIMING_PROTECTION`.

OS		TASK/ISR		
		TIMING_PROTECTION		
Timing-Measurement	**Global-Config**	**FALSE**	**TRUE**	
			OnlyMeasure	
			FALSE	**TRUE**
FALSE[1]		No timing protection, no timing measurement,	Timing protection but no timing measurement	Timing protection but no timing measurement, warning as the subattribute OnyMeasure is overridden
TRUE	**Only-Measure-All**	No timing protection but timing measurement	No timing protection but timing measurement, warning as the subattribute OnlyMeasure is overridden	No timing protection but timing measurement
	AsSelected	No timing protection and no timing measurement	Timing protection and measurement	No Timing protection but timing measurement
	ProtectAnd-MeasureAll	No timing protection but timing measurement	Timing protection and measurement	Timing protection and measurement, warning as the subattribute OnlyMeasure is overridden.

Table 3-3 Interdependence between the OS attribute `TimingMeasurement` and the task/ISR attribute `TIMING_PROTECTION`

3.2.4.3 Hook functions

The runtime of the hook functions PreTaskHook and PostTaskHook (Chapters 6.6.2 and 6.6.3) is considered to belong to the runtime of the currently active task.

[1] Optimization: The timing measurement API is unavailable

Caution
Depending on the implementation the execution of the ProtectionHook might be delayed until the PreTaskHook or PostTaskHook has finished. In case of a protection violation during the PostTaskHook while a task state change into the states SUSPENDED or WAITING, the call of the ProtectionHook gets lost.
Therefore, the PreTaskHook and the PostTaskHook should be used for debugging purposes only. If these functions are used in safety critical applications, extra care is necessary.

The runtime of the hook functions UserPreIsrHook and UserPostIsrHook (Chapters 6.6.7 and 6.6.8) is considered to belong to the runtime of the currently active ISR.

3.2.5 Schedule Tables

MICROSAR OS implements schedule tables as defined by the AUTOSAR standard V3.0. The document [1] provides the base description of schedule tables and their usage. This chapter is meant as an extension that clarifies and corrects some points, provides details about points left open and describes corrections and extensions by MICROSAR OS.

AUTOSAR defines schedule tables for all scalability classes, but synchronization to a global time only for SC2 and SC4. MICROSAR OS offers some additional error checking to the Autosar standard.

3.2.5.1 Synchronization

In SC2 and SC4 or the High-Resolution Schedule Tables option, it is possible to synchronize a schedule table to a global time source. The schedule table must be marked with the OIL attribute `LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION`.

3.2.5.1.1 Starting a synchronizable Schedule Table

One way to start a synchronizable schedule table is to use the functions `StartScheduleTableRel(Tablename,TimeOffset)` and `StartScheduleTableAbs(Tablename, Time)`. The time parameters refer to the local time and not to the global time. It is assumed that the schedule table will not have an offset to global time: when synchronization starts, the start of the schedule table is moved to global time zero. Note that the schedule table always starts at the stated start time. A call of `SyncScheduleTable` does not influence this start time. Synchronization starts after execution of the first expiry point.

The recommended way to start a synchronizable schedule table is to use the combination of `StartScheduleTableSynchron(Tablename)` and `SyncScheduleTable()`. The first expiry point is executed at global time 0. The schedule table starts execution after a global time is available, i.e. after calling `SyncScheduleTable()` for the schedule table. If `SyncScheduleTable()` is never called for the schedule table, the schedule table is never executed. The following algorithm describes a possibility to set up a timeout:

Set up an alarm to the timeout time. When the alarm expires and `GetScheduleTableStatus()` indicates that the schedule table is still waiting, call `SyncScheduleTable()` with an arbitrary time. Note: if the call to

`SyncScheduleTable()` is done in an interrupt, it may occur between the two API calls, and thus gets overridden by the arbitrary time..

3.2.5.1.2 Autostart

For an automatic start of a schedule table on startup of the OS, the attribute AUSOSTART must be set. The sub-attribute `TYPE` defines how the start is performed. The possibilities are: `ABSOLUT`, `RELATIVE` and `SYNCHRON`. These types of autostart are similar to the normal start of a schedule table using `StartScheduleTableAbs`, `StartScheduleTableRel` or `StartScheduleTableSynchron`. For `ABSOLUT` and `RELATIVE`, an absolute or relative start time needs to be provided. In case of `SYNCHRON`, the schedule table starts after `SynchScheduleTable` has been called and the global time reaches zero.

Note, that just like for `StartScheduleTableSynchron()`, the schedule table will wait forever if `SyncScheduleTable()` is never called.

3.2.5.1.3 Suspending a Schedule Table and keeping its Synchronization

The AUTOSAR standard does not define a way to suspend the execution of a schedule table and keep its synchronization for later restart. The suggested approach is to use `NextScheduleTable()` to append a schedule table that effectively does nothing.

Currently, AUTOSAR does not define a way to retrieve the internal schedule table time (neither the currently estimated global time nor the time relative to the first expiry point of the schedule table).

3.2.5.1.4 Providing a Global Time

The current global time is handed to the schedule table via `SyncScheduleTable()`. If a deviation of the schedule table to the global time is found, the schedule table starts to synchronize at the next expiry point that allows synchronization.

The global time must be a continuous range of integers: So e.g. FlexRay's time tuple (cycle, macroticks) cannot directly be used as global time, but must be converted.

The provided global time must have the same resolution as the local time and the same period as the schedule table time (i.e. the `LENGTH` of the schedule table). If this is not the case, it must be converted before being handed to `SyncScheduleTable()`.

Example

Converting the FlexRay time:

Be `gMacroPerCycle` the number of `macroticks` per cycle, `cycle` the current cycle number, `macroticks` the current macrotick number and `f` is the factor to convert the FlexRay tick length the HW counter tick length:

```
GlobalTime = (gMacroPerCycle * cycle + macroticks) * f
```

3.2.5.1.5 Exact Synchronization

There is always a time span between reading the global time and handing it to the operating system. Therefore, synchronization is never absolutely exact.

If an interrupt interferes, the time span may be unexpectedly large. While this may be ignorable if the resolution is large compared with the interrupt running times, it is noticeable when using a fine grained global time, for example in conjunction with High-Resolution Schedule Tables, or if some (higher prior) interrupts have long running times. It is desirable to be undisturbed by (higher prior) interrupts during synchronization.

The AUTOSAR Standard V3.0 demands, that calling `SyncScheduleTable()` is not allowed in between function pairs

`DisableAllInterrupts/EnableAllInterrupts,`
`SuspendAllInterrupts/ResumeAllInterrupts,`
`SuspendOSInterrupts/ResumeOSInterrupts.`

MICROSAR OS offers a way to diverge from the standard in this point: if the macro `osdSyncScheduleTableAllowsDisabledInterrupts` is defined at compile time, `SyncScheduleTable()` can be called in between these function pairs.

In this case, a Sync procedure may look like the following example:

```
DisableAllInterrupts();
now=getCurrentGlobalTime();
SyncScheduleTable(MyScheduleTable, now);
EnableAllInterrupts();
```

Caution

This is unnecessary if `SyncScheduleTable()` is called from an interrupt which does not allow nesting.

Also note, that `SuspendAllInterrupts()/DisableAllInterrupts()` still allow timing protection interrupts (if timing protection is used).

3.2.5.1.6 Calculating with Times

Please notice that all AUTOSAR OS API functions expect time parameters in ticks!

The macros `OS_TICKS2xx_<CounterName>()` (whereas `xx` denotes `NS`, `US`, `MS` or `SEC`; described by the AUTOSAR standard) may be used to convert tick values (as returned for example by `GetElapsedTime()` and `GetCounterValue()`) to real time. Additionally, MICROSAR OS added the macros `OS_xx2TICKS_<CounterName>()` to convert in the opposite direction.

Time constants (defined in the `COUNTER` section of the OIL file) help convert from real time to ticks. The macros `USEC()`, `MSEC()`, and `SEC()` are for `SystemTimer` based times only, they cannot be used for High-Resolution Schedule Tables, for this type of Schedule Tables the macros `HRUSEC()`, `HRMSEC()` and `HRSEC()` are provided. Depending on the hardware settings, certain nanosecond times may not be representable accurately. If this happens, the generator will issue a warning. However, the time conversion macros are based on the C pre-processor and cannot deliver such a warning. In addition, overflows may happen, and the calculation is not as sophisticated as the algorithm in the generator.

Therefore, if large times, very small times or very high precision is needed, time constants should be preferred.

3.2.5.1.7 Limits of the Synchronization Algorithm

The synchronization algorithm as described by the AUTOSAR standard V3.0 only corrects deviations of the past – it does not make assumptions about deviations of the present or the future. Differences in clock speed (between local time and global time) are not completely compensated.

Simplified synchronization algorithm: When `SyncScheduleTable` is called, the difference between the local schedule table time and the provided global time is computed and stored internally. Nothing more happens until an expiry point expires. Then, the times between subsequent expiry points are adapted. The adaptation stops once the computed deviation is compensated or a new global time is provided. In case new deviations between the global and local time occur, they are considered after the next call of `SyncScheduleTable` in the same way as just described.

As a result of this algorithm, a permanent deviation in the speed of global and local clock might not be compensated completely.

Example
The local schedule table time[2] runs 10 % slower then the global time. Whenever the global time reaches a multiple of 100, the function `SyncScheduleTable` is called to provide the global time to the schedule table. Both times start simultaneously at zero. When the global time reaches 100, the local time is 90 because of the 10 % difference. `SyncScheduleTable` is called and computes a difference of 10. We assume now that the difference is compensated until the next call of `SyncScheduleTable` occurs. The local time is then: 90+10+90=190 while the global time is 200. Again we have the same difference, so 10 needs to be corrected. The same occurs for all subsequent calls of `SyncScheduleTable`, too.

Although the computed difference between current time values of global and local time is corrected, the same difference occurs in the next synchronization step. However, using synchronization the deviation stays constant. Without synchronization it would accumulate.

3.2.5.1.8 Details about using NextScheduleTable

The AUTOSAR V3.0 standard leaves open certain details of using `NextScheduleTable` with synchronizable schedule tables. The following describes the implementation of `NextScheduleTable` MICROSAR OS.

If two schedule tables are chained using the API function `NextScheduleTable()`, the second schedule table takes over the synchronized schedule table time of the predecessor[3].

[2] The local time is based on the MCU's internal clock.

[3] Therefore, if the two schedule tables have a different LENGTH, switching from one to the other is undefined: it may work, but may also lead to unexpected results. This is not checked at runtime (and cannot be checked at generation time).

When switching to a schedule table that does not allow synchronization, the remaining difference to the global time is saved: upon switching to a schedule table that allows synchronization it will immediately start to synchronize.

3.2.5.1.9 Concurrent Actions

If a task is activated at an expiry point, and an event for this task is set at the same expiry point, always all tasks will be activated before events are set. However, if two schedule tables are using the same counter; one of these schedule tables activates a task and the other schedule table sets an event for this task at the same time, the behaviour is undefined.

3.2.5.2 High-Resolution Schedule Tables

The standard schedule tables use the System Timer Interrupt or a software counter, and thus offer the same resolution (defined by `TickTime`, typically one millisecond). While it is possible to choose a smaller `TickTime`, this increases the interrupt load. High-Resolution Schedule Tables offer a microsecond resolution or better[4] without unnecessary additional interrupt load: At each expiry point, a timer interrupt is reprogrammed so it will be reactivated exactly at the following expiry point[5].

Note that high resolution schedule tables are not supported by all MICROSAR OS implementations.

It is possible to use standard schedule tables and High-Resolution Schedule Tables at the same time. High-Resolution Schedule Tables support the full AUTOSAR V3.0 API, including synchronization (see 3.2.5.1) and are particularly suited for FlexRay.

3.2.5.2.1 Setup

To create a High-Resolution Schedule Table, create a Schedule Table and choose the `HiResSystemTimer` as the underlying counter. This counter is available only on the MICROSAR OS implementations that support high resolution schedule tables; it is automatically available if supported.

3.2.5.3 Cyclical Expiry Point Actions

Cyclical expiry point actions are a Vector specific extension of the Autosar Standard to ease the configuration of schedule table actions. In case an expiry point action shall be executed cyclicly within a schedule table, the user may select the sub-attribute `Cyclic`. This allows him to define a cycle time in the sub-attribute `CycleTime`. This informs the generator that the expiry point action shall occur repeatedly with the configured cycle time starting at the offset of the expiriy point, the action belongs to.

In case, the sub-attributes `Cyclic` and `CycleTime` have been configured, the generator of the OS copies the expiry point actions to the configured locations within the schedule table before it generates the schedule table. In case, there is already an expiry point at a location where a cyclical expiry point action shall occur, the cyclical action is simply added to the actions of that expiry point. In case there is no expiry point configured at the location

[4] The actually achievable best resolution depends on the hardware, hardware settings and application

[5] while the activation of the schedule table handler is done as exact as the underlying counter (the hardware) allows it, a certain time span will expire until the expiry point is actually processed. Interrupts, non-preemptive tasks and interrupt disabling times impose an additional jitter.

where a cyclical expiry point action schall occur, the generator invents an expiry point. Please note that generator invented expiry points do not allow synchronization as there is no configuration of the synchronization step width possible.

3.2.6 Service Protection

Service Protection is relevant for SC3 and SC4. Some features of the Service Protection effecting also SC1 and SC2 are described below.

3.2.6.1 Missing TerminateTask

SC1 and SC2:

In the scalability classes SC1 and SC2, a missing call of `TerminateTask` at the end of a task is detected only if the OS attribute `STATUS` is set to `EXTENDED`. In case, a missing call of `TerminateTask` is detected, MICROSAR OS calls the `ErrorHook` with the error number `E_OS_MISSINGEND` and after that calls `ShutdownOS`. (A release of resources is not performed for SC1 and SC2.)

3.2.6.2 Call of System Services with Interrupts disabled

SC1 and SC2:

In the scalability classes SC1 and SC2, the call of a system service with disabled interrupts is detected only if the OS attribute `STATUS` is set to `EXTENDED`. This detection is only possible when the OSEK system services for interrupt disabling and enabling are used exclusively. The reason is that most of the system services can be called from interrupt level, where at least some interrupts are disabled by the hardware.

In the case that a call of a system service with disabled interrupts is detected, the error hook is called with the error number `E_OS_DISABLEDINT` and the system service is not performed. The system service returns `E_OS_DISABLEDINT`.

This detection is quite similar to the detection of a system call with disabled interrupts in osCAN/OSEK OS, but in osCAN, this test was performed only when the attribute `OSInternalChecks` was set to `Additional` and the error number reported was `E_OS_SYS_DIS_INT`

3.2.7 Trusted Functions

Trusted functions are available for SC3 and SC4 only. MICROSAR OS OSEK/AUTOSAR provides two possibilities to call trusted functions: by direct call of API function `CallTrustedFunction` or by using generated stub functions.

It is possible to mix applications using direct calls and applications using generated stub functions.

Caution
Inside trusted functions there is full access to all memory. Therefore each trusted function with address arguments for return values must check the access rights of the caller before writing results through an address argument. The API provides the functions `CheckTaskMemoryAccess` and `CheckISRMemoryAccess` for address checking.

3.2.7.1 Generated Stub Functions

The generation of stubs for trusted functions is a Vector specific extension of the Autosar OS standard to ease the usage of trusted functions.

To enable stub generation for an application, the `TRUSTED` attribute of this application and the sub-attribute `GenerateStub` must be set to `TRUE`.

In this case a caller stub with the name `Call_<name>` is generated for each trusted function of the application, where `<name>` is the name of the trusted function. The generated stub function Call_<name> packs its parameters into a structure as needed by the standard API function CallTrustedFunction and calls that API function. Another generated stub-function performs an unpacking of the parameters so that the users trusted function needs not to get all the parameters via one pointer but can have a set of parameters and a return value like any legal C-function.

In case the sub-attribute `GenerateStub` is set to `TRUE`, the user has to define the parameters and the return value of the trusted function as well. The sub-attribute `Params` shall contain a comma separated list of type and parameter name (like they would occur in a function definition in C). The sub-attribute `ReturnType` shall define the return type of the function.

The stubs are generated into the file trustfct.c.

See 10.8 for an example using generated stub functions for trusted applications.

Caution
The generator does not produce prototypes for the trusted functions to be called by the trusted function stubs. The prototypes shall be provided by the writer of these functions and included into the file usrostyp.h. Parameter types and the return type need to be defined there also in case they are no simple types of the C-language. The file usrostyp.h is described in chapter 5.2.

3.3 Error Handling

3.3.1 Error Messages

If the kernel detects errors the OSEK error handling is called. The hook routine `ErrorHook` is called if selected.

Depending on the situation in which an error was detected the error handling will return to the current active task or the system will be shut down.

3.3.2 OSEK / AUTOSAR OS Error Numbers

The OSEK specification defines several error numbers which are returned by the API functions. A certain error number has different meanings for different API functions. The user has to know the API function to interpret the error number correctly.

With the AUTOSAR OS specification, the range of error numbers was extended. The following table shows all specified error numbers.

Error Code		Description
0	E_OK	Service executed successfully
1	E_OS_ACCESS	Several APIs: general access of object failure
2	E_OS_CALLEVEL	Several APIs: service accessed from wrong context
3	E_OS_ID	Several APIs: service called with wrong ID
4	E_OS_LIMIT	Several APIs: service called too often
5	E_OS_NOFUNC	Several APIs: (warning) service not executed
6	E_OS_RESOURCE	Several APIs: service called with occupied resource
7	E_OS_STATE	Several APIs: object is in wrong state
8	E_OS_VALUE	Several APIs: passed parameter has wrong value
9	E_OS_SERVICEID	Several APIs: service can not be called
10	E_OS_ILLEGAL_ADDRESS	Several APIs: invalid address passed
11	E_OS_MISSINGEND	Several APIs: task terminated without TerminatTask
12	E_OS_DISABLEDINT	Several APIs: service called with disabled interrupts
13	E_OS_STACKFAULT	Stack monitoring detected fault
14	E_OS_PROTECTION_MEMORY	Memory access violation
15	E_OS_PROTECTION_TIME	Execution time budget exceeded
16	E_OS_PROTECTION_ARRIVAL	Arrival before the timeframe expired
17	E_OS_PROTECTION_LOCKED	Task/ISR blocked too long (e.g. by disabled interrupts)
18	E_OS_PROTECTION_EXCEPTION	A trap occurred

Table 3-4 OSEK/AUTOSAR OS error numbers

The additional implementation specific error numbers are defined as:

Error Code		Description
20	E_OS_SYS_ASSERTION	This error is generated if the kernel detects an internal inconsistency. The reason and an exact explanation is described below.
21	E_OS_SYS_ABORT	This error is generated if the kernel has to shut down the system but the reason was not an API function.
22	E_OS_SYS_DIS_INT	This error number is no longer used. It is replaced by the AUTOSAR OS conformant number E_OS_DISABLEDINT.
23	E_OS_SYS_API_ERROR	This error is generated if an error occurs in an API function and there is no error code specified in the OSEK specification. The reason and an exact explanation is described below.
24	E_OS_SYS_ALARM_MANAGEMENT	A general warning issued in certain cases involving the alarm management. Detailed description in the implementation specific manual
25	E_OS_SYS_WARNING	A general warning issued in certain cases. Detailed description in the implementation specific manual.

Table 3-5 Implementation specific error numbers

More implementation specific errors may be described in ref. [4].

3.3.3 MICROSAR OS Error Numbers

In addition to the OSEK error numbers, all MICROSAR OS implementations provide unique error numbers for an exact error description. All error numbers are defined as a 16-bit value. The error numbers are defined in the header file `osekerr.h` and are defined according to the following syntax:

```
0xgfee
  ||+--- consecutive error number
  |+---- number of function in the function group
  +----- number of function group
```

The error numbers common to all MICROSAR OS implementations are described below. The implementation specific error numbers have a function group number >= 0xA000 and are described in the document [4].

To access these error numbers the `ERRORHOOK` has to be enabled. The numbers are then accessible via the macro `OSErrorGetosCANError()`.

Error Types:

Error Type	Description
OSEK	OSEK / AUTOSAR error. After calling the `ErrorHook`, the program is continued.
assertion	System assertion error. After calling the `ErrorHook` the operating system is shut down. Assertion checking is enabled by setting the attribute `OSInternalChecks` to `Additional` and the attribute `STATUS` to `EXTENDED` in the OIL Configurator.

Error Type	Description
syscheck	System error. After calling the `ErrorHook` the operating system is shut down. Refer to the specific error for a description how to enable or disable error checking.

Table 3-6 Error types

3.3.3.1 Error Numbers of Group Task Management / (1)

Group (1) contains the functions:

API Function	Abbreviation	Function Number
`ActivateTask`	AT	1
`TerminateTask`	TT	2
`ChainTask`	HT	3
`Schedule`	SH	4
`GetTaskState`	GS	5
`GetTaskID`	GI	6
`osMissingTerminateError`	MT	7

Table 3-7 API functions of group Task Management / (1)

Error numbers of group (1):

Error Code		Description	
		Error Type	Reason
0x1101	`osdErrATWrongTaskID`	OSEK	Called with invalid task ID
0x1102	`osdErrATWrongTaskPrio`	assertion	Task has wrong priority level
0x1103	`osdErrATMultipleActivation`	OSEK	number of activation of activated task exceeds limit
0x1104	`osdErrATIntAPIDisabled`	OSEK	Interrupts are disabled with functions provided by OSEK
0x1105	`osdErrATAlarmMultipleActivation`	OSEK	Number of activation of activated task exceeds limit (task activation is performed by alarm-expiration or expiry point action)
0x1106	`osdErrATNoAccess`	OSEK	Calling application has no access rights for this task
0x1107	`osdErrATCallContext`	OSEK	Called from invalid call context
0x1201	`osdErrTTDisabledInterrupts`	OSEK	`TerminateTask` called with disabled interrupts
0x1202	`osdErrTTResourcesOccupied`	OSEK	`TerminateTask` called with occupied resources
0x1203	`osdErrTTNotActivated`	assertion	`TerminateTask` attempted for a task with activation counter == 0 (not activated)

Error Code		Description	
		Error Type	Reason
0x1204	osdErrTTOnInterrupt Level	OSEK	TerminateTask called from an interrupt service routine
0x1205	osdErrTTNoImmediate TaskSwitch	assertion	TerminateTask has tried to start the Scheduler without success.
0x1206	osdErrTTCallContext	OSEK	Called from invalid call context
0x1301	osdErrHTInterrupts Disabled	OSEK	ChainTask called with disabled interrupts
0x1302	osdErrHTResources Occupied	OSEK	ChainTask called with occupied resources
0x1303	osdErrHTWrongTaskID	OSEK	New task has invalid ID
0x1304	osdErrHTNotActivated	assertion	Tried to terminate a task which have an activation counter which is zero
0x1305	osdErrHTMultiple Activation	OSEK	Number of activation of new task exceeds limit
0x1306	osdErrHTOnInterrupt Level	OSEK	ChainTask called on interrupt level
0x1307	osdErrHTWrongTask Prio	assertion	ChainTask was called from wrong priority level
0x1308	osdErrHTNoImmediate TaskSwitch	assertion	ChainTask has tried to activate the Scheduler without success.
0x1309	osdErrHTCallContext	OSEK	Called from invalid call context
0x130A	osdErrHTNoAccess	OSEK	Calling application has no access rights for this task
0x1401	osdErrSHInterrupts Disabled	OSEK	Schedule called with disabled interrupts
0x1402	osdErrSHOnInterrupt Level	OSEK	Schedule called on interrupt level
0x1403	osdErrSHScheduleNot Allowed	assertion	Schedule called from task with enabled stack sharing by setting NotUsingSchedule in the OIL Configurator
0x1404			No longer used
0x1405	osdErrSHResources Occupied	OSEK	Called with an occupied resource
0x1406	osdErrSHCallContext	OSEK	Called from invalid call context
0x1501	osdErrGSWrongTaskID	OSEK	Called with invalid task ID
0x1502	osdErrGSIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x1503	osdErrGSIllegalAddr	OSEK	Caller has no write access rights for address argument
0x1504	osdErrGSCallContext	OSEK	Called from invalid call context
0x1505	osdErrGSNoAccess	OSEK	Calling application has no access rights for

Error Code		Description	
		Error Type	Reason
			this task
0x1601	osdErrGIIntAPIDisabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x1602	osdErrGIIllegalAddr	OSEK	Caller has no write access rights for address argument.
0x1603	osdErrGICallContext	OSEK	Called from invalid call context
0x1701	osdErrMTMissingTerminateTask	syscheck	Exit of task without the call of TerminateTask or ChainTask. This error is detected in EXTENDED STATUS only.

Table 3-8 Error numbers of group Task Management / (1)

3.3.3.2 Error Numbers of Group Interrupt Handling / (2)

Group (2) contains the functions:

API Function	Abbreviation	Function Number
EnableAllInterrupts	EA	4
DisableAllInterrupts	DA	5
ResumeOSInterrupts	RI	6
SuspendOSInterrupts	SI	7
osUnhandledException	UE	8
osSaveDisableLevelNested	SD	9
osRestoreEnableLevelNested	RE	A
osSaveDisableGlobalNested	SG	B
osRestoreEnableGlobalNested	RG	C
ResumeAllInterrupts	RA	D
SuspendAllInterrupts	SA	E

Table 3-9 API functions of group Interrupt Handling / (2)

Error numbers of group (2):

Error Code		Description	
		Error Type	Reason
0x2401	osdErrEAIntAPIWrongSequence	assertion	DisableAllInterrupts not called before
0x2501	osdErrDAIntAPIDisabled	assertion	Interrupts are disabled with functions provided by OSEK

Error Code		Description	
		Error Type	Reason
0x2801	osdErrUEUnhandled Exception	syscheck	An unhandled exception or interrupt was detected. This error check is always enabled.
0x2901	osdErrSDWrongCounter	assertion	Wrong counter value detected
0x2A01	osdErrREWrongCounter	assertion	Wrong counter value detected
0x2B01	osdErrSGWrongCounter	assertion	Wrong counter value detected
0x2C01	osdErrRGWrongCounter	assertion	Wrong counter value detected

Table 3-10 Error numbers of group Interrupt Handling / (2)

3.3.3.3 Error Numbers of Group Resource Management / (3)

Group (3) contains the functions:

API Function	Abbreviation	Function Number
GetResource	GR	1
ReleaseResource	RR	2

Table 3-11 API functions of group Resource Management / (3)

Error numbers of group (3):

Error Code		Description	
		Error Type	Reason
0x3101	osdErrGRWrongResource ID	OSEK	Invalid resource ID
0x3102	osdErrGRPriority Occupied	assertion	Ceiling priority of the specified resource already in use
0x3103	osdErrGRResource Occupied	OSEK	Resource already occupied
0x3104	osdErrGRNoAccess Rights	assertion	Task has no access to the specified resource
0x3105	osdErrGRWrongPrio	OSEK	Specified resource has a wrong priority. Possible reason: the task has no access rights to this resource.
0x3106	osdErrGRIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x3107	osdErrGRNoAccess	OSEK	Calling application has no access rights for this resource
0x3108	osdErrGRCallContext	OSEK	Called from invalid call context
0x3109	osdErrGRISRNoAccess	OSEK	Calling ISR has no access rights for this

Error Code		Description	
		Error Type	Reason
	Rights		resource
0x3201	osdErrRRWrongResourceID	OSEK	Invalid resource ID
0x3202	osdErrRRCeilingPriorityNotSet	assertion	Ceiling priority of the resource not found in the ready bit field
0x3203	osdErrRRWrongTask	assertion	Resource occupied by a different task
0x3204	osdErrRRWrongPrio	OSEK	Specified resource has a wrong priority. Possible reason: the task has no access rights to this resource.
0x3206	osdErrRRNotOccupied	OSEK	The specified resource is not occupied by the task
0x3207	osdErrRRWrongSequence	OSEK	At least one other resource must be released before
0x3208	osdErrRRIntAPIDisabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x3209	osdErrRRNoAccess	OSEK	Calling application has no access rights for this resource
0x320A	osdErrRRCallContext	OSEK	Called from invalid call context
0x320B	osdErrRRISRNoAccessRights	OSEK	Calling ISR has no access rights for this resource

Table 3-12 Error numbers of group Resource Management / (3)

3.3.3.4 Error Numbers of Group Event Control / (4)

Group (4) contains the functions:

API Function	Abbreviation	Function Number
SetEvent	SE	1
ClearEvent	CE	2
GetEvent	GE	3
WaitEvent	WE	4

Table 3-13 API functions of group Event Control / (4)

Error numbers of group (4):

Error Code		Description	
		Error Type	Reason
0x4101	osdErrSEWrongTaskID	OSEK	Invalid task ID
0x4102	osdErrSENotExtendedTask	OSEK	Cannot SetEvent to basic task

Error Code		Description	
		Error Type	Reason
0x4103	osdErrSETaskSuspended	OSEK	Cannot `SetEvent` to task in `SUSPENDED` state. The error code might occur in case of API call `SetEvent` or in case of alarm/schedule table action to set an event.
0x4104	osdErrSEWrongTask Prio	assertion	Wrong task priority detected
0x4105	osdErrSEIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x4106	osdErrSECallContext	OSEK	Called from invalid call context
0x4107	osdErrSENoAccess	OSEK	Calling application has no access rights for this task
0x4201	osdErrCENotExtended Task	OSEK	A basic task cannot clear an event
0x4202	osdErrCEOnInterrupt Level	OSEK	`ClearEvent` called on interrupt level
0x4203	osdErrCEIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x4204	osdErrCECallContext	OSEK	Called from invalid call context
0x4301	osdErrGEWrongTaskID	OSEK	Invalid task ID
0x4302	osdErrGENotExtended Task	OSEK	Cannot `GetEvent` from basic task
0x4303	osdErrGETaskSuspended	OSEK	Cannot `GetEvent` from a task in `SUSPENDED` state
0x4304	osdErrGEIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x4305	osdErrGEIllegalAddr	OSEK	Caller has no write access rights for address argument
0x4306	osdErrGECallContext	OSEK	Called from invalid call context
0x4307	osdErrGENoAccess	OSEK	Calling application has no access rights for this task
0x4401	osdErrWENotExtended Task	OSEK	`WaitEvent` called by basic task
0x4402	osdErrWEResources Occupied	OSEK	`WaitEvent` called with occupied resources
0x4403	osdErrWEInterrupts Disabled	OSEK	`WaitEvent` called with disabled interrupts
0x4404	osdErrWEOnInterrupt Level	OSEK	`WaitEvent` called on interrupt level
0x4405	osdErrWECallContext	OSEK	Called from invalid call context

Table 3-14 Error numbers of group Event Control / (4)

3.3.3.5 Error Numbers of Group Alarm Management / (5)

Group (5) contains the functions:

API Function	Abbreviation	Function Number
GetAlarmBase	GB	1
GetAlarm	GA	2
SetRelAlarm	SA	3
SetAbsAlarm	SL	4
CancelAlarm	CA	5
osWorkAlarms	WA	6

Table 3-15 API functions of group Alarm Management / (5)

Error numbers of group (5):

Error Code		Description	
		Error Type	Reason
0x5101	osdErrGBWrongAlarmID	OSEK	Invalid alarm ID
0x5102	osdErrGBIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x5103	osdErrGBIllegalAddr	OSEK	Caller has no write access rights for address argument
0x5104	osdErrGBCallContext	OSEK	Called from invalid call context
0x5105	osdErrGBNoAccess	OSEK	Calling application has no access rights for this alarm
0x5201	osdErrGAWrongAlarmID	OSEK	Invalid alarm ID
0x5202	osdErrGANotActive	OSEK	Alarm not active
0x5203	osdErrGAIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x5204	osdErrGAIllegalAddr	OSEK	Caller has no write access rights for address argument
0x5205	osdErrGACallContext	OSEK	Called from invalid call context
0x5206	osdErrGANoAccess	OSEK	Calling application has no access rights for this alarm
0x5301	osdErrSAWrongAlarmID	OSEK	Invalid alarm id
0x5302	osdErrSAAlreadyActive	OSEK	Alarm already active
0x5303	osdErrSAWrongCycle	OSEK	Specified cycle is out of range
0x5304	osdErrSAWrongDelta	OSEK	Specified delta is out of range
0x5305	osdErrSAIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x5306	osdErrSAZeroIncrement	OSEK	SetRelAlarm was called with the

Error Code		Description	
		Error Type	Reason
			parameter increment set to zero. (This is no longer allowed with AUTOSAR OS)
0x5307	osdErrSACallContext	OSEK	Called from invalid call context
0x5308	osdErrSANoAccess	OSEK	Calling application has no access rights for this alarm
0x5401	osdErrSLWrongAlarmID	OSEK	Invalid alarm ID
0x5402	osdErrSLAlreadyActive	OSEK	Alarm already active
0x5403	osdErrSLWrongCycle	OSEK	Specified cycle is out of range
0x5404	osdErrSLWrongStart	OSEK	Specified start is out of range
0x5405	osdErrSLIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x5406	osdErrSLCallContext	OSEK	Called from invalid call context
0x5407	osdErrSLNoAccess	OSEK	Calling application has no access rights for this alarm
0x5501	osdErrCAWrongAlarmID	OSEK	Invalid alarm ID
0x5502	osdErrCANotActive	OSEK	Alarm not active
0x5503	osdErrCAIntAPI Disabled	OSEK	Interrupts are disabled with functions provided by OSEK
0x5504	osdErrCAAlarmInternal	syscheck	Internal error detected while alarm was cancelled. This error is only detected when OSInternalChecks is set to Additional.
0x5505	osdErrCACallContext	OSEK	Called from invalid call context
0x5506	osdErrCANoAccess	OSEK	Calling application has no access rights for this alarm

Table 3-16 Error numbers of group Alarm Management / (5)

3.3.3.6 Error Numbers of Group Operating System Execution Control / (6)

Group (6) contains the functions:

API Function	Abbreviation	Function Number
osCheckStackOverflow	SO	1
osSchedulePrio	SP	2
osGetStackUsage	SU	3
osCheckLibraryVersionAnd Variant	CL	4
osErrorHook	EH	5
StartOS	ST	6

Table 3-17 API functions of group Operating System Execution Control / (6)

Error numbers of group (6):

Error Code		Description	
		Error Type	Reason
0x6101	`osdErrSOStackOverflow`	syscheck	Task stack overflow detected. This error is only detected when the OIL attribute `WithStackCheck` is set to `TRUE`.
0x6201	`osdErrSPInterrupts Enabled`	assertion	Scheduler called with enabled interrupts
0x6301	`osdErrSUWrongTaskID`	assertion	Called with invalid task ID
0x6401	`osdErrCLWrongLibrary`	syscheck	Wrong library linked to application. This error check is always enabled.
0x6501	`osdErrEHInterrupts Enabled`	assertion	`ErrorHook` called with enabled interrupts
0x6601	`osdErrSTMemoryError`	assertion	`StartOS` failed while initializing memory.
0x6602	`osdErrSTNoImmediate TaskSwitch`	assertion	`StartOS` tried to activate the Scheduler without success.
0x6603	`OsdErrSTWrongAppMode`	syscheck	`StartOS` was called with an invalid parameter value. This error is only detected if the attribute `STATUS` is set to `EXTENDED`.

Table 3-18 Error numbers of group Operating System Execution Control / (6)

3.3.3.7 Error Numbers of Schedule Table Control / (7)

(Note: Schedule table errors for synchronization may be found in chapter 3.3.3.9)

Group (7) contains the functions:

API Function	Abbreviation	Function Number
`StartScheduleTableRel`	SR	1
`StartScheduleTableAbs`	SS	2
`StopScheduleTable`	SP	3
`GetScheduleTableStatus`	SG	4
`NextScheduleTable`	SN	5
`osWorkScheduleTables`	WS	6

Table 3-19 API functions of group Schedule Table Control / (7)

Error numbers of group (7):

Error Code		Description	
		Error Type	Reason
0x7101	osdErrSRWrongID	OSEK	StartScheduleTableRel was called with an invalid schedule table ID.
0x7102	osdErrSRAlready RunningOrNext	OSEK	StartScheduleTableRel was called for a schedule table that is already running or next.
0x7103	osdErrSRZeroOffset	OSEK	StartScheduleTableRel was called with the parameter Offset set to zero.
0x7104	osdErrSROffsetTooBig	OSEK	StartScheduleTableRel was called with the parameter Offset bigger than MAXALLOWEDVALUE of the respective counter.
0x7105	osdErrSRIntAPI Disabled	OSEK	StartScheduleTableRel was called with disabled interrupts.
0x7106	osdErrSRCallContext	OSEK	Called from invalid call context
0x7107	osdErrSRNoAccess	OSEK	Calling application has no access rights for this schedule table
0x7109	osdErrSRImplicite Sync	OSEK	StartScheduleTableRel was called for an implicitly synchronized ScheduleTable
0x7201	osdErrSSWrongID	OSEK	StartScheduleTableAbs was called with an invalid schedule table ID.
0x7202	osdErrSSAlready RunningOrNext	OSEK	StartScheduleTableAbs was called for a schedule table, which is already running or next.
0x7203	osdErrSSTickvalueToo Big	OSEK	StartScheduleTableAbs was called with the parameter TickValue bigger than MAXALLOWEDVALUE of the respective counter.
0x7204	osdErrSSIntAPI Disabled	OSEK	StartScheduleTableAbs was called with disabled interrupts.
0x7205	osdErrSSCallContext	OSEK	Called from invalid call context
0x7206	osdErrSSNoAccess	OSEK	Calling application has no access rights for this schedule table
0x7301	osdErrSPWrongID	OSEK	StopScheduleTable was called with an invalid schedule table ID.
0x7302	osdErrSPNotRunning	OSEK	StopScheduleTable was called for a schedule table, which is in stopped or next state.
0x7303	osdErrSPIntAPI Disabled	OSEK	StopScheduleTable was called with disabled interrupts.
0x7304	osdErrSPCallContext	OSEK	Called from invalid call context
0x7305	osdErrSPNoAccess	OSEK	Calling application has no access rights for this schedule table

Error Code		Description	
		Error Type	Reason
0x7306	osdErrSPUnknownCase	Assertion	An internal error occured
0x7401	osdErrSGWrongID	OSEK	GetScheduleTableStatus was called with an invalid schedule table ID.
0x7402	osdErrSGIntAPI Disabled	OSEK	GetScheduleTableStatus was called with disabled interrupts
0x7403	osdErrSGCallContext	OSEK	Called from invalid call context
0x7404	osdErrSGNoAccess	OSEK	Calling application has no access rights for this schedule table
0x7405	osdErrSGIllegalAddr	OSEK	Caller has no write access rights for address argument
0x7501	osdErrSNWrongCurrent ID	OSEK	NextScheduleTable was called with an invalid schedule table ID for the parameter ScheduleTableID_current.
0x7502	osdErrSNWrongNextID	OSEK	NextScheduleTable was called with an invalid schedule table ID for the parameter ScheduleTableID_next.
0x7503	osdErrSNNotRunning	OSEK	NextScheduleTable was called to chain a schedule table after another schedule table, that is currently not running.
0x7504	osdErrSNAlready RunningOrNext	OSEK	NextScheduleTable was called to chain a running schedule table after another schedule table.
0x7505	osdErrSNDifferent Counters	OSEK	NextScheduleTable was called to chain two schedule tables, which are driven by different counters.
0x7506	osdErrSNIntAPI Disabled	OSEK	NextScheduleTable was called with interrupts disabled.
0x7507	osdErrSNCallContext	OSEK	Called from invalid call context
0x7508	osdErrSNNoAccess	OSEK	Calling application has no access rights for this schedule table
0x7601	osdErrWSUnknownAction	Assertion	An invalid action was found in a schedule table
0x7602	osdErrWSUnknown Reaction	Assertion	An internal error occured

Table 3-20 Error numbers of group Schedule Table Control / (7)

3.3.3.8 Error Numbers of Group Other SC1 Functions / (8)

Group (8) contains the functions:

API Function	Abbreviation	Function Number
IncrementCounter	IC	1
GetISRID	II	2

Table 3-21 API functions of group Other SC1 Functions / (8)

Error numbers of group (8):

Error Code		Description	
		Error Type	Reason
0x8101	osdErrICWrongCounterID	OSEK	IncrementCounter was called for an invalid counter or a hardware counter.
0x8102	osdErrICIntAPIDisabled	OSEK	IncrementCounter was called with interrupts disabled.
0x8103	osdErrICCallContext	OSEK	Called from invalid call context
0x8104	osdErrICNoAccess	OSEK	Calling application has no access rights for this counter
0x8201	osdErrIIIntAPIDisabled	OSEK	GetISRID was called with interrupts disabled.
0x8202	osdErrIICallContext	OSEK	Called from invalid call context

Table 3-22 Error numbers of group Other SC1 Functions / (8)

3.3.3.9 Error Numbers of Group Other SC2, SC3, SC4 Functions / (9)

Group (9) contains the functions:

API Function	Abbreviation	Function Number
GetTaskMinInterArrivalTime	TM	0
CallTrustedFunction	CT	3
TerminateApplication	TA	4
SyncScheduleTable	SY	5
SetScheduleTableAsync	AY	6
BlockingTimeMonitoring	BM	7
GetTaskMaxExecutionTime	TE	8
GetISRMaxExecutionTime	IE	9
GetTaskMaxBlockingTime	TB	A
GetISRMaxBlockingTime	IB	B
StartScheduleTableSynchron	TS	C
ExecutionTimeMonitoring	ET	D
ISR exit	IX	E

API Function	Abbreviation	Function Number
GetISRMinInterArrivalTime	MI	F

Table 3-23 API functions of group Other SC2, SC3, SC4 Functions / (9)

Error numbers of group (9):

Error Code		Description	
		Error Type	Reason
0x9001	osdErrTMWrongTaskID	OSEK	Called with wrong TASK ID
0x9002	osdErrTMNoAccess	OSEK	The calling application has no access rights for the TASK
0x9003	osdErrTMIllegalAddr	OSEK	The caller has no access rights for the memory region
0x9301	osdErrCTWrongFctIdx	OSEK	Invalid function index for trusted function
0x9302	osdErrCTCallContext	OSEK	Called from invalid call context
0x9303	osdErrCTIntAPI Disabled	OSEK	Called with interrupts disabled
0x9401	osdErrTAWrongRestart Option	OSEK	Invalid restart option
0x9402	osdErrTACallContext	OSEK	Called from invalid call context
0x9403	osdErrTAIntAPI Disabled	OSEK	Called with interrupts disabled
0x9501	osdErrSYCallContext	OSEK	Called from invalid call context
0x9502	osdErrSYWrongID	OSEK	Called with wrong schedule table ID
0x9503	osdErrSYNoAccess	OSEK	Calling application has no access rights for this schedule table
0x9504	osdErrSYIntAPI Disabled	OSEK	Called with interrupts disabled
0x9505	osdErrSYSTNotRunning	OSEK	The Schedule table is currently not running
0x9506	osdErrSYGlobalTimeToo Big	OSEK	The Global Time is larger than the LENGTH of the schedule table
0x9507	osdErrSYSyncKindNot Explicit	OSEK	SyncScheduleTable was called for a Schedule table which is not explicitly synchronized.
0x9601	osdErrAYCallContext	OSEK	Called from invalid call context
0x9602	osdErrAYWrongID	OSEK	Called with wrong schedule table ID
0x9603	osdErrAYNoAccess	OSEK	Calling application has no access rights for this schedule table
0x9604	osdErrAYIntAPI	OSEK	Called with interrupts disabled

Error Code		Description	
		Error Type	Reason
	Disabled		
0x9702	osdErrBMResAlready Measured	Assertion	A blocking time measurement was started that is already running. This might happen if timing protection is active and `SuspendAllInterrupts` is called after `DisableAllInterrupts` has already been called.
0x9703	osdErrBMInvalidProces sInStart	Assertion	Internal error: attempt to start Block Timing Protection with an invalid task or ISR
0x9704	osdErrBMInvalidProces sInStop	Assertion	Internal error: attempt to stop Block Timing Protection with an invalid task or ISR
0x9801	osdErrTEWrongTaskID	OSEK	`GetTaskMaxExecutionTime` was called with an invalid task identifier
0x9802	osdErrTENoAccess	OSEK	The calling application has no access rights for this task
0x9803	osdErrTEIllegalAddr	OSEK	The caller has no access rights for this memory region
0x9901	osdErrIEWrongISRID	OSEK	`GetISRMaxExecutionTime` was called with an invalid ISR identifier
0x9902	osdErrIENoAccess	OSEK	The calling application has no access rights for this ISR
0x9903	osdErrIEIllegalAddr	OSEK	The caller has no access rights for this memory region.
0x9A01	osdErrTBWrongTaskID	OSEK	Called with wrong Task ID
0x9A02	osdErrTBWrongBlock Type	OSEK	Called with wrong blocking type
0x9A03	osdErrTBWrongResource ID	OSEK	Called with wrong resource ID
0x9A04	osdErrTBNoAccessTo Task	OSEK	The calling application has no access rights for the task
0x9A05	osdErrTBNoAccessTo Resource	OSEK	The calling application has no access rights for the resource
0x9A06	osdErrTBIllegalAddr	OSEK	The caller has no access rights for this memory region
0x9B01	osdErrIBWrongISRID	OSEK	Called with wrong ISR ID
0x9B02	osdErrIBWrongBlock Type	OSEK	Called with wrong blocking type
0x9B03	osdErrIBWrongResource ID	OSEK	Called with wrong resource ID
0x9B04	osdErrIBNoAccessToISR	OSEK	The calling application has no access rights for the ISR
0x9B05	osdErrIBNoAccessTo Resource	OSEK	The calling application has no access rights for the resource

Error Code		Description	
		Error Type	Reason
0x9B06	osdErrIBIllegalAddr	OSEK	The caller has no access rights for the memory region
0x9C01	osdErrTSCallContext	OSEK	Called from invalid call context
0x9C02	osdErrTSWrongID	OSEK	Called with invalid schedule table id.
0x9C03	osdErrTSNoAccess	OSEK	Calling application has no access rights for this schedule table
0x9C04	osdErrTSIntAPI Disabled	OSEK	Called with interrupts disabled
0x9C05	osdErrTSSTAlready Running	OSEK	The schedule table is already running or scheduled to run after a currently running schedule table
0x9C06	osdErrTSGlobalTimeToo Big	OSEK	The offset to Global Time is larger than the LENGTH of the schedule table
0x9C08	osdErrTSSyncKindNot Explicit	OSEK	StartScheduleTableSynchron was called for a Schedule table which is not explic-itly synchronized.
0x9D01	osdErrETNoCurrent Process	Assertion	Execution Time Monitoring has detected an invalid process ID
0x9E01	osdErrIXResources Occupied	OSEK	An ISR of category 2 was left with resources still occupied.
0x9E02	osdErrIXIntAPI Disabled	OSEK	An ISR of category 2 was left with interrupts disabled by DisableAllInterrupts, SuspendAllInterrupts or SuspendOSInterrupts
0x9F01	osdErrMIWrongISRID	OSEK	Called with wrong ISR ID
0x9F02	osdErrMINoAccess	OSEK	The calling application has no access rights for the ISR
0x9F03	osdErrMIIllegalAddr	OSEK	The caller has no access rights for the memory region

Table 3-24 Error numbers of group Other SC2, SC3, SC4 Functions / (9)

3.3.3.10 Error Numbers of Group Messages / (B)

Group (B) contains the functions:

API Function	Abbreviation	Function Number
StartCOM	SC	1
StopCOM	TC	2
SendMessage	SM	3

API Function	Abbreviation	Function Number
ReceiveMessage	RM	4
GetMessageStatus	MS	5
GetCOMApplicationMode	AM	6
InitMessage	IM	7
COMErrorHook	CR	8

Table 3-25 API functions of group Messages / (B)

Error numbers of group (B):

Error Code		Description	
		Error Type	Reason
0xB101	osdErrSCWrongModeID	OSEK	StartCOM was called with invalid COM application mode.
0xB102	osdErrSCCallContext	OSEK	Called from invalid call context
0xB201	osdErrTCWrongModeID	OSEK	StopCOM was called with invalid shutdown mode.
0xB202	osdErrTCCallContext	OSEK	Called from invalid call context
0xB301	osdErrSMWrongID	OSEK	SendMessage was called with invalid message ID.
0xB302	osdErrSMNoAccess	OSEK	The calling application has no access rights for this message.
0xB303	osdErrSMCallContext	OSEK	Called from invalid call context
0xB401	osdErrRMWrongID	OSEK	ReceiveMessage was called with invalid message ID.
0xB402	osdErrRMLimit	OSEK	The maximum number of messages was exceeded for a queued message.
0xB403	osdErrRMNoMessage	OSEK	ReceiveMessage was called for a queued message with empty queue.
0xB404	osdErrRMNoAccess	OSEK	The calling application has no access rights for this message.
0xB405	osdErrRMIllegalAddr	OSEK	The caller has no write access rights for the referenced memory region.
0xB406	osdErrRMCallContext	OSEK	Called from invalid call context
0xB501	osdErrMSWrongID	OSEK	GetMessageStatus was called with invalid message ID.
0xB502	osdErrMSLimit	OSEK	The maximum number of messages was exceeded for a queued message.
0xB503	osdErrMSNoMessage	OSEK	GetMessageStatus was called for a queued message with empty queue.
0xB504	osdErrMSNoAccess	OSEK	The calling application has no access rights for this message.

Error Code		Description	
		Error Type	Reason
0xB505	osdErrMSCallContext	OSEK	Called from invalid call context
0xB701	osdErrIMWrongID	OSEK	InitMessage was called with invalid message ID.
0xB702	osdErrIMNoAccess	OSEK	The calling application has no access rights for this message.
0xB703	osdErrIMCallContext	OSEK	Called from invalid call context
0xB801	osdErrCRInterrupts Enabled	assertion	COMErrorHook called with enabled interrupts.

Table 3-26 Error numbers of group Messages / (B)

Caution
Implementation specific error numbers are described in the document [4].

3.3.4 ErrorInfoLevel

Every MICROSAR OS implementation offers an additional service for error treatment. To use this feature the OS properties EXTENDED_STATUS and ERRORHOOK have to be enabled. The OS property ErrorInfoLevel has to be set on Modulnames.

When this is done the macros OSErrorGetosCANModulName() and OSErrorGetosCANLineNumber() are enabled. Usage of the macros is as follows:

> OSErrorGetosCANModulName(): Returns the name of the file in which the error occurred.

> OSErrorGetosCANLineNumber(): Returns the line number in which the error occurred.

3.3.5 Reactions on Error Situations

Depending on which errors have occurred, different reactions are performed:

> Errors detected from wrong usage of API functions: Call of ErrorHook and return to the calling task or interrupt routine.

> Errors detected in the kernel: Call of ErrorHook and call of ShutdownOS (which calls ShutdownHook).

4 Installation

The MICROSAR OS package might be delivered together with other MICROSAR embedded software. In this case, the installation is described elsewhere. If MICROSAR OS is delivered stand alone, it comes up with an installation program, which installs the operating system source files, the OIL Configurator and the XML Converter.

4.1 Installation Requirements

The installation program and the OIL Configurator are 32-bit Windows programs.

Requirements:

> Microsoft Windows95, Windows98, Windows NT, Windows 2000, Windows XP, Windows Vista

> 64 MByte of free disk space (for a complete installation)

4.2 Installation Disk

All parts of the OSEK system, the OIL Configurator, and the code generator are delivered with a Windows installation program. The installation program copies all files onto the local hard disk and sets all paths in the INI files. The installation program asks the user for an installation path; this path is the root path for all installed components. The selected path is referred to in the following as `root`. The delivered installation uses the path `C:\OSEK` as the default `root` path.

There are two possible installation styles than can be selected:

> MICROSAR style: compatible with Vector AUTOSAR stack

> osCAN style: compatible with osCAN

The installation paths are determined depending on the selected style

The installed components are:

Components	osCAN style	MICROSAR style
OIL Configurator	root\OILTOOL	root\Generators\Tools\OilTool
OSEK system	root\HwPlatform	root\BSW\Os

Table 4-1 Installed components

4.3 OIL Configurator

Info
Please note that 'OIL Configurator' and 'OIL Tool' are used as synonyms in this document.

The OIL Configurator is a common tool for different OSEK implementations. The implementation specific parts are the code generator and the OIL implementation files for the code generator.

Components	osCAN style	MICROSAR style
OIL Configurator	root\OILTOOL	root\Generators\Tools\OilTool
XML to OIL Converter	root\OILTOOL	root\Generators\Tools\OilTool
OIL implementation files	root\OILTOOL\GEN	root\Generators\Os
Code generator	root\OILTOOL\GEN	root\Generators\Os

Table 4-2 System configuration and generation tools

4.3.1 INI Files of the OIL Tool

The OIL Configurator has two INI files which are in the directory of the OIL Configurator:

- `OILGEN.INI`
- `OILCFG.INI`

4.3.2 OIL Implementation Files

The implementation files are copied onto the local hard disk by the installation program. The OIL tool has knowledge about these files through the INI file `OILGEN.INI` (the correct path is set by the installation program).

The implementation files are described in the hardware specific part of this manual [4].

4.3.3 Code Generator

The code generator `GENxxxx.EXE` is copied onto the local hard disk by the installation program. The code generator is defined in the INI-file `OILGEN.INI`. ('xxxx' has to be replaced by a hardware dependent abbreviation)

4.4 OSEK Operating System

4.4.1 Installation Paths

The delivered operating system parts are organized in different subdirectories. The delivered examples assume the following structure for osCAN style installations:

> root\HwPlatform\APPL\Compiler\Derivative	Sample applications
> root\HwPlatform\BIN	executable files (e.g. make tool)

> root\HwPlatform\bswmd_files	XML parameter descrption files
> root\HwPlatform\DOC	Documentation
> root\HwPlatform\INCLUDE	OSEK include files
> root\HwPlatform\LIB	OSEK library (only if a library is available)
> root\HwPlatform\SRC	OSEK sources (C and Assembler)

The following structure is used by MICROSAR style installations:

> root\Demo\Os	Sample applications
> root\Generators\Os	executable files (e.g. make tool)
> root\Generators\Components_Schemes\ Os_<platform and derivate>_bswmd \bswmd	XML parameter descrption files
> root\Doc\TechnicalReferences > root \Doc\UserManuals	Documentation
> root\BSW\Os	OSEK include files
> root\BSW\Os	OSEK library (only if a library is available)
> root\BSW\Os	OSEK sources (C and Assembler)

4.5 Applications

The APPL (osCAN style) or Demo\Os (MICROSAR style) directories contain subdirectories with application examples. The paths of examples are structured as follows:

> `[APPL | Demo\Os] \Compiler\Derivative\ExampleName`

Chapter 10 describes the delivered application examples.

Files in these directories, e.g. the startup code, are of exemplary nature and may be adapted to run the examples on specific hardware.

4.6 XML Converter

AUTOSAR uses for configuration files the XML format. An XML Schema (ref. [8]) defines the structure. For each derivative there is an ECU Parameter Definition File (file extension is `arxml`) which defines all attributes (standard attribute and vendor/platform specific attributes).

The Vector implementation of AUTOSAR OS uses OIL [6] as an intermediate configuration file format. ECU Configuration files are converted to OIL files before code generators generate code. The conversion can be started from command line and / or make files. The converter (`AsrToOil.exe`) will be installed in the directory of the OIL Configurator.

4.6.1 Invoking the Converter

AsrToOil.exe uses the DLL `OILDEF32.DLL`, which also is installed into the directory of the OIL Configurator.

AsrToOil is started with two parameters:

> Parameter 1: Name of AUTOSAR ECU Configuration file

> Parameter 2: Name of output file (OIL file)

Example

```
AsrToOil genExampleTricore1766_ecuc.arxml genExampleTricore1766.oil
```

Caution

AsrToOil does not need any other files for conversion.

4.6.2 Parameter Definition Files

Parameter Definition Files for the implementation can be found in the directory `root\HwPlatform\BSWMD_files` (osCAN style) or

`root\Generators\Components\_Schemes\Os_<platform and derivate>_bswmd\bswmd` (MICROSAR style).

The files have the name `OS_<platform and derivate>_bswmd.arxml`.

5 Integration

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

5.1 Scope of Delivery

The delivery of the OS contains the files which are described in the chapters 5.1.1 and 5.1.2:

5.1.1 Static Files

The static file list is described in the platform specific technical reference [4]

5.1.2 Dynamic Files

The dynamic files are generated by the code generator GENxxxx (xxxx is replaced by hardware platform name) and by the application template generator GENTMPL.

5.1.2.1 Code Generator GENxxxx

File Name	Description
tcb.c	tcb contains the task control block and other OS object
tcb.h	task and other OS object related information, like task Ids
msg.c	Message related data
msg.h	Message related information, e.g. message Ids
trustfct.h	Header containing trusted function information
trustfct.c	Trusted function data and generated stubs
libconf	Information for usage in makefiles, not available on all platforms, see chapter 5.1.2.1.1
<OILFileName>.ort	Generated if kernel aware debugging with the ORTI interface is enabled,

Table 5-1 Files generated by code generator GENxxxx

In addition to the files listed in Table 5-1 some hardware dependent files are generated which are described in the hardware specific technical reference [4].

5.1.2.1.1 Generated file libconf

The file libconf is meant for the inclusion into makefiles. It sets some variables in accordance to general configuration settings of MICROSAR OS to inform the make process about them. Dependent on the platform, the file may contain more information or be even unavailable, so please see the hardware specific technical reference [4].

The table below describes the generated variables.

Variable	Meaning
LIB	Is set to 1 in case MICROSAR OS is configured to library variant. Is set to 0 if not.
STATUS_LEVEL	Reflects the setting of the configuration attribute STATUS of MICROSAR OS. Possible values: EXTENDED_STATUS = 1 STANDARD_STATUS = 0
DEBUG_SUPPORT	Is set to 1 in case the configuration attribute ORTIDebugSupport of MICROSAR OS is selected. Is set to 0 if not.

Table 5-2 Variables generated into the file libconf

5.1.2.2 Application Template Generator GENTMPL

File Name	Description
main.c	This module contains the main function (chapter 8.2.2.5) and the code templates for the OSEK objects defined in the OIL file. The templates are generated for the OSEK objects `Task` (chapter 8.2.2.1), `ISR` (chapter 8.2.2.2), `Hook Routines` (chapter 8.2.2.3) and `Event` (chapter 8.2.2.4).

Table 5-3 Files generated by application template generator GENTMPL

5.2 Include Structure

The header file `tcb.h` is included into the file `os.h`. The user must include `os.h` in every module of his application. The header `tcb.h` is included automatically. Always recompile all files after a new generation of `tcb.h`.

If an application is using COM messages or trusted functions, an include file named `usrostyp.h` must be present in the include path. This file must contain all user specific data types used for messages and trusted functions.

6 API Description

6.1 Standard API - Overview

This chapter gives an overview of all standard API functions defined for the OS. The following synonyms present the standard specifications:

> ASR: AUTOSAR standard, reference [1]

> OSEK: OSEK standard, reference [3]

These standard specifications contain the detailed API descriptions. In case part of an API function is implementation specific, the detailed API description is given in a further subchapter in this document.

API Function Prototype	Standard Specification		Scalability Class			
	OSEK	ASR	1	2	3	4
Task Handling						
`StatusType ActivateTask (TaskType TaskID)`	■		■	■	■	■
`StatusType TerminateTask (void)`	■		■	■	■	■
`StatusType ChainTask (TaskType TaskID)`	■		■	■	■	■
`StatusType Schedule (void)`	■		■	■	■	■
`StatusType GetTaskID (TaskRefType TaskID)`	■		■	■	■	■
`StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)`	■		■	■	■	■
Event Control						
`StatusType SetEvent (TaskType TaskID, EventMaskType Mask)`	■		■	■	■	■
`StatusType ClearEvent (EventMaskType Mask)`	■		■	■	■	■
`StatusType GetEvent (TaskType TaskID, EventMaskRefType Mask)`	■		■	■	■	■
`StatusType WaitEvent (EventMaskType Mask)`	■		■	■	■	■
Interrupt Handling The behavior of the interrupt handling functions is implementation specific. For a detailed description see hardware specific technical reference [4].						
`void EnableAllInterrupts (void)`	■		■	■	■	■
`void DisableAllInterrupts (void)`	■		■	■	■	■
`void ResumeAllInterrupts (void)`	■		■	■	■	■
`void SuspendAllInterrupts (void)`	■		■	■	■	■
`void ResumeOSInterrupts (void)`	■		■	■	■	■
`void SuspendOSInterrupts (void)`	■		■	■	■	■

API Function Prototype	Standard Specification		Scalability Class			
	OSEK	ASR	1	2	3	4
Resource Management The behaviour of the resource management functions is implementation specific						
`StatusType` **`GetResource`** `(ResourceType ResID)`	■		■	■	■	■
`StatusType` **`ReleaseResource`** `(ResourceType ResID)`	■		■	■	■	■
Alarms						
`StatusType` **`GetAlarmBase`** `(AlarmType AlarmID, AlarmBaseRefType Info)`	■		■	■	■	■
`StatusType` **`GetAlarm`** `(AlarmType AlarmID, TickRefType Tick)`	■		■	■	■	■
`StatusType` **`SetRelAlarm`** `(AlarmType AlarmID, TickType Increment, TickType cycle)`	■		■	■	■	■
`StatusType` **`SetAbsAlarm`** `(AlarmType AlarmID, TickType Start, TickType cycle)`	■		■	■	■	■
`StatusType` **`CancelAlarm`** `(AlarmType AlarmID)`	■		■	■	■	■
Execution Control						
`void` **`StartOS`** `(AppModeType Mode)`	■		■	■	■	■
`void` **`ShutdownOS`** `(StatusType Error)`	■		■	■	■	■
`ISRType` **`GetISRID`** `(void)`		■	■	■	■	■
`AppModeType` **`GetActiveApplicationMode`** `(void)`	■		■	■	■	■
`ApplicationType` **`GetApplicationID`** `(void)`		■			■	■
`StatusType` **`TerminateApplication`** `(RestartType RestartOption)`		■			■	■
`StatusType` **`CallTrustedFunction`** `(TrustedFunctionIndexType FunctionIndex, TrustedFunctionParameterRefType FunctionParams)`		■			■	■
Hook Routines The context for called hook routines is implementation specific. For a detailed description see see hardware specific technical reference [4].						
`void` **`ErrorHook`** `(StatusType Error)`	■					
`void` **`PreTaskHook`** `(void)`	■					
`void` **`PostTaskHook`** `(void)`	■					
`void` **`StartupHook`** `(void)`	■					
`void` **`ShutdownHook`** `(StatusType Error)`	■					
`ProtectionReturnType` **`ProtectionHook`** `(StatusType Fatalerror)`		■		■	■	■
`void` **`StartupHook_<App>`** `(void)`		■			■	■

API Function Prototype	Standard Specification		Scalability Class			
	OSEK	ASR	1	2	3	4
void **ErrorHook_<App>** (StatusType Error)		■			■	■
void **ShutdownHook_<App>** (StatusType Fatalerror)		■			■	■
System Control						
StatusType **StartCOM** (void)	■		■	■	■	■
StatusType **StopCOM** (Scalar ShutdownMode)	■		■	■	■	■
StatusType **MessageInit** (void)	■		■	■	■	■
Message Control						
StatusType **SendMessage** (SymbolicName Message, AccessNameRef Data)	■		■	■	■	■
StatusType **ReceiveMessage** (SymbolicName Message, AccessNameRef Data)	■		■	■	■	■
StatusType **GetMessageResource** (SymbolicName Message)	■		■	■	■	■
StatusType **ReleaseMessageResource** (SymbolicName Message)	■		■	■	■	■
StatusType **GetMessageStatus** (SymbolicName Message)	■		■	■	■	■
Schedule Tables						
StatusType **StartScheduleTableRel** (ScheduleTableType ScheduleTableID, TickType Offset)		■	■	■	■	■
StatusType **StartScheduleTableAbs** (ScheduleTableType ScheduleTableID, TickType Start)		■	■	■	■	■
StatusType **StopScheduleTable** (ScheduleTableType ScheduleTableID)		■	■	■	■	■
StatusType **NextScheduleTable** (ScheduleTableType ScheduleTableID_From, ScheduleTableType ScheduleTableID_To)		■	■	■	■	■
StatusType **StartScheduleTableSynchron** (ScheduleTableType ScheduleTableID)		■		■		■
StatusType **SyncScheduleTable** (ScheduleTableType ScheduleTableID, TickType Value)		■		■		■
StatusType **SetScheduleTableAsync** (ScheduleTableType ScheduleTableID)		■		■		■
StatusType **GetScheduleTableStatus** (ScheduleTableType ScheduleTableID, ScheduleTableStatusRefType ScheduleStatus)		■	■	■	■	■

API Function Prototype	Standard Specification		Scalability Class			
	OSEK	ASR	1	2	3	4
Counters						
`StatusType IncrementCounter (CounterType CounterID)`		■	■	■	■	■
`StatusType GetCounterValue (CounterType CounterID,TickRefType Value)`		■	■	■	■	■
`StatusType GetElapsedCounterValue (CounterType CounterID, TickRefType Value, TickRefType ElapsedValue)`		■	■	■	■	■
Access Rights Management						
`AccessType CheckISRMemoryAccess (ISRType ISRID, MemoryStartAddressType Address, MemorySizeType Size)`		■			■	■
`AccessType CheckTaskMemoryAccess (TaskType TaskID, MemoryStartAddressType Address, MemorySizeType Size)`		■			■	■
`ObjectAccessType CheckObjectAccess (ApplicationType ApplID, ObjectTypeType ObjectType,…)`		■			■	■
`ApplicationType CheckObjectOwnership (ObjectTypeType ObjectType,…)`		■			■	■

Table 6-1 Standard API functions

6.2 API Functions defined by Vector - Overview

This chapter gives an overview of all API functions defined for the OS by Vector. Further chapters contain detailed descriptions of these API functions.

API Function Prototype	Scalability Class			
	1	2	3	4
Measurement API For a detailed description see chapter 6.4.				
`StatusType GetTaskMaxExecutionTime (TaskType TaskID, TimeRefType MaxTime)`		■		■
`StatusType GetISRMaxExecutionTime (ISRType TaskID, TimeRefType MaxTime)`		■		■
`StatusType GetTaskMaxBlockingTime (TaskType TaskID, BlockTypeType BlockType, ResourceType ResourceID, TimeRefType MaxTime)`		■		■

API Function Prototype	Scalability Class			
	1	2	3	4
StatusType **GetISRMaxBlockingTime** (ISRType ISRID, BlockTypeType BlockType, ResourceType ResourceID, TimeRefType MaxTime)		■		■
StatusType **osGetISRMinInterArrivalTime** (ISRType ISRID, osTPTimeStampRefType MinTime)		■		■
StatusType **osGetTaskMinInterArrivalTime** (TaskType ISRID, osTPTimeStampRefType MinTime)		■		■
Hook Routines The context for called hook routines is implementation specific. For a detailed description see see hardware specific technical reference [4].				
void **PreAlarmHook** (void)	■	■	■	■

Table 6-2 Vector API functions

6.3 API Macros

For each API function, four macros are defined:

Macro	Description
OS_APIabbreviation_ENTRY	Called at start of the API function
OS_APIabbreviation_EXIT	Called at exit of the API function
OS_APIabbreviation_START_CRITICAL	Called at start of critical section in API (after disabling interrupts)
OS_APIabbreviation_END_CRITICAL	Called at end of critical section in API (before enabling interrupts)

Table 6-3 API function call macros

For each hook function, two macros are defined:

Macro	Description
OS_Hookabbreviation_ENTRY	Called at start of hook function
OS_Hookabbreviation_EXIT	Called at exit of hook function

Table 6-4 Hook routine macros

Before each call of the dispatcher, the following macro is defined:

Macro	Description
OS_START_DISPATCH	Called at start of dispatcher

Table 6-5 Dispatcher start macro

In the standard configuration, these macros are empty by including the file EMPTYMAC.H. By defining `osdTestMacros` to a value of 1 to 4, the include file TESTMAC1.H - TESTMAC4.H is included for user defined macros. The macro `osdTestMacros` has to be defined as a command line parameter of the C-compiler respectively in the development environment.

The macro values have to be globally defined when calling the compiler.

Caution
Depending on the specific implementation and application, not all macros are called.

Caution
The MICROSAR OS features of ORTI debug support and internal trace are implemented by using the files TESTMAC1.H and TESTMAC2.H, so these features do not work together with user-defined macros in another include file.

6.4 Timing Measurement API

6.4.1 GetTaskMaxExecutionTime

Prototype	
`StatusType` **`GetTaskMaxExecutionTime`** `( TaskType TaskID, TimeRefType MaxTime )`	
Parameter	
TaskID	The task to be questioned
MaxTime	Maximum execution time, measured in all finished time frames.
Return code	
E_OK	No errors
E_OS_ID	The `TaskID` is not valid.
Functional Description	
The maximum execution time of finished executions of the questioned task since StartOS. The value is in ticks of the ExecutionTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	
> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)	

Expected Caller Context
> task or cat2 ISR

Table 6-6 GetTaskMaxExecutionTime

6.4.2 GetISRMaxExecutionTime

Prototype	
`StatusType GetISRMaxExecutionTime ( ISRType TaskID, TimeRefType MaxTime )`	
Parameter	
TaskID	The task to be questioned
MaxTime	Maximum execution time of the respective ISR for all finished ISR activations.
Return code	
E_OK	No errors
E_OS_ID	The `ISRID` is not valid.
Functional Description	
The maximum execution time of finished executions of the questioned ISR since StartOS. The value is in ticks of the ExecutionTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	
> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)	
Expected Caller Context	
> task or cat2 ISR	

Table 6-7 GetISRMaxExecutionTime

6.4.3 GetTaskMaxBlockingTime

Prototype	

`StatusType GetTaskMaxBlockingTime (`
`TaskType TaskID,`
`BlockTypeType BlockType,`
`ResourceType ResourceID,`
`TimeRefType MaxTime )`	

Parameter	
TaskID	The task to be questioned
BlockType	`OS_ALL_INTERRUPTS`, `OS_OS_INTERRUPTS` or `OS_RESOURCE`
ResourceID	If `BlockType` == `OS_RESOURCE`, ResourceID specifies the Resource

MaxTime	Maximum of all measured times.
Return code	
E_OK	No errors
E_OS_ID	The `TaskID`, the `BlockType` or the `ResourceID` are invalid.
Functional Description	
The maximum blocking time of finished locking sequences of the questioned task and the resource or interrupt lock type since StartOS. The value is in ticks of the BlockingTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	
> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)	
Expected Caller Context	
> task or cat2 ISR	

Table 6-8 GetTaskMaxBlockingTime

6.4.4 GetISRMaxBlockingTime

Prototype	

`StatusType` **`GetISRMaxBlockingTime`** `(`
`ISRType ISRID,`
`BlockTypeType BlockType,`
`ResourceType ResourceID,`
`TimeRefType MaxTime )`	

Parameter	
`ISRID`	The ISR to be questioned
BlockType	`OS_ALL_INTERRUPTS`, `OS_OS_INTERRUPTS` or `OS_RESOURCE`
ResourceID	If `BlockType` == `OS_RESOURCE`, ResourceID specifies the Resource
MaxTime	Maximum of all measured times.
Return code	
E_OK	No errors
E_OS_ID	The `TaskID`, the `BlockType` or the `ResourceID` are invalid.
Functional Description	
The maximum blocking time of finished locking sequences of the questioned ISR and the resource or interrupt lock type since StartOS. The value is in ticks of the BlockingTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	

> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)
Expected Caller Context
> task or cat2 ISR

Table 6-9 GetISRMaxBlockingTime

6.4.5 GetTaskMinInterArrivalTime

Prototype	
`StatusType` **`GetTaskMinInterArrivalTime`**`( TaskType TaskID, osTPTimeStampRefType MinTime )`	
Parameter	
TaskID	The task to be questioned
MinTime	Minimum time between two task arrivals
Return code	
E_OK	No errors
E_OS_ID	The `TaskID` is not valid.
E_OS_ACCESS	No access rights to task (SC4 only)
E_OS_ILLEGAL_ADDRESS	Memory address of `MinTime` not writeable (SC4 only)
Functional Description	
Returns the minimum time span between two arrivals of a task (see [1]) as measured since StartOS. The value is in ticks of the InterArrivalTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	
> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)	

Expected Caller Context
> task or cat2 ISR

Table 6-10 GetTaskMinInterArrivalTime

6.4.6 GetISRMinInterArrivalTime

Prototype	
`StatusType GetISRMinInterArrivalTime ( ISRType IsrID, osTPTimeStampRefType MinTime )`	
Parameter	
IsrID	The ISR to be questioned
MinTime	Minimum time between two ISR arrivals
Return code	
E_OK	No errors
E_OS_ID	The `ISRID` is not valid.
E_OS_ACCESS	No access rights for this ISR (SC4 only)
E_OS_ILLEGAL_ADDRESS	Memory address of `MinTime` not writeable (SC4 only)
Functional Description	
Returns the minimum time span between two arrivals of an ISR (see [1]) as measured since StartOS. The value is in ticks of the InterArrivalTime hardware timer. The number of ticks per ms of this timer is printed into the HTML list file.	
Particularities and Limitations	
> Available in Scalability Classes 2 and 4. > This function is synchronous. > This function is reentrant. > This function is only available if the attribute `TimingMeasurement` is set to `TRUE` (is selected)	
Expected Caller Context	
> task or cat2 ISR	

Table 6-11 GetISRMinInterArrivalTime

6.5 Implementation specific Behavior

The behaviour of the functions listed in this chapter is implementation specific.

6.5.1 Interrupt Handling

In general the usage of the interrupt API functions is allowed before the operating system is started. The affected functions are:

> `DisableAllInterrupts`

> `EnableAllInterrupts`

- `SuspendAllInterrupts`
- `ResumeAllInterrupts`
- `SuspendOSInterrupts`
- `ResumeOSInterrupts`

The implementation specific behaviour is of these functions is described in [4].

6.5.1.1 EnableAllInterrupts

Prototype	
`void EnableAllInterrupts ( void )`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service restores the state saved by `DisableAllInterrupts`. This service is a counterpart of `DisableAllInterrupts` service, which has to be called before, and its aim is the completion of the critical section of code. No API service calls are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. Usually, this service enables recognition of interrupts by the central processing unit. This function might be implemented using a global interrupt flag or an interrupt level register.	
Particularities and Limitations	
> --	
Expected Caller Context	
> The service may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines.	

Table 6-12 EnableAllInterrupts

6.5.1.2 DisableAllInterrupts

Prototype	
`void DisableAllInterrupts ( void )`	
Parameter	
--	--

Return code	
Void	--
Functional Description	
This service disables all interrupts for which the hardware supports disabling. The state before is saved for the `EnableAllInterrupts` call. This service is intended to start a critical section of the code. This section shall be finished by calling the `EnableAllInterrupts` service. No API service calls are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. Usually, this service disables recognition of interrupts by the central processing unit. Note that this service does not support nesting. If nesting is needed for critical sections e.g. for libraries `SuspendOSInterrupts`/`ResumeOSInterrupts` or `SuspendAllInterrupt`/`ResumeAllInterrupts` should be used. This function might be implemented using a global interrupt flag or an interrupt level register.	
Particularities and Limitations	
> --	
Expected Caller Context	
> The service may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines.	

Table 6-13 DisableAllInterrupts

6.5.1.3 ResumeAllInterrupts

Prototype	
`void ResumeAllInterrupts ( void )`	
Parameter	
--	--
Return code	
void	--

Functional Description
This service restores the recognition status of all interrupts saved by the `SuspendAllInterrupts` service. This service is the counterpart of `SuspendAllInterrupts` service, which has to have been called before, and its aim is the completion of the critical section of code. No API service calls beside `SuspendAllInterrupts`/`ResumeAllInterrupts` pairs and `SuspendOSInterrupts`/`ResumeOSInterrupts` pairs are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. `SuspendAllInterrupts`/`ResumeAllInterrupts` can be nested. In case of nesting pairs of the calls `SuspendAllInterrupts` and `ResumeAllInterrupts` the interrupt recognition status saved by the first call of `SuspendAllInterrupts` is restored by the last call of the `ResumeAllInterrupts` service. This function might be implemented using a global interrupt flag or an interrupt level register.
Particularities and Limitations
> --
Expected Caller Context
> The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from all hook routines.

Table 6-14 ResumeAllInterrupts

6.5.1.4 SuspendAllInterrupts

Prototype	
`void SuspendAllInterrupts ( void )`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling. This service is intended to protect a critical section of code from interruptions of any kind. This section shall be finished by calling the `ResumeAllInterrupts` service. No API service calls beside `SuspendAllInterrupts`/`ResumeAllInterrupts` pairs and `SuspendOSInterrupts`/`ResumeOSInterrupts` pairs are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. This function might be implemented using a global interrupt flag or an interrupt level register.	

Particularities and Limitations
> --
Expected Caller Context
> The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from all hook routines

Table 6-15 SuspendAllInterrupts

6.5.1.5 ResumeOSInterrupts

Prototype	
`void ResumeOSInterrupts ( void )`	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service restores the recognition status of interrupts saved by the `SuspendOSInterrupts` service. This service is the counterpart of `SuspendOSInterrupts` service, which has to have been called before, and its aim is the completion of the critical section of code. No API service calls beside `SuspendAllInterrupts`/`ResumeAllInterrupts` pairs and `SuspendOSInterrupts`/`ResumeOSInterrupts` pairs are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. `SuspendOSInterrupts`/`ResumeOSInterrupts` can be nested. In case of nesting pairs of the calls `SuspendOSInterrupts` and `ResumeOSInterrupts` the interrupt recognition status saved by the first call of `SuspendOSInterrupts` is restored by the last call of the `ResumeOSInterrupts` service. This function might be implemented using a global interrupt flag or an interrupt level register	
Particularities and Limitations	
> --	
Expected Caller Context	
> The service may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines.	

Table 6-16 ResumeOSInterrupts

6.5.1.6 SuspendOSInterrupts

Prototype	
`void SuspendOSInterrupts ( void )`	
Parameter	
--	--

Return code	
void	--
Functional Description	
This service saves the recognition status of interrupts of category 2 and disables the recognition of these interrupts. This service is intended to protect a critical section of code. This section shall be finished by calling the `ResumeOSInterrupts` service. No API service calls beside `SuspendAllInterrupts`/`ResumeAllInterrupts` pairs and `SuspendOSInterrupts`/`ResumeOSInterrupts` pairs are allowed within this critical section. The implementation should adapt this service to the target hardware providing a minimum overhead. It is intended only to disable interrupts of category 2. However, if this is not possible in an efficient way more interrupts may be disabled. This function might be implemented using a global interrupt flag or an interrupt level register.	
Particularities and Limitations	
> --	
Expected Caller Context	
> The service may be called from an ISR and from the task level, but not from hook routines.	

Table 6-17 SuspendOSInterrupts

6.5.2 Resource Management

The affected functions are:

> GetResource

> ReleaseResource

The implementation specific behaviour is of these functions is described in [4].

6.5.2.1 GetResource

Prototype	
`StatusType GetResource ( ResourceType ResID )`	
Parameter	
`ResID`	Reference to resource
Return code	
`E_OK`	No error
`E_OS_ID`	Resource `ResID` is invalid
`E_OS_ACCESS`	Attempt to get a resource which is already occupied by any task or ISR, or the statically assigned priority of the calling task or interrupt routine is higher than the calculated ceiling priority.

Functional Description
This call serves to enter critical sections in the code that are assigned to the resource referenced by `ResID`. A critical section shall always be left using `ReleaseResource`. Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section (strictly stacked, Restrictions when using resources). Nested occupation of one and the same resource is also forbidden! It is recommended that corresponding calls to `GetResource` and `ReleaseResource` appear within the same function. It is not allowed to use services which are points of rescheduling for non preemptable tasks (`TerminateTask`, `ChainTask`, `Schedule` and `WaitEvent`) in critical sections. Additionally, critical sections are to be left before completion of an interrupt service routine. Generally speaking, critical sections should be short. The service may be called from an ISR and from task level. Depending on the possibility to manipulate interrupt levels, this function may be used on interrupt level or not and may be implemented differently. If used on task level, the behavior and functionality is always the same (according to the specification).
Particularities and Limitations
> --
Expected Caller Context
> Task level or cat2 ISR

Table 6-18 GetResource

6.5.2.2 ReleaseResource

Prototype	
`StatusType ReleaseResource ( ResourceType ResID )`	
Parameter	
`ResID`	Reference to resource
Return code	
`E_OK`	No error
`E_OS_ID`	Resource `ResID` is invalid
`E_OS_NOFUNC`	Attempt to release a resource which is not occupied by any task or ISR, or another resource shall be released before.
`E_OS_ACCESS`	Attempt to release a resource which has a lower ceiling priority than the statically assigned priority of the calling task or interrupt routine.

Functional Description	
ReleaseResource is the counterpart of `GetResource` and serves to leave critical sections in the code that are assigned to the resource referenced by `ResID`. For information on nesting conditions, see particularities of `GetResource`. The service may be called from an ISR and from task level. Depending on the possibility to manipulate interrupt levels, this function may be used on interrupt level or not and may be implemented differently. If used on task level, the behavior and functionality is always the same (according to the specification).	
Particularities and Limitations	
> --	
Expected Caller Context	
> Task level or cat2 ISR	

Table 6-19 ReleaseResource

6.5.3 Execution Control

The affected functions are:

- StartOS
- ShutdownOS

The implementation specific behaviour is of these functions is described in [4].

6.5.3.1 StartOS

Prototype	
`void StartOS ( AppModeType Mode )`	
Parameter	
`Mode`	application mode
Return code	
`void`	--
Functional Description	
The user can call this system service to start the operating system in a specific mode. Only allowed outside of the operating system, therefore implementation specific restrictions may apply. After calling `StartOS` the program never returns to the call level of `StartOS`.	
Particularities and Limitations	
> --	

Expected Caller Context
> C main function

Table 6-20 StartOS

6.5.3.2 ShutdownOS

Prototype	
`void ShutdownOS ( StatusType Error )`	
Parameter	
`Error`	error occurred
Return code	
`void`	--
Functional Description	
The user can call this system service to abort the overall system (e.g. emergency off). The operating system also calls this function internally, if it has reached an undefined internal state and is no longer ready to run. If a `ShutdownHook` is configured the hook routine `ShutdownHook` is always called (with `Error` as argument) before shutting down the operating system. If `ShutdownHook` returns, further behaviour of `ShutdownOS` is implementation specific. In case of a system where OSEK OS and OSEKtime OS coexist, `ShutdownHook` has to return. `Error` needs to be a valid error code supported by OSEK OS. In case of a system where OSEK OS and OSEKtime OS coexist, `Error` might also be a value accepted by OSEKtime OS. In this case, if enabled by an OSEKtime configuration parameter, OSEKtime OS will be shut down after OSEK OS shutdown. After this service the operating system is shut down. Allowed at task level, ISR level, in `ErrorHook` and `StartupHook`, and also called internally by the operating system. If the operating system calls `ShutdownOS` it never uses `E_OK` as the passed parameter value. After the call of `ShutdownHook` MICROSAR OS disables all interrupts and will never return to the call level. The `ShutdownHook` is called with disabled interrupts.	
Particularities and Limitations	
> --	

Expected Caller Context
> --

Table 6-21 ShutdownOS

6.6 Hook Routines

This chapter describes the prototypes of the called hook routines. The context of the called hook routines is implementation specific and described in [4].

All hook routines defined in the OSEK specification are called and, if enabled, must be defined by the user. The hook routines are called with interrupts disabled (if not stated otherwise).

6.6.1 StartupHook

Prototypes	
`void StartupHook ( void ) /* general startup hook */` `void StartupHook_<App> ( void ) /* application specific startup hook */`	
Parameter	
--	--
Return code	
void	--
Functional Description	
The user may call the initialization routines for hardware drivers.	
Particularities and Limitations	
> --	
Call Context	
> interrupt or task context > The `StartupHook` routine is called while the operating system is initialized.	

Table 6-22 StartupHook

6.6.2 PreTaskHook

Prototype	
`void PreTaskHook ( void )`	
Parameter	
--	--
Return code	
void	--
Functional Description	
The user can use the API function `GetTaskID` to determine the new task.	

Particularities and Limitations
> --
Call Context
> interrupt or task context > `PreTaskHook` is called after a task is set into the RUNNING state (not into the READY state). > For particularities of using PreTaskHook when using timing protection, please see [4]

Table 6-23 PreTaskHook

6.6.3 PostTaskHook

Prototype	
`void` **`PostTaskHook`** `( void )`	
Parameter	
--	--
Return code	
void	--
Functional Description	
The user can use the API function `GetTaskID` to determine the currently left task.	
Particularities and Limitations	
> --	
Call Context	
> interrupt or task context > `PostTaskHook` is called before a task is taken out of the RUNNING state. > For particularities of using the PostTaskHook when using timing protection, please see [4]	

Table 6-24 PostTaskHook

6.6.4 ErrorHook

Prototype	
`void` **`ErrorHook`** `( StatusType ErrorCode ) /* general error hook */` `void` **`ErrorHook_<App>`** `( StatusType ErrorCode ) /* appl. spec. error hook */`	
Parameter	
ErrorCode	Error code of API which detected the error and called the error hook
Return code	
void	--

Functional Description
The user may use the error number parameter to decide how to react on the error. Additional error information is available in the error hook if the attributes `USEGETSERVICEID` and `USEPARAMETERACCESS` are set to `TRUE`. This information can be accessed by access macros; for details refer to the OSEK specification [3]. All possible access macros are supported by MICROSAR OS. If `EXTENDED_STATUS` is enabled and `ErrorInfoLevel` is set to `Modulnames`, additional error information is available in the `ErrorHook`. The variable `osActiveTaskModule` is a pointer to the module name and the variable `osActiveTaskLineNumber` is the line number in the C module where the API function was called. Inspecting these two variables allows the user to locate the source code which caused the error message.
Particularities and Limitations
> --
Call Context
> interrupt or task context > `ErrorHook` is called every time an API function is called with wrong parameters or if the system detects an error (e.g. stack overflow).

Table 6-25 ErrorHook

6.6.5 ShutdownHook

Prototype	
`void ShutdownHook ( StatusType ErrorCode ) /* general shutdown hook */` `void ShutdownHook_<App> (StatusType ErrorCode) /* appl. spec. shutdown hook */`	
Parameter	
ErrorCode	Error code of API which detected the error and called the shutdown hook or the parameter which was passed to ShutdownOS.
Return code	
void	--
Functional Description	
The ShutdownHook is called by ShutdownOS	
Particularities and Limitations	
> --	
Call Context	
> interrupt or task context > The system calls the `ShutdownHook` routine if the function `ShutdownOS` was called.	

Table 6-26 ShutdownHook

6.6.6 ProtectionHook

Prototype	
`ProtectionReturnType **ProtectionHook** ( StatusType Fatalerror )`	
Parameter	
Fatalerror	depending on the detected protection error
Return code	
ProtectionReturnType	The return value determines the strategy of further operation
Functional Description	
called on occurrence of a protection error. The application code has to decide about the recovery strategy and pass an appropriate return value to the OS	
Particularities and Limitations	
> In the scalability class SC1, no call of the `ProtectionHook` is supported.	
Call Context	
> interrupt or task context > The `ProtectionHook` is called if a TimingProtection failure (SC2, SC4), a memory protection failure (SC3, SC4), or processor exception (e.g. division by zero, illegal instruction etc.) is detected by MICROSAR OS.	

Table 6-27 ProtectionHook

6.6.7 UserPreISRHook

Prototype	
`Void **UserPreISRHook** ( ISRType isr )`	
Parameter	
Isr	The identifier of the ISR that is about to be entered
Return code	
Void	--
Functional Description	
Called just before entering an ISR routine of a category 2 interrupt. This Hook is intended to be used as a development aid. For example, it may be used to measure interrupt run times. This Hook is only available if the attribute CallISRHooks is set to `TRUE`	
Particularities and Limitations	
> Only API functions that are allowed in cat2 ISRs are allowed to be called in the UserPreISRHook.	
Call Context	
> The UserPreISRHook runs in the exact same context as the ISR that is executed afterwards. This includes settings for interrupt nesting, timing protection, timing measurement and memory protection. > All OS API functions, incl. GetISRID(), GetApplicationID(), CheckObjectAccess() etc, work just as if called from within the ISR	

Table 6-28 UserPreISRHook

6.6.8 UserPostISRHook

Prototype	
`Void UserPostISRHook ( ISRType isr )`	
Parameter	
Isr	The identifier of the ISR that was just left
Return code	
Void	--
Functional Description	
Called just after leaving an ISR routine of a category 2 interrupt. This Hook is intended to be used as a development aid. For example, it may be used to measure interrupt run times. This Hook is only available if the attribute CallISRHooks is set to `TRUE`	
Particularities and Limitations	
The UserPostISRHook is called only after a regular return from the ISR routine. In particular: > If an ISR is interrupted by a higher priority ISR, the UserPostISRHook is not called before entering the new ISR. > It an ISR is killed, the UserPostISRHook is not called. > Only API functions that are allowed in cat2 ISRs are allowed to be called in the UserPostISRHook.	
Call Context	
> The UserPreISRHook runs in the exact same context as the ISR that was just executed. This includes settings for interrupt nesting, timing protection, timing measurement and memory protection. > All OS API functions, incl. GetISRID(), GetApplicationID(), CheckObjectAccess() etc, work just as if called from within the ISR	

Table 6-29 UserPostISRHook

6.6.9 PreAlarmHook

Prototype	
`void PreAlarmHook (void)`	
Parameter	
void	--
Return code	
void	--
Functional Description	
Called in the system timer ISR just before the alarm handling of the OS. This Hook is only available if the attribute PreAlarmHook is set to `TRUE`	
Particularities and Limitations	
> Only API functions that are allowed in cat2 ISRs are allowed to be called in the PreAlarmHook. > The execution time of the PreAlarmHook is not considered by the timing protection.	

Call Context
> The PreAlarmHook runs in the same context as the system timer ISR. If an owner application is configured, the system timer ISR and the PreAlarmHook is executed with the application rights of this owner application. > Interrupts of category 2 are disabled during the execution of the PreAlarmHook.

Table 6-30 PreAlarmHook

7 Configuration

Since AUTOSAR OS 3.0.0 specification, XML is used to define and describe an OS configuration. OIL is only intended to act as an intermediate description language, or if the OS shall be used stand-alone without any other AUTOSAR software modules.

All OSEK objects and their attributes have to be defined by one of these description languages.

7.1 Configuration and generation process

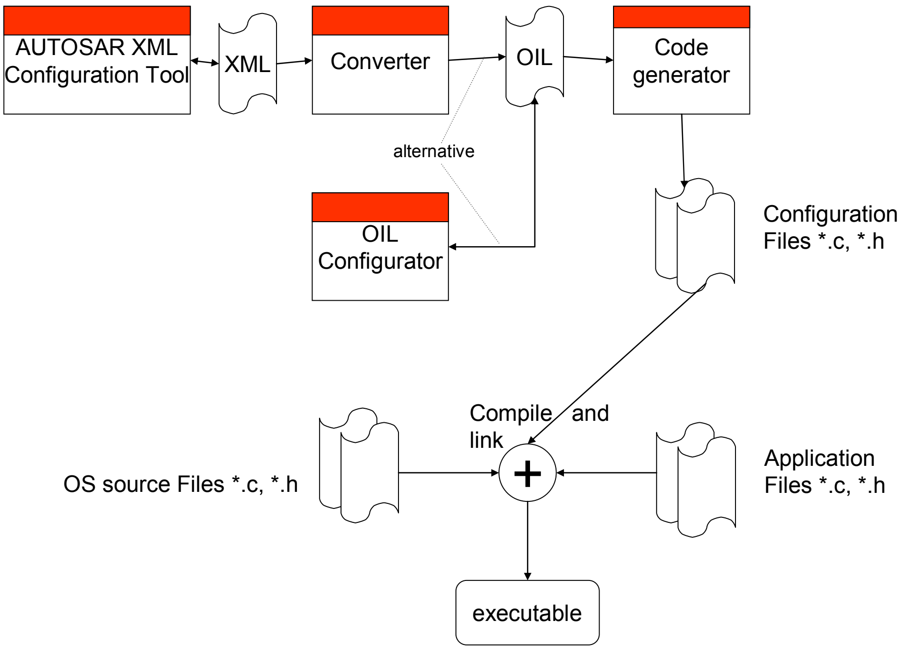

Figure 7-1 System overview of software parts

The figure above shows the complete configuration process of a MICROSAR OS. First all OS objects have to be defined. This can be done either by an AUTOSAR XML configuration tool or by the OIL Configurator (in OIL). If the configuration is done in XML then the XML file has to be converted to OIL.

An OIL file is the base of the code generation process. After code generation all files (OS source files, application files and generated OS configuration files) have to be compiled and linked to an executable.

7.1.1 XML Configuration

A configuration which is based on XML must conform to the AUTOSAR XML schema [8].

To edit a MICROSAR OS XML configuration the DaVinci Configurator Pro or Base of Vector Informatik GmbH can be used. Other tools which are able to edit AUTOSAR configurations may also be used.

After finishing the XML configuration it has to be converted to OIL. The result file has to be passed to the code generator to generate the configuration files.

7.1.2 OIL Configurator

The OIL specification is based on the document "OIL: OSEK Implementation Language – Version: 2.3" (ref. [6]). Additional Attributes are defined by Vector Informatik GmbH; the resulting version of OIL is 3.2.

The OIL Configurator is a Windows based program which is used to configure an OSEK application. The OIL Configurator reads and writes OIL files (OSEK Implementation Language). The usage of the OIL Configurator is described in the online help of the OIL Configurator.

The OIL Configurator has separate property tabs for each OSEK object type. Each object has several standard attributes which are defined in the OIL specification. Additional attributes which are implementation specific are described in the hardware specific document [4].

7.2 Configuration Variants

The OS supports the configuration variants

> `VARIANT-PRE-COMPILE`

The MICROSAR OS system is typically delivered with the source code. The kernel is implemented in several optimized variants which are enabled from the OIL Configurator using C defines. The source code of the operating system has to be compiled if the configuration has changed. For some implementations a library version of the operating system is also supplied. For different configurations different libraries have to be linked to the application.

The configuration classes of the OS parameters depend on the supported configuration variants. For their definitions please see the `OS_<platform and derivate>_bswmd.arxml` file.

7.3 Configuration of the XML / OIL Attributes

Some of the attributes of an OSEK object are standard for all OSEK implementations, and some are specific for each implementation.

This chapter describes the attributes the user can set for each OSEK object. Please note that setting an attribute to `TRUE` is used as a synonym for selecting it and setting to `FALSE` is used as a synonym for deselecting it. The reason is that a selection in the OIL Configurator corresponds to setting the attribute to `TRUE` in the OIL file (this can be checked by opening the OIL file with a normal text editor).

Caution
If a library version of the operating system is used, some attributes or attribute values are not available or predefined.

Some OS objects may have additional attributes to control the input of the TimingAnalyzer tool. The TimingAnalyzer is a tool for offline WCET analysis which is shipped with the MICROSAR OS installation. Those attributes are listed in the chapters of the corresponding OS objects.

7.3.1 OS

The OS object can only be defined once. The OS object controls general aspects of the operating system.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	n.a.	--	> OIL: Freely selectable name, not used by the code generator. > XML: Not available in XML
Comment	n.a.	--	Any comment.
CC	OsOSCC	BCC1, BCC2, ECC1, ECC2, **AUTO**	Conformance class.
STATUS	OsStatus	STANDARD, EXTENDED	Level for status (error) messages.
SCALABILITY CLASS	OsScalabilityClass	SC1, SC2, SC3, SC4, **AUTO**.	Scalability class.
SCHEDULE	OsOSSchedule	NON, FULL, MIXED, **AUTO**	Scheduling policy.
n.a.	OsHooks	***hook routines s stated below (as Booleans)***	> XML: Used as a container to store hook routine information. > OIL: not available
STARTUPHOOK	OsStartupHook	TRUE, **FALSE**	Selects if the hook routine `StartupHook` is called at system startup. > XML: This attribute is placed in container `OsHooks`
ERRORHOOK	OsErrorHook	TRUE, **FALSE**	Selects if the hook routine `ErrorHook` is called if an error occurs. > XML: This attribute is placed in container `OsHooks`
SHUTDOWNHOOK	OsShutdownHook	TRUE, **FALSE**	Selects if the hook routine `ShutdownHook` is called at system shutdown. > XML: This attribute is placed in container `OsHooks`
PRETASKHOOK	OsPreTaskHook	TRUE,	Selects if the hook routine `PreTaskHook` is

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
		FALSE	called at every task switch. > XML: This attribute is placed in container `OsHooks`
POSTTASKHOOK	OsPostTaskHook	TRUE, **FALSE**	Selects if the hook routine `PostTaskHook` is called at every task switch. > XML: This attribute is placed in container `OsHooks`
PROTECTIONHOOK	OsProtectionHook	TRUE, **FALSE**	Selects if the hook routine `ProtectionHook` is called when a protection error is detected. This can only be selected in scalability classes SC2, SC3, and SC4. > XML: This attribute is placed in container `OsHooks`
CallISRHooks	OsOSCallISRHooks	TRUE, **FALSE**	Selects whether the routines UserPreISRHook and UserPostISRHook are called at the entry/exit of each ISR execution.
USEGET SERVICEID	OsUseGetServiceId	TRUE, **FALSE**	Enables the use of access macros for the service ID information in the error hook.
USEPARAMETER-ACCESS	OsUseParameterAccess	TRUE, **FALSE**	Enables the use of access macros for the context related information in the error hook.
USERES SCHEDULER	OsUseRes Scheduler	**TRUE**, FALSE	If deselected, the OS can perform certain optimizations. It needs to be selected if the OSEK resource `RES_SCHEDULER` is used in the application. It is always safe to have this attribute selected.
STACK MONITORING	OsStackMonitoring	TRUE, **FALSE**	If selected, a stack check is performed with each task switch. This attribute is only available if the implementation supports stacks. (In osCAN OSEK implementations this feature was selected by the attribute `WithStackCheck`, which is now obsolete) See also chapter 3.2.2.3 for details.
StackUsageMeasurement	OsStackUsageMeasurement	TRUE, FALSE, **AUTO**	If selected, the stacks are filled with an indicator value during StartOS. This allows measuring the stack usage of tasks and ISRs. See also chapter 3.2.2.5. If AUTO is selected StackUsageMeasurement uses the same setting as STACKMONITORING.
UseGeneratedFastAlarm	OsOSUse GeneratedFast Alarm	TRUE, FALSE	If selected, code is generated for each alarm; if deselected, alarms are handled with sorted lists. Enabling generated alarms leads to a better performance if only few alarms are used.
ErrorInfoLevel	OsOSErrorInfo Level	STANDARD, Modulnames	If set to `STANDARD`, the operating system will report standard OSEK error codes and additional unique error numbers. If set to `Modulenames`, additional information about the

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
			error location will be reported. Setting to `Modulenames` increases ROM size.
OSInternalChecks	OsOSInternal Checks	STANDARD, ADDITIONAL	If set to `STANDARD`, the operating system will perform standard OSEK error checking. If set to Additional, some additional checks will be performed. Setting to Additional will increase the execution time of API functions.
Compiler	OsOSCompiler	*Implementation specific*	The compiler can be chosen. If there is only one compiler this attribute is also set by default.
TickTime	OsOSTickTime		Duration of the system tick in µs.
ORTIDebug Support	OsOSORTIDebug Support	TRUE, **FALSE**	The OS generator produces an ORTI file when this attribute/parameter is set. The attribute/parameter is only available on implementations supporting ORTI.
ORTIDebugLevel	OsOSORTIDebug Level	**ORTI_20**, ORTI_21_Standard, ORTI_21_Additional ORTI_22_Standard, ORTI_22_Additional	This attribute/parameter provides the possibility to define an ORTI standard. The OS generator produces an ORTI file in accordance to the selected version of the standard. The OS supports the versions 2.1 and 2.2 of the standard in two variants. The “Standard” variant supports only those ORTI features, which do not cost overhead in runtime and memory consumption. The “Additional” variant also supports ORTI features with runtime and memory overhead. For the version 2.0 of the ORTI standard, the OS always provides the maximum information regardless of the overhead. The attribute/parameter is available only on implementations with ORTI support. Some implementations only support a subset of the selection alternatives.
StackModel	OsOSStackModel	**STANDARD**, SingleStackOptimized, SingleStackOptimizedCS, SingleStackOsek	Defines how the OS handles stacks. > STANDARD: each task is run on its firmly defined task stack (that is either an own stack or under certain conditions the stack of another task). > SingleStackOptimized: all tasks are executed on the SystemStack. Extended tasks are not allowed. Usage of the TerminateTask API in subfunctions/libraries is also not allowed. Usage of the ChainTask API is not allowed generally. **These restrictions are nonconform to the OSEK standard!** > SingleStackOptimizedCS: all tasks are run on the stack that was setup by the

Attribute Name		Values The default value is written in bold	Description
OIL	XML		
			compiler/startup code. The OS does not generate any stack. Extended tasks are not allowed.Usage of the TerminateTask API in subfunctions/libraries is also not allowed. Usage of the ChainTask API is not allowed at all. Stack monitoring and stack usage measurement is not available. **These restrictions are nonconform to the OSEK standard!** > SingleStackOsek: all basic tasks are run on the SystemStack (comparable to nested ISRs) while any extended tasks use standard stack handling. Each specific OS implementation may support a subset of the possible values. If only the stack model STANDARD is supported the attribute is not available. See chapter 3.2.2.6 for details.
APIOptimization	OsOSAPI Optimization	automatic, manuell	> Automatic: The on / off switching of API functions is done automatically in dependency of the OS configuration. > manual: The API functions may be switched on / off by user directly. See chapter 7.3.1.1 for information about the sub-attributes.
InternalTrace	OsOSInternal Trace	TRUE, **FALSE**	Switches the usage of the Internal trace feature of MICROSAR OS on or off. See chapter 7.3.1.2 for information about the sub-attributes.
ProtectionHook Reaction	OsOSProtection HookReaction	**ALL,** KILLTASKISR, KILLAPPL, KILLAPPL_RESTART, SHUTDOWN	Defines the reaction if the protection hook is called. See chapter 7.3.1.3 for information about the sub-attributes.
Timing Measurement	OsOSTiming Measurement	**TRUE,** FALSE	Switches the Timing measurement feature of MICROSAR OS on or off. If this attribute is set to `FALSE` and timing measurement is selected for a task or ISR, that setting is ignored with a warning. See chapter 7.3.1.4 for information about the sub-attributes. Chapter 3.2.4.2 provides more detailed information about configuration of timing measurement.
TypeHeader Include	OsOSTypeHeader Include	TRUE, **FALSE**	If selected, the AUTOSAR type headers are included in the file os_cfg.h. This is included in the file os.h, which has to be included in all

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
			source files that use API functions of MICROSAR OS OSEK/AUTOSAR. The AUTOSAR type headers are not necessary for the usage of MICROSAR OS OSEK/AUTOSAR, so it is safe to deselect this attribute.
EnumeratedUnhandledISRs	OsOSEnumeratedUnhandledISRs	TRUE, **FALSE**	Determines the handling of unassigned interrupt sources. The default of this attribute is `FALSE`. `FALSE`: This is the normal handling for unassigned interrupt sources. If no interrupt service routine is defined in OIL for an interrupt source the corresponding interrupt vector will be directed to one common unhandled exception handler. This is the recommended setting for the final application software. `TRUE`: During application development there may be interrupts issued by unassigned interrupt sources. In such case it could be a big effort to determine the interrupt source. If this attribute is set to `TRUE` the interrupt vector of each unassigned interrupt source will be directed to a unique unhandled exception routine. If an unhandled exception occurs in that case, the interrupt source which causes this exception can easily be determined by the variable "osISRUnhandledException_Number". The corresponding interrupt source can be distinguished by having a look into the interrupt vector table which normally is generated to intvect.c. *This feature is optional. Please refer to /osCAN_HW/ to find out whether a specific implementation of osCAN supports this feature.*
ConditionalGenerating	/MICROSAR/Board/BoardGeneral/BoardConditionalGenerating	TRUE, **FALSE**	Determines whether the OS code generator creates the files only if the relevant configuration has been modified since the last generator run (ConditionalGenerating = TRUE). If ConditionalGenerating = FALSE, the OS files are always generated. For details about this attribute see Chapter 8.1.3.
PreAlarmHook	OsOSPreAlarmHook	TRUE, **FALSE**	Determins whether the hook function PreAlarmHook shall be called on each system timer interrupt. PreAlarmHook has the following signature: `void PreAlarmHook(void);` and is called before the alarm handling is executed. It uses the ISR context of the system timer ISR. All API calls which are allowed in an

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
			ISR of category 2 are allowed in the PreAlarmHook. Interrupts of category 2 are disabled during the execution of the PreAlarmHook. This attribute is not available on all platforms.

Table 7-1 OS attributes

7.3.1.1 APIOptimization / OSOSAPIOptimization

> If attribute is set to `MANUAL`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
<APIName>	OsOS<APIName>	*List of names of APIs (as Booleans)*	If the API function corresponding to the attribute name is used in the application, the attribute has to be selected.

Table 7-2 Sub-attributes of APIOptimization = Manual

> If attribute is set to `AUTOMATIC`:

If possible, the inclusion of used API functions is done automatically. Example: If no alarm is defined, all API functions for alarm management will be removed from the kernel.

7.3.1.2 InternalTrace / OsOSInternalTrace

> If attribute is set to `TRUE`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TraceDepth	OsOSTraceDepth	-	Size of the trace buffer.
TimeStamp	OsOSTimeStamp		This attribute defines what type of time stamp is used in the trace buffer. If set to None, no time stamp is used. If set to `SystemCounter`, the time stamp is generated from the system tick. If set to `UserDefined`, the time stamp must be provided by the application.

Table 7-3 Sub-attributes of InternalInterface = `TRUE`

> If attribute is set to `FALSE`:

No sub-attributes.

7.3.1.3 ProtectionHookReaction / OsOSProtectionHookReaction

> If attribute is set to `ALL`:

This is the default case. The OS provides all mechanisms for all possible return values from the ProtectionHook. The OS monitors all states and data and can be able, for example, to kill tasks, ISRs or even complete applications.

> If attribute is set to `SELECTED`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
KILLTASKISR	OsOSKILLTASK ISR	TRUE **FALSE**	If selected the return value `PRO_KILLTASKISR` will be handled. All necessary information will be monitored in the OS
KILLAPPL	OsOSKILLAPPL	TRUE **FALSE**	If selected the return value `PRO_KILLAPPL` will be handled. All necessary information will be monitored in the OS
KILLAPPL_ RESTART	OsOSKILLAPPL_ RESTART	TRUE **FALSE**	If selected the return value `PRO_KILLAPPL_RESTART` will be handled. All necessary information will be monitored in the OS
SHUTDOWN	OsOS SHUTDOWN	**TRUE** FALSE	If selected the return value `PRO_SHUTDOWN` will be handled. If for example only `SHUTDOWN` is selected, the OS is more efficient as the monitoring and housekeeping for tasks, ISRs, and applications is switched off.

Table 7-4 Sub-attributes of ProtectionHookReaction = SELECTED

Caution
If the Protection hook returns a value which is not configured by means of the sub-attributes of `ProtectioHookReaktion`, the OS performs a Shutdown.

7.3.1.4 TimingMeasurement / OsOSTimingMeasurement

If this attribute is set to `FALSE` no timing measurement is performed for any task or ISR.

This attribute can be set to `TRUE` in the scalability classes SC2 and SC4 only.

Please see also 3.2.4.2 and 7.3.2.2.

> If this attribute is set to `TRUE`, the subattribute GlobalConfig allows globally overriding of theTask/ISR settings for Timing Protection and Timing Measurement:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
GlobalConfig	OsGlobalConfig	ProtectAndMeasureAll **AsSelected** OnlyMeasureAll	> ProtectAndMeasureAll: The OS provides timing measurement for all tasks and ISRs regardless of their setting in the attribute TIMING_PROTECTION. Timing protection however is provided only for all tasks and ISRs that have the attribute TIMING_PROTECTION set to TRUE. In case the subattribute OnlyMeasure is set to TRUE, that setting is ignored with a warning. In case the attribute TIMING_PROTECTION of a task or ISR is set to FALSE, the OS provides no timing protection. > AsSelected: The os provides timing protection for a task or ISR if that is configured, the attribute OnlyMeasure is honored. > OnlyMeasureAll: The OS does not provide timing protection for any Task or ISR. Instead, it provides timing measurement for all tasks and ISRs. In case a task or ISR is configured to have timing protection and has the subattribute OnlyMeasure set to FALSE, that setting is overridden with a warning.

Table 7-5 Sub-attributes of TimingMeasurement = TRUE

7.3.1.5 PreAlarmHook / OsOSPreAlarmHook

If this attribute is set to TRUE the hook function PreAlarmHook is called druing each system timer interrupt.

If PreAlarmHook is configured it is possible to assign an OwnerApplication by setting AssignOwnerApplication = TRUE.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
AssignOwnerApplication	OsOSAssignOwnerApplication	TRUE, **FALSE**	> TRUE: The subattribute OwnerApplication is available > FALSE: The system timer ISR is not assigned to any application and executed with system rights.
AssignOwnerApplication/	OsOSAssignOwnerApplication/O	-	SystemTimer ISR is assigned

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
OwnerApplication	sOSOwnerApplication		to the given application. The PreAlarmHook is executed with the rights of the configured owner application. **Note:** All Alarms configured for the SystemTimer have to gain access to this application.

Table 7-6 Sub-attributes of `PreAlarmHook` = `TRUE`

Info
PreAlarmHook is not available on all platforms. Refer to the hardware specific manual for details.

7.3.2 Task

In the section Task all tasks and their attributes have to be defined.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the task. This name is used as an argument to all task-related OSEK API functions (e.g. `ActivateTask`). The task function (or task body) has to be defined using the C macro `TASK`() (which appends the suffix `func` to the task name).
Comment	n.a.	--	Any comment.
TYPE	OsTaskTYPE	BASIC, EXTENDED, AUTO	Type of the task: either BASIC or EXTENDED. If set to AUTO, the type is calculated based on the settings for events and the activation count.
SCHEDULE	OsTaskSchedule	NON, FULL	Scheduling policy for this task.
PRIORITY	OsTaskPriority	-	The priority of the task. A higher number represents a higher priority (according to the OSEK specification). The priority may be set with gaps, though the gaps will be eliminated by the code generator. With BCC1 and ECC1 each level may only be assigned to one task. With BCC2 and ECC2 several tasks may be set on the same priority level.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ACTIVATION	OsTaskActivation	-	The number of activations which are recorded in the kernel while the task is possibly running or delayed by higher priority tasks. If `ACTIVATION` is set to a value bigger than 1, no events can be received.
AUTOSTART	OsTaskAutostart	-	If set to `TRUE`, the task will be activated at startup of the operating system. See chapter 7.3.2.1 for details about the sub-attributes.
EVENT	OsTaskEventRef	-	Reference to an event which is used by this task. This attribute can only be used for extended tasks (the attribute `TYPE` might be set to `EXTENDED` or `AUTO`). This attribute can be used multiply if more than one `EVENT` has to be assigned. If events are used with this task, the attribute `ACTIVATION` cannot be bigger than 1.
RESOURCE	OsTaskResource Ref	-	Reference to a resource which is occupied by this task. This attribute can be used multiply if more than one `RESOURCE` shall be assigned.
StackSize	OsTaskStackSize	-	Task stack size in byte. This attribute is only available if the implementation supports configurable task stacks.
NotUsingSchedule	OsTaskNotUsing Schedule	TRUE, **FALSE**	In certain cases stacks may be shared between different tasks. This depends on the usage of the API function `Schedule`. If the application programmer does not use `Schedule`, stacks may be shared. In this case stack sharing may be enabled by the attribute `NotUsingSchedule`. This attribute is only available if the implementation supports stacks. See chapters 3.2.2.4.2 and 3.2.2.4.3 for the details.
TIMING_PROTECTI ON	OsTaskTiming Protection	-	Selects timing protection for the task. See chapter 7.3.2.2 for information about the sub-attributes.
ACCESING_ APPLICATION	OsTaskAccessing Application	-	Defines access rights of an application for this task. This attribute can be defined multiply, so different applications might have access right to the same task. This attribute can be used in scalability classes SC3 and SC4 only.

Table 7-7 Task attributes

7.3.2.1 AUTOSTART / OsTaskAutostart

> OIL: If attribute set to `FALSE`:

No sub-attributes.

> XML: If this container is not present:

AUTOSTART switched off.

> If attribute is set to TRUE:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
APPMODE	OsTaskAppMode Ref	-	Defines an application mode, the task is started in automatically. This attribute might be defined several times to start the task in different application modes.

Table 7-8 Sub-attributes of TASK->AUTOSTART=TRUE

7.3.2.2 TIMING_PROTECTION / OsTaskTimingProtection

Please note that TIMING_PROTECTION = TRUE can only be selected in the scalability classes SC2 and SC4.

> If attribute is set to FALSE:

No sub-attributes.

> If this attribute is not defined in XML:

Timing protection is switched off

The value of this attribute might be overridden by the OS attribute TimingMeasurement, as described in chapters 7.3.1.4 and 3.2.4.2

> If attribute is set to TRUE:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
EXECUTION BUDGET	OsTaskExecution Budget	-	Defines the maximum execution time for the task
TIMEFRAME	OsTaskTimeFrame	-	Defines the minimum time between task activations
MAXOS INTERRUPT LOCKTIME	OsTaskOsInterrupt LockBudget	-	Maximum time OS interrupts are locked (by SuspendOSInterrupts)
MAXALL INTERRUPT LOCKTIME	OsTaskAllInterrupt LockBudget	-	Maximum time ALL interrupts are locked (by SuspendAllInterrupts or DisableAllInterrupts)
LOCKINGTIME = RESOURCELOCK	OsTaskResource Lock	-	Is intended to be a container for sub-attributes concerning the locking time of resources
LOCKINGTIME = RESOURCELOCK/ RESOURCE	Inside the container OsTaskResource Lock:	-	The resource for which the locking time is specified.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
	OsTaskResource LockResourceRef		
LOCKINGTIME = RESOURCELOCK/ RESOURCELOCK TIME	Inside the container OsTaskResource Lock: OsTaskResource LockBudget	-	Maximum time the resource is locked (by `GetResource`)
OnlyMeasure	OsOnlyMeasure	TRUE **FALSE**	If set to `FALSE`, timing values of this task are measured and violations against the configured values lead to a call of the ProtectionHook. If set to `TRUE`, the timing values are still measured but no call of the ProtectionHook occurs.

Table 7-9 Sub-attributes of TASK-> `TIMING_PROTECTION=TRUE`

7.3.2.3 Task attributes concerning the timing analyzer

The following attributes have to be used when working with the timing analyzer tool. They are used as input for this tool.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ComputationTime	OsTask ComputationTime	-	The worst case execution time (in nanoseconds)
Period	OsTaskPeriod	-	The minimum activation period of the task (in nanoseconds)
Deadline	OsTaskDeadline	-	The deadline of the task (in nanoseconds)
PRIORITY	OsTaskPriority	-	Priority of the task
UseResource Occupation	OsTaskUse Resource Occupation	-	If set to `TRUE` the occupation of resources can be taken into consideration by the analysis tool.
UseResource Occupation=TRUE/ Resource	OsTaskUse Resource Occupation =TRUE/OsTask Resource	-	Reference to the resource which is occupied.
UseResource Occupation=TRUE/ OccupationTime	OsTaskUse Resource Occupation =TRUE/OsTask OccupationTime	-	Maximum resource occupation time (in nanoseconds)

Table 7-10 Task attributes concerning the timing analyzer

7.3.3 Counter

The Vector MICROSAR OS implementation always provides at least one counter: the SystemTimer. The resolution of the SystemTimer can be changed by the attribute `TickTime` in the OS object. The attribute values for the SystemTimer are set by the implementation and can not be changed by the user.

New counters can be added in the OIL Configurator.

Attribute Name		Values	Description
OIL	**XML**	The default value is written in bold	
Name	Short-Name	-	Name of the counter. This name is used for the Alarm configuration.
Comment	n.a.		Any comment.
MINCYCLE	OsCounterMin Cycle	-	The `MINCYCLE` attribute specifies the minimum allowed number of ticks for a cyclic alarm linked to the counter.
MAXALLOWED VALUE	OsCounterMax AllowedValue	-	The `MAXALLOWEDVALUE` attribute defines the maximum allowed counter value.
TICKSPERBASE	OsCounterTicks PerBase	-	The `TICKSPERBASE` attribute specifies the number of ticks required to reach a counter specific unit. The interpretation is application specific.
TYPE	OsCounterType	SOFTWARE, HARDWARE	Defines the type of the counter. Possible settings are: `SOFTWARE` or `HARDWARE`. `SOFTWARE` means the counter is incremented by means of the system service IncrementCounter, which has to be called by the application. `HARDWARE` means the counter is incremented by MICROSAR OS internally.
TYPE = HARDWARE/ DRIVER	OsDriver	-	> OIL: This is a sub-attribute of `TYPE` in case that `TYPE` = `HARDWARE`. Possible values are `OSINTERNAL` and `GPT`. In case of `OSINTERNAL` the system counter is altered by the OS. In case of `GPT` the Counter is triggered by AUTOSAR GPT module (this is currently not supported in MICROSAR OS) > XML: This is a separate container of the Counter object. which holds the reference to a GPT channel
n.a.	OsGptChannelRef	-	Sub-attribute of `OSDriverReference` to GPT channel which alters the counter.
TYPE = HARDWARE/ TIMECONSTANTS	OsTimeConstant	-	> OIL: This is a sub-attribute of `TYPE` in case that `TYPE` = `HARDWARE`.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
= TIMECONSTANT			> XML: This is a separate container of Counter object which holds time constants related to the counter.
TIMECONSTANT/ CONSTNAME	OsConstName	-	> OIL: This is a Sub-attribute of `TIMECONSTANT`. Defines the access name to the constant. > XML: This is a sub-attribute of `OsTimeConstant`. Defines the access name to the constant.
TIMECONSTANT/ NS	OsTimeValue	-	> OIL: Time constant value in nanoseconds (integer value) > XML: Time constant value in seconds (float value)
NANOSECONDS PERTICK	OsCounter SecondsPerTick	-	> OIL: Defines the length of one tick of the counter in nanoseconds (this is an integer value) > XML Defines the length of one tick of the counter in seconds (this is a float value)
ACCESING_ APPLICATION	OsCounter Accessing Application	-	Defines access rights of an application for this counter. This attribute can be used multiply, so different applications might have access rights to this counter. This attribute can only be used in scalability classes SC3 and SC4.

Table 7-11 Attributes of COUNTER

7.3.4 Alarm

The action of an alarm has to be defined statically.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the alarm. This name is used as an argument to all alarm related OSEK API functions (e.g. `SetRelAlarm`).
Comment	n.a.	--	Any comment
COUNTER	OsAlarmCounter Ref		Reference to the counter which drives the alarm. This is per default the `SystemTimer`
ACTION	OsAlarmAction	> OIL: SETEVENT, ACTIVATETASK, ALARM CALLBACK-, INCREMENT-COUNTER	See chapter 7.3.4.1 for more information.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
		> XML: Choice container: OsAlarmActivateTask, OsAlarmCallback, OsAlarmIncrementCounter, OsAlarmSetEvent	
AUTOSTART	OsAlarmAutostart	TRUE **FALSE**	> OIL: If set to `TRUE`, the alarm will be activated at startup of the system. > XML: If attribute is present the alarm will be activated at startup of the system. See chapter 7.3.4.2 for more information about the sub-attributes and chapter 3.2.1.3 for more information about static alarms.
ACCESING_APPLICATION	OsAlarmAccessingApplication		Defines access rights of an application for this alarm. This attribute can be used multiply, so different applications might have access rights to this alarm. This attribute can be used in scalability classes SC3 and SC4 only.
n.a.	OsAlarmStaticAlarm		Only used for XML configurations. This attribute is used to define alarms with no autostart as static alarms. If the container `OsAlarmAutostart` exists this container is ignored. See also chapter 3.2.1.3.

Table 7-12 Attributes of ALARM

7.3.4.1 ACTION / OsAlarmAction

> Attribute / ChoiceContainer is set to `ACTIVATETASK` / `OsAlarmActivateTask`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TASK	OsAlarmActivateTaskRef	-	Task to be activated

Table 7-13 Sub-attributes of ACTION = ACTIVATETASK

> Attribute / ChoiceContainer is set to `SETEVENT` / `OsAlarmSetEvent`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TASK	OsAlarmSetEvent TaskRef	-	Task to which the event should be sent
EVENT	OsAlarmSetEvent Ref		Event to be sent to the specified task

Table 7-14 Sub-attributes of ACTION = SETEVENT

> Attribute / ChoiceContainer is set to `ALARMCALLBACK` / `OsAlarmCallback`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ALARMCALLBACK NAME	OsAlarmCallback Name	-	Name of the callback function to be activated

Table 7-15 Sub-attributes of ACTION = ALARMCALLBACK

> Attribute / ChoiceContainer is set to `INCREMENTCOUNTER` / `OsAlarmIncrementCounter`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
COUNTER	OsAlarmIncrement CounterRef	-	Name of the counter to be incremented

Table 7-16 Sub-attributes of ACTION = ALARMCALLBACK

7.3.4.2 AUTOSTART / OsTaskAutostart

> OIL: This attribute can be either `TRUE` or `FALSE`. Depending on the value there may be different sub-attributes.

> XML: This attribute can be present in the configuration or it can be omitted. In case this container is present it has sub-attributes which are described below.

> If `AUTOSTART` is set to `TRUE` (OIL) or if the container is present (XML):

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ALARMTIME	OsAlarmAlarm Time	-	Static alarm time in units specified by the attribute `AlarmUnit`

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TYPE	OsAlarmAutostartType	ABSOLUTE RELATIVE	Valid for autostart-ed alarms: the value corresponds to a call of the API-Functions SetRelAlarm or SetAbsAlarm
CYCLETIME	OsAlarmCycleTime		Static cycle time in units specified by the attribute `AlarmUnit`
APPMODE	OsAlarmAppModeRef		List of application modes where the alarm is started automatically.
AlarmUnit	OsAlarmAlarmUnit	USEC, MSEC, SEC, Ticks	
StaticAlarm	OsAlarmStaticAlarm		A static alarm is used, i.e. the alarm time and cycle time settings cannot be changed at runtime. See also chapter 3.2.1.3.

Table 7-17 Sub-attributes of AUTOSTART = `TRUE`

> If `AUTOSTART` is set to `FALSE` (OIL) it has the following sub-attributes:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
StaticAlarm	--	**-**	A static alarm is used, i.e. the alarm time and cycle time settings cannot be changed at runtime. See also chapter 3.2.1.3.

Table 7-18 Sub-attributes of AUTOSTART = `FALSE`

> If the OIL attribute `StaticAlarm` is set to `TRUE`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
AlarmTime	--	-	Static alarm time in units specified by the attribute `AlarmUnit`
CycleTime	--		Static cycle time in units specified by the attribute `AlarmUnit`
AlarmUnit	--	USEC, MSEC, SEC, Ticks	

Table 7-19 Sub-attributes of AUTOSTART = `FALSE` and StaticAlarm = `TRUE`

To configure a static alarm without autostart, XML configurations need the extra container OsAlarmStaticAlarm. This container is only relevant if the Container OsAlarmAutostart is no present in the configuration. Otherwise, the container is ignored.

Sub-attributes of `OsAlarmStaticAlarm`:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
--	OsAlarmAlarm Time	-	Static alarm time in units specified by the attribute `AlarmUnit`
--	OsAlarmCycle Time		Static cycle time in units specified by the attribute `AlarmUnit`
--	OsAlarmAlarmUnit	USEC, MSEC, SEC, Ticks	

Table 7-20 Sub-attributes of StaticAlarm = `TRUE`

7.3.5 Resource

Resources have to be defined with the following attributes:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the resource. This name is used as an argument to all resource related OSEK API functions (e.g. `GetResource`).
Comment	n.a.	--	Any comment
RESOURCE PROPERTY	OsResource Property	STANDARD, LINKED, INTERNAL	This attribute can take the following values: > `STANDARD`: A normal resource which is not linked to another resource and is not an internal resource. > `LINKED`: A resource which is linked to another resource with the property `STANDARD` or `LINKED`. > `INTERNAL`: An internal resource which cannot be accessed by the application.
ACCESING_ APPLICATION	OsResource Accessing Application		Defines access rights of an application for this resource. This attribute can be used multiply, so different applications might have access rights to this resource. This attribute can only be used in scalability classes SC3 and SC4.
RESOURCE PROPERTY =LINKED / LINKED RESOURCE	OsResourceLinked ResourceRef		> OIL: If the resource property is set to `LINKED` the `LINKEDRESOURCE` attribute holds a reference to a resource. > XML: This attribute holds a reference to a

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
			resource.

Table 7-21 Attributes of RESOURCE

7.3.6 Event

Events in the OSEK operating system are always implemented as bits in bit-fields. The user could use bit-masks like ‘0x0001,’ but to achieve portability between different OSEK implementations, the user should use event names which are mapped by the code generator to defined bits.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the event. This name is used as an argument to all event related OSEK-API-functions (e.g. `SetEvent`).
Comment	n.a.	--	Any comment
MASK	OsEventMask	-	> OIL: `Eventmask` or `AUTO` > XML: If `EventMasks` shall be defined automatically this attribute shall be omitted

Table 7-22 Sub-attributes of EVENT

Caution
If the user selects `AUTO` for the mask, the code generator will search for free bits in the bit mask of the receiving task. It is important to specify each task which receives an event otherwise the code generator will generate wrong bit-masks.

7.3.7 ISR

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the interrupt service routine.
Comment	n.a.	--	Any comment
CATEGORY	OsIsrCategory	-	> OIL: Number of category for the interrupt service routine (1-2) > XML: this attribute can be `CATEGORY_1` or `CATEGORY_2`
RESOURCE	OsIsrResourceRef	-	Reference to a resource which is occupied by this task. This attribute can be used multiply if more than one `RESOURCE` has to be assigned. **Remark**: This attribute is only available if the

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
			implementation supports resource management for ISRs. Details are described in the platform-specific part of this documentation [4].
MESSAGE	OsIsrMessage	-	Reference to a message which is accessed by this ISR.
TIMING_ PROTECTION	OsIsrTiming Protection	-	Selects timing protection for the ISR. See chapter 7.3.7.2 for information about the sub-attributes.
EnableNesting	OsIsrEnable Nesting	-	If set to `TRUE` the OS will call the user ISR in a way that interrupts will be enabled again during user ISR. Thus it is possible that the user ISR can be interrupted by other ISRs.
UseSpecialFunction Name	OsIsrUseSpecial FunctionName	-	Normally the attribute `Name/Short-Name` defines the C-Name of the ISR. Since this name must be unique it wouldn't be possible to map different interrupt Sources to a single ISR. This can be done by this attribute. If this attribute is set to `TRUE` there it is possible to define the function name of the ISR in a separate sub-attribute. These names haven't to be unique.
ACCESSING_ APPLICATION	OsIsrAccessing Application	-	Defines access rights of an application for this ISR. This attribute can be used multiply, so different applications might have access rights to this alarm. This attribute can be used in scalability classes SC3 and SC4 only.

Table 7-23 Attributes of ISR

7.3.7.1 UseSpecialFunctionName / OsIsrUseSpecialFunctionName

If this attribute is set to `TRUE` a function name can be specified which is taken as ISR name instead of the `Name(OIL)/Short-Name(XML)` attribute.

This can be used to map several interrupt sources to one ISR routine.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
FunctionName	OsIsrFunctionName	-	Name of the ISR routine.

Table 7-24 Sub-attributes of UseSpecialFunctionname / OsIsrUseSpecialFunctionName

Example

Given two Interrupts MyISR1 and MyISR2. Both shall trigger the same ISR routine. Activate SpecialFuntionName for MyISR2, and set FunctionName to “MyISR1”. Now, MyISR2 is mapped onto the MyISR1 routine, which is implemented as usual:

```
ISR(MyISR1)
{
   …
}
```

Note

The `ISR()` macro *MUST* be used for the definition of category 2 ISR handlers. If this is not possible, a wrapper using this macro can be used to call the respective ISR handler:

```
ISR(MyISR1) /* “MyISR1” is the configured */
{           /* FunctionName in OIL or XML */
  MyISRHandlerFunction(); /* “MyISRHandlerFunction” is  */
}                         /* the name of the actual ISR */
                          /* handler provided by the    */
                          /* application                */
```

7.3.7.2 TIMING_PROTECTION / OsIsrTimingProtection

> OIL: This attribute has to be set to `TRUE` to switch on timing protection. The sub-attributes are only visible if `TIMING_PROTECTION` is `TRUE`.

> XML: This attribute has to be present to switch on timing protection.

Please note that timing protection can only be switched on in the scalability classes SC2 and SC4.

Attribute Name		Values	Description
OIL	**XML**	The default value is written in bold	
EXECUTIONTIME	OsIsrExecution Budget	-	The parameter contains the maximum allowed execution time of the interrupt. > OIL: the times are given in nanoseconds. > XML: the times are given in seconds.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TIMEFRAME	OsIsrTimeFrame	-	This parameter contains the minimum inter-arrival time between successive interrupts > OIL: the times are given in nanoseconds. > XML: the times are given in seconds.
MAXOSINTERRUPTLOCKTIME	OsIsrOsInterruptLockBudget	-	This parameter contains the maximum time for which the ISR is allowed to lock all Category 2 interrupts (via `SuspendOSInterrupts`()) (in seconds). > OIL: the times are given in nanoseconds. > XML: the times are given in seconds.
MAXALLINTERRUPTLOCKTIME	OsIsrAllInterruptLockBudget	-	This parameter contains the maximum time for which the ISR is allowed to lock all interrupts (via `SuspendAllInterrupts`() or `DisableAllInterrupts`()) > OIL: the times are given in nanoseconds. > XML: the times are given in seconds.
LOCKINGTIME	OsIsrResourceLock	-	> OIL: This is a (empty) list of all lock times, in nanoseconds > XML: This container holds resource lock times, in seconds. Resource lock times are the maximum times an ISR is allowed to hold a resource.
OnlyMeasure	OsOnlyMeasure	TRUE **FALSE**	If set to `FALSE`, timing values of this task are measured and violations against the configured values lead to a call of the ProtectionHook. If set to `TRUE`, the timing values are still measured but no call of the ProtectionHook occurs. The value of this attribute might be overridden by the OS attribute `TimingMeasurement`, as described in chapters 7.3.1.4 and 3.2.4.2.

Table 7-25 Sub-attributes of `TIMING_PROTECTION` / OsIsrTimingProtection

7.3.7.2.1 LOCKINGTIME / OsIsrResourceLock

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
LOCKINGTIME= RESOURCELOCK/ RESOURCELOCKTIME	OsIsrResourceLockBudget	-	The parameter contains the maximum allowed time an ISR is allowed to hold a resource > OIL: the times are given in nanoseconds. > XML: the times are given in seconds.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
LOCKINGTIME= RESOURCELOCK/ RESOURCE	OsIsrResource LockResourceRef		Holds the reference to this resource

Table 7-26 Sub-attributes of LOCKINGTIME / OsIsrResourceLock

7.3.7.3 ISR Attributes concerning the Timing Analyzer

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ComputationTime	OsIsrComputation Time	-	The worst case execution time (in nanoseconds)
Period	OsIsrPeriod		The minimum activation period of the ISR (in nanoseconds)
Deadline	OsIsrDeadline		The deadline of the ISR (in nanoseconds)
AnalysisPriority	OsIsrAnalysis Priority		The `AnalysisPriority` corresponds to the Task attribute `PRIORITY` / `osTaskPriority`. The `AnalysisPriority` is an extension of the priority values from tasks to ISRs, so all ISR priorities must have higher values as all task priorities to get correct analysis results. (Some OS Implementations use an attribute similar to `priority` for the hardware interrupt level. Therefore to the timing analysis an own attribute was introduced).
UseResource Occupation	OsIsrUseResource Occupation		If set to `TRUE` the occupation of resources can be taken into consideration by the analysis tool.
UseResource Occupation= TRUE/Resource	OsIsrUseResource Occupation=TRUE /OsIsrResource		Reference to the resource which is occupied.
UseResource Occupation= TRUE/ OccupationTime	OsIsrUseResource Occupation=TRUE /OsIsrOccupation Time		Maximum resource occupation time (in nanoseconds)

Table 7-27 ISR attributes concerning the timing analyzer

7.3.8 Message

Messages have the following attributes:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the message. This name is used as an argument to all message related OSEK-API functions (e.g. `SendMessage`).
Comment	n.a.	--	Any comment
MESSAGE PROPERTY	OsMessage Property	> OIL (enumeration): SEND_ STATIC_ INTERNAL RECEIVE_ UNQUEUED_INTERNAL RECEIVE_ QUEUED_ INTERNAL > XML (choice container): OsMessageSendStaticInternal OsMessageReceiveUnqueued Internal OsMessageReceiveQueued Internal	
NOTIFICATION	OsMsgNotification	> OIL (enumeration): Asynchronous notification mechanism: NONE ACTIVATETASK SETEVENT CALLBACK FLAG > XML (choice container): OsMsgNone OsMsgActivateTask OsMsgSetEvent OsMsgComCallback OsMsgOsMsgFlag	

Table 7-28 Attributes of Message

Note
In Scalability Classes 3 and 4, the notifications CALLBACK and FLAG are not allowed.

7.3.8.1 MESSAGEPROPERTY / OsMessageProperty

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
MESSAGE PROPERTY= SEND_STATIC_ INTERNAL/ CDATATYPE	OsMsgCDataType	-	Specifies the data type of the sending message
MESSAGE PROPERTY= RECEIVE_ UNQUUED_ INTERNAL/ SENDING MESSAGE	OsMsgSendingMessage	-	Reference to the message which sends this data
MESSAGE PROPERTY= RECEIVE_ UNQUUED_ INTERNAL/ INITIALVALUE	OsMsgInitialValue	-	The initial value of the message. **Caution** This works only if the data type is an integral type and if the value is different than 0. If the value must be initialized to 0, then the startup code must handle this.
MESSAGE PROPERTY= RECEIVE_ QEUED_INTERNAL /SENDING MESSAGE	OsMsgSendingMessage	-	Reference to the message which sends this data
MESSAGE PROPERTY= RECEIVE_QEUED_ INTERNAL/QUÈUE SIZE	OsMsgQueueSize	-	Defines the size of the receive queue.

Table 7-29 Sub-attributes of MESSAGEPROPERTY / OsMessageProperty

7.3.8.2 NOTIFICATION / OsMsgNotification

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
NOTIFICATION= ACTIVATETASK/ TASK	OsMsgTaskRef	-	Reference to the task which shall be activated
NOTIFICATION= SETEVENT/TASK	OsMsgTaskRef	-	Reference to the task for which an event shall be set
NOTIFICATION= SETEVENT/EVENT	OsMsgEventRef	-	Event to be set
NOTIFICATION= CALLBACK/ CALLBACK ROUTINENAME	OsMsgCom CallbackRoutine Name	-	Name of the callback function
NOTIFICATION= CALLBACK/ MESSAGE	OsMsgMessage Ref	-	References to messages which shall be accessible by the callback routine

Table 7-30 Sub-attributes of NOTIFICATION / OsMsgNotification

7.3.9 COM / OsCom

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
COMERRORHOOK	OsComErrorHook	TRUE, **FALSE**	The COM error hook routine is used if the value is set to `TRUE`. The hook routine is not used if the value is set to `FALSE`
COMUSEGET SERVICEID	OsComUseGet ServiceId	-	If selected, the usage of the access macros to the service ID information in the COM error hook is enabled.
COMUSE PARAMETER ACCESS	OsComUse ParameterAccess	-	If selected, the usage of the access macros to the context related information in the COM error hook is enabled.
COMSTART COMEXTENSION	OsComStartCom Extension	-	If set to `TRUE`, the user-supplied function `StartCOMExtension` is called from the OSEK COM function `StartCOM`.
COMAPPMODE	OsComAppMode	-	List of all supported COM application modes.
COMSTATUS	OsComStatus	COMEXTENDED, COMSTANDARD	Extended error checking is done if the value of `COMSTATUS` is set to `COMEXTENDED`. Standard error checking is done if the value of `COMSTATUS` is set to `COMSTANDARD`.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
UseCOM	OsComUseCOM	TRUE, FALSE	Switches on the COM module.

Table 7-31 Attributes of COM

Note
In Scalability Classes 3 and 4, the callback function `StartCOMExtension` is executed with the application rights of the caller of StartCOM.

Note
API functions SendMessage and ReceiveMessage disable interrupts during the access to the message buffers. If TimingProtection is active, the detection of a timing violation is delayed until the copying of the message buffer finished.

7.3.10 NM

The section NM is not used with the current MICROSAR OS implementation.

7.3.11 APPMODE / OsAppMode

Application modes have to be defined with the following attributes:

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the application mode. This name is used as an argument to all related OSEK-API-functions and for the definition of the `AUTOSTART` functionality of tasks and alarms.
Comment	n.a.	--	Any comment
n.a.	OsAppModeId		Internal ID of an `Appmode`. The value of this attribute is ignored by MICROSAR OS.

Table 7-32 Attributes of Appmode / OsAppMode

7.3.12 Application / OsApplication

The object APPLICATION is meant for the usage with scalability classes SC3 and SC4.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	--	Freely selectable name, not used by the code generator.
Comment	n.a.	--	Any comment.
n.a.	OsApplication Hooks	TRUE, FALSE	> OIL: not available > XML: container for switches concerning the hook routines
STARTUPHOOK	OsAppStartup Hook	TRUE, FALSE	If set to `TRUE`, the application specific hook routine `StartupHook_<Name>` is called at system startup (<Name> is the value of `APPLICATION` attribute Name). (XML: is contained in `OsApplicationHooks`)
ERRORHOOK	OsAppErrorHook	TRUE, FALSE	If set to `TRUE`, the application specific hook routine `ErrorHook_<Name>` is called if an error occurs inside this application (<Name> is the value of `APPLICATION` attribute Name). (XML: is contained in `OsApplicationHooks`)
SHUTDOWNHOOK	OsAppShutdown Hook	TRUE, FALSE	If set to `TRUE`, the application specific hook routine `ShutdownHook_<Name>` is called at system shutdown (<Name> is the value of `APPLICATION` attribute Name). (XML: is contained in `OsApplicationHooks`)
TRUSTED	OsTrusted	TRUE, FALSE	> OIL: Defines whether the application is trusted or not. See chapter 7.3.12.1 for information about the sub-attributes. > XML: This is only a boolean which marks the Application as trusted application
HAS_RESTART TASK	n.a.	TRUE, FALSE	> OIL: If this attribute is set to `TRUE` the Application uses a restart task. See chapter 7.3.12.2 for information about the sub-attributes.
n.a.	OsRestartTask	-	> OIL: there is no comparable attribute since the restart task reference is a sub-attribute of `HAS_RESTARTTASK`. > XML: Reference to restart task for this application.
TASK	OsAppTaskRef	-	Reference to all tasks belonging to this application.
ISR	OsAppIsrRef	-	Reference to all ISRs belonging to this application.
ALARM	OsAppAlarmRef	-	Reference to all alarms belonging to this application.
SCHEDULETABLE	OsAppSchedule TableRef	-	Reference to all schedule tables belonging to this application.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
COUNTER	OsAppCounterRef	-	Reference to all counters belonging to this application.
RESOURCE	OsAppResource Ref	-	Reference to all resources belonging to this application.
MESSAGE	OsAppMessage	-	Reference to all messages belonging to this application.
n.a.	OsApplication TrustedFunction	-	Container which is used to define trusted functions.

Table 7-33 Attributes of Application / OsApplication

7.3.12.1 Trusted Functions

- OIL: trusted functions are defined as sub-attributes of `'TRUSTED=TRUE'`.
- XML: there are containers for trusted functions.

The preconditions for editing the sub-attributes in the next table are `TRUSTED=TRUE/TRUSTED_FUNCTION=TRUE` for OIL and the existence of (at least one) `OsApplicationTrustedFunction` container.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TRUSTED_ FUNCTION =TRUE/TRUSTED FUNCTION =TRUE/NAME	OsTrustedFunction Name	-	List of trusted functions provided by this application.
TRUSTED_ FUNCTION =TRUE/TRUSTED FUNCTION =TRUE/Params	OsApplication Params	-	Parameter (arguments) of trusted function. Empty string means void. Used for stub generation only. See attribute `GenerateStub`.
TRUSTED_ FUNCTION =TRUE/TRUSTED FUNCTION =TRUE/ReturnType	OsApplication ReturnType	-	Return value data type of trusted function. Empty string means void. Used for stub generation only. See attribute `GenerateStub`.
TRUSTED_ FUNCTION =TRUE/Generate Stub	OsApplication GenerateStub	-	If set to `TRUE`, stub functions are generated for all trusted functions of this application.

Table 7-34 Sub-attributes for trusted functions

7.3.12.2 HAS_RESTARTTASK

This is an OIL attribute which has no equivalent attribute in XML. Its sub-attribute is only available if `HAS_RESTARTTASK` = `TRUE`.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
RESTARTTASK	n.a.	-	Reference to restart task for this application.

Table 7-35 Sub-attributes for Application->HAS_RESTARTTASK=`TRUE`

7.3.13 Scheduletable

Schedule tables have to be defined with the following attributes.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
Name	Short-Name	-	Name of the `SCHEDULETABLE`. This name is used as an argument to all related OSEK API functions.
Comment	n.a.	--	Any comment
COUNTER	OsScheduleTable CounterRef	-	Defines the counter used as time basis for this schedule table.
REPEATING	OsScheduleTable Repeating	-	If selected, the schedule table is performed periodically after it is started. If deselected, the schedule table is performed once per activation.
DURATION	OsScheduleTable Duration	-	Defines the length of the schedule table. This is the time from the first expiry point to the end of the schedule table or in the case of a periodic schedule table, between two subsequent first expiry points. The length might be defined in units of nsec, usec, msec, sec or ticks.
AUTOSTART	OsScheduleTable Autostart	-	> OIL: If set to `TRUE`, the schedule table is activated at startup of the operating system. > XML: If attribute is present, the schedule table is activated at startup of the operating system. See chapter 7.3.13.1 for more information about the sub-attributes.
LOCAL_TO_ GLOBAL_TIME_SY NCHRONIZATION	OsScheduleTable Sync	-	Defines if the schedule table shall be synchronized to a global time source. This attribute is meant for the usage with scalability classes SC2 and SC4. Sub-attributes are described in chapter 7.3.13.6.
EXPIRY_POINT	OsScheduleTable ExpiryPoint	-	Defines an expiry point for this schedule table
ACCESSING_	OsSchTbl	-	Defines access rights of an application for this

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
APPLICATION	Accessing Application		schedule table. This attribute can be used multiply, so different applications might have access rights to this alarm. This attribute can be used in scalability classes SC3 and SC4 only.

Table 7-36 Attributes of SCHEDULETABLE

7.3.13.1 AUTOSTART / OsScheduleTableAutostart

> OIL: If this attribute is set to `TRUE` sub-attributes are visible and the schedule table is auto started.

> XML: If this container is present in the configuration the schedule table is auto started

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
APPMODE	OsScheduleTable AppModeRef	-	Defines an application mode in which the schedule table is started automatically. This attribute might be defined several times to start the schedule table in several application modes.
TYPE	OsScheduleTable AutostartType	ABSOLUT RELATIVE SYNCHRON	Defines the method how the schedule table is autostarted.
TYPE=ABSOLUT/ ABSVALUE	OsScheduleTable AbsValue	-	Absolute autostart tick value when the schedule table starts. Only used if the `OsScheduleTableAutostartType` is `ABSOLUTE`.
TYPE=RELATIVE/ RELOFFSET	OsScheduleTable RelOffset	-	Relative offset in ticks when the schedule table starts. Only used if the `OsScheduleTableAutostartType` is `RELATIVE`.

Table 7-37 Sub-attributes for auto start of a schedule table

7.3.13.2 EXPIRY_POINT / OsScheduleTableExpiryPoint

An expiry point consists of a sequence of actions which are performed on a given tick time of the schedule table. There are the following sub-attributes. Some of them also have sub-attributes.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
ACTION	n.a.	> OIL : ACTIVATETASK SETEVENT ADJUST	> OIL: a list of actions
OFFSET	OsScheduleTblExpPointOffset	-	Defines the time at which the defined actions occur. The time is absolute to the start of the schedule table and is given in ticks of the underlying counter.
ACTION=ADJUST	OsScheduleTblAdjustableExpPoint	-	> XML: containers for holding the sub-attributes in case of expiry point action `ADJUST`
ACTION =ACTIVATETASK	OsScheduleTableTaskActivation	-	> XML: containers for holding the sub-attributes in case of expiry point action `ACTIVATESTASK`
ACTION =SETEVENT	OsScheduleTableEventSetting	-	> XML: containers for holding the sub-attributes in case of expiry point action `SETEVENT`

Table 7-38 Sub-attributes of expiry points

7.3.13.3 Expiry point action ADJUST

> OIL: the following attributes are visible if the expiry point action is `ADJUST`

> XML: the following attributes are located in the container `OsScheduleTblAdjustableExpPoint`

Those attributes are only relevant in SC2 or SC4 if synchronization mechanisms are used.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
MAXADVANCE	OsScheduleTableMaxAdvance		The maximum positive adjustment that can be made to the expiry point offset to achieve synchronization (in counter ticks)
MAXRETARD	OsScheduleTableMaxRetard	-	The maximum negative adjustment that can be made to the expiry point offset to achieve synchronization (in counter ticks).

Table 7-39 Sub-attributes of expiry point action ADJUST

7.3.13.4 Expiry point action ACTIVATETASK

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TASK	OsScheduleTable ActivateTaskRef		Reference to the task to be activated.
Cyclic	OsScheduleTable Cyclic	TRUE, FALSE	> OIL: If set to `TRUE` this action is repeatedly added to the schedule table. > XML: This is a choice container. If set to `TRUE` the action will be repeatedly added to the schedule table. The cycle time is located in a sub-attribute. See chapter 3.2.5.3 for more details.
Cyclic=TRUE/Cycle Time	OsScheduleTable CycleTime		If the action is declared as cyclic this attribute holds the cycle time in counter ticks.

Table 7-40 Sub-attributes of expiry point action ACTIVATETASK

7.3.13.5 Expiry point action SETEVENT

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
TASK	OsScheduleTable SetEventTaskRef	-	Task to which the event should be send
EVENT	OsScheduleTable SetEventRef	-	Event to be sent to the specified task
Cyclic	OsScheduleTable Cyclic	TRUE, FALSE	> OIL: If set to `TRUE` this action is repeatedly added to the schedule table. > XML: This is a choice container. If set to `TRUE` the action will be repeatedly added to the schedule table. The cycle time is located in a sub-attribute. See chapter 3.2.5.3 for more details.
Cyclic=TRUE/Cycle Time	OsScheduleTable CycleTime	-	If the action is declared as cyclic this attribute holds the cycle time in counter ticks.

Table 7-41 Sub-attributes of expiry point action SETEVENT

7.3.13.6 LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION / OsScheduleTableSync

> OIL: If set to `TRUE` the synchronization of this schedule table is switched on and the sub-attributes will be visible.

> XML: If this attribute is present in the configuration the synchronization is used for this schedule table.

This is only relevant in SC2 and SC4.

Attribute Name		Values	Description
OIL	XML	The default value is written in bold	
SYNC_STRATEGY	OsScheduleTbl SyncStrategy	EXPLICIT IMPLICIT NONE	Defines the synchronization strategy of this schedule table. > `EXPLICIT`: The schedule table is driven by an OS counter but processing needs to be synchronized with a different counter which is not an OS counter object. > `IMPLICIT`: The counter driving the schedule table is the counter with which synchronisation is require > `NONE`: no synchronization is applied at all
SYNC_STRATEGY =EXPLICIT/PRECISION	OsScheduleTbl ExplicitPrecision	-	Defines the synchronization tolerance (in ticks) for this schedule table. If the absolute value of the deviation between the schedule table counter and the synchronization counter is smaller than this value schedule table state is set to `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS`

Table 7-42 Sub-attributes SCHEDULETABLE-> LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION = `TRUE`

8 System Generation

This chapter describes the generation of the executable program. The definition of the OIL / XML file was described in the chapter 7 Configuration. The general steps programming an application using the OSEK operating systems are illustrated in chapter 7.1 Configuration and generation process.

The dependencies on include files are described in chapter 5.2 Include Structure.

8.1 Code Generator

The code generator `GENxxxx.EXE` is delivered with the MICROSAR OS package (xxxx is replaced by the hardware platform name). The code generator is implemented as a 32-bit Windows console application and can be started from the OIL Configurator or directly from the command line.

The code generator has different command-line options. When started without any parameters, a list of all parameters is printed:

```
GENxxxx.EXE, Version: 5.00, Vector Informatik GmbH, 2008
Usage: GENxxxx.EXE [options] <Filename>
-s : print symboltable
-r <Filename> : write errors into file
-g : generate code
-d <Pathname> : path to write generated code
-m : prints list of known implementations
-i <Pathname>: include path for implementation files
-x : include path equals to generator exe path
-f <Filename>: read options and filename from command file
-y : perform a syntax check on OIL file
```

8.1.1 Generated Files

The code generator generates several files as described in chapter 5.1.2 Dynamic Files.

The files always have the same name and are written to the generation path specified in the OIL Configurator or with the command line option -d.

The '.c' modules have to be compiled and have to be linked to the application.

8.1.2 Automatic Documentation

Automatic documentation of the generation process is provided by two list files which are generated by the code generator. A basic list file is generated in text format. The more detailed list file is generated in HTML format and can be used to publish a system design in the internet or an intranet. Both files have the same name as the OIL file, i.e.:

> `<OILFileName>.lst` (basic list file in text format)

> `<OILFileName>.htm` (extended list file in HTML format)

The files are located in the directory of the OIL file.

8.1.3 Conditional Generation

If MICROSAR OS is configured using an OIL file, the OS object provides the attribute ConditionalGenerating. If AUTOSAR ECUC files are used for configuration, the parameter for conditional generation is not located in the OS configuration but in the BSWMD file of the Board, as other modules also use this parameter.

If conditional generation is selected, the generated files are overridden only if the OS configuration has changed since the last generator run. This allows using the file modification date of the generated files to decide which modules need recompilation after configuration changes. Compile times of a complete AUTOSAR stack may be dramatically reduced with this setting in the development cycle.

Setting ConditionalGenerating = FALSE forces the MICROSAR OS code generator to generate the files newly on each run. This is the recommended setting for the final, productive build.

8.1.4 Generated files backup

To avoid a mixed set of generated files from various runs of the generator, already existing files are either deleted or renamed before the new generation starts.

Before previously generated files are overwritten in a new run of a generator, the complete file set is renamed to files with the original name plus a “.bak” suffix (backup file set). This file set contains the last valid generated file set even if a consecutive generator run fails for any reason.

8.2 Application Template Generator

The application template generator `GENTMPL.EXE` is delivered with the OSEK implementation. The template generator is implemented as a 32-bit Windows console application and can be started from the OIL Configurator or directly from the command line.

The generator has different command-line options. Started without any parameter a list of all parameters is printed:

```
GENTMPLAS.EXE, Version: 5.00, Vector Informatik GmbH, 2008
Version of general code: 5.00
Usage: GENTMPLAS.EXE [options] <Filename>
       -s : print symboltable
       -r <Filename> : write errors into file
       -g : generate code
       -d <Pathname> : path to write generated code
       -m : prints list of known implementations
       -i <Pathname>: include path for implementation files
       -x : include path equals to generator exe path
       -f <Filename>: read options and filename from command file
```

All implementations are supported by the template generator.

An application template C module is generated by means of the OSEK objects defined in the OIL file.

Caution
This generator does not overwrite previously generated files. They have to be (re)moved by the user first.

8.2.1 Generated Files

The template generator generates the file `main.c` (see also chapter 5.1.2.2).

The file has always the same name and will be written into the generation path specified in the OIL Configurator.

The file has to be compiled and linked into the application.

8.2.2 Generated Code

Templates are generated for the following OSEK objects:

8.2.2.1 Task

Basic Tasks:

```
TASK(basicTask)  /* Priority: 1000  Schedule: FULL  Type: BASIC */
{
   TerminateTask();
} /* END OF basicTask */
```

Extended Tasks:

A sample event dispatcher is generated as described in the Event subsection below.

8.2.2.2 ISR

Generated code for ISRs of category 2:

```
ISR(timerInterrupt)  /* ISR of CATEGORY 2 */
{

} /* END OF timerInterrupt */
```

No code is generated for ISRs of category 1.

8.2.2.3 Hook Routines

Templates for hook routines are generated as required.

8.2.2.4 Event

A sample event dispatcher is generated for all events received by an extended task.

```
TASK(Control)  /* Priority: 100  Schedule: FULL  Type: EXTENDED */
{
   EventMaskType ev;

   /* SetRelAlarm(TCycle, MSEC(...), MSEC(...)); */
   for(;;)
   {

      WaitEvent(evA | evB | evC | evCycle | evD);
      GetEvent(Control, &ev);
      ClearEvent(ev);

      if(ev & evA)
      {
         /* user-code */
      }

      if(ev & evB)
      {
         /* user-code */
      }

      if(ev & evC)
      {
         /* user-code */
      }

      if(ev & evCycle)
      {
         /* user-code */
      }

      if(ev & evD)
      {
         /* user-code */
      }
   }

} /* END OF Control */
```

8.2.2.5 Main Function

The generated main function calls `StartOS`.

8.3 Compiler

The supported compiler package has to be installed and the search path of the compiler, assembler and linker has to be set. If special options are required they are described in the hardware specific manual.

8.3.1 Include Paths

The operating system is delivered with include files in the subdirectory `root\HwPlatform\include` (osCAN style) or `root\BSW\Os` (MICROSAR style). The delivered examples include the generated header files like `tcb.h` and `msg.h`, which are stored in the subdirectory `'.\tcb'` of each example. Compiler and assembler have to be called with search paths for these files.

9 AUTOSAR Standard Compliance

9.1 Deviations

Currently no known deviations

9.2 Limitations

9.2.1 API Function OS_GetVersionInfo

The function

```
void OS_GetVersionInfo(Std_VersionInfoType *version info)
```

is not supported by MICROSAR OS. The version information can be collected by the following #defines:

Vendor ID	OS_VENDOR_ID
Module ID	OS_MODULE_ID
Major version number	OS_SW_MAJOR_VERSION
Minor version number	OS_SW_MINOR_VERSION
Patch version number	OS_SW_PATCH_VERSION

10 Examples

Each implementation of MICROSAR OS is delivered with a set of example programs. These programs are described in the following subchapters. The example program “tutorial” is not described here but rather in the separate document [7].

10.1 Gen

This general example consists of three basic and two extended tasks. It shows the usage of the following operating system objects:

- basic and extended tasks
- events
- alarms
- resources

The program works as follows:

1. `basicTaskFirst` gets resource `resBasic`
2. `basicTaskFirst` activates task `basicTaskSecond`
3. `basicTaskFirst` releases resource `resBasic` -> `basicTaskSecond` is running
4. `basicTaskSecond` sets – if not already done – `myFirstAlarm` which will expire after 1 second and terminates
5. `basicTaskFirst` sets event `evExT1_1` to `extendedTaskFirst` -> `extendedTaskFirst` is running
6. `extendedTaskFirst` sets event `evExT2_1` to `extendedTaskSecond` and terminates -> `extendedTaskSecond` is running
7. `extendedTaskSecond` calls `ChainTask`(`extendedTaskFirst`) -> `extendedTaskFirst` is waiting for event
8. step 1
9. step 2
10. step 3
11. step 4
12. step 5
13. `extendedTaskFirst` activates `extendedTaskSecond`
14. goto 1

10.2 Driverless Vehicle

10.2.1 Introduction

This example demonstrates the usage of the timing protection feature. The application shows the control unit of a driverless vehicle. A camera input is simulated by an array for the image data, which is used to calculate nominal values for speed and direction.

The camera is assumed to provide 10 frames per second and to signal an IRQ when a new image is available. This is simulated by a cyclic IRQ with a period of 100 ms. The corresponding ISR `CameraDataAvailable` activates the basic task `CameraProcessor`. This task processes the image data and calculates nominal values for speed and direction. The image processing takes 50 ms and has to be finished when the next image is available. To show timing protection for execution time the macro `VIOLATE_EXECUTION_TIME` can be defined. After 101 images are processed, this task will then violate its execution time budget.

The cooling of the engine is controlled by a task `CoolingControl`, which is an extended task, reacting on two events. One is a cyclic event every second and the other is `HighTempEvent`. `TempHighEvent` is set by an ISR `HighTemp`, which is triggered by an external temperature sensor. As recommended for extended tasks, `CoolingControl` is implemented as an endless loop, which calls `WaitEvent` and then processes the pending events. To allow a timing analysis, `WaitEvent` only waits for the cyclic event and the regulation of the cooling unit is executed on each iteration. This is simulated by a call of `usrCoolingControl()`. If the `HighTempEvent` is pending, the task additionally stores “high temperature data” in the error log, which is simulated by a call of the function `usrStoreTemperatureHigh(void)`. The calculation is assumed to take 10 ms and it is required that each cooling regulation is finished within 1 second. To show timing protection for interrupt lock times, the API functions `SuspendOSInterrupts()` and `ResumeOSInterrupts()` are called. Depending on the value of the define `VIOLATE_INT_LOCK` 20 us or 100 us are consumed while interrupts are disabled.

The control unit shows the nominal values for speed and direction on an attached display, which should be refreshed every 500ms. This is done by task `DisplayUpdate` and assumed to take 10 ms. Updating a real display may be added in the function `usrUpdateDisplay(uint16 speed, uint16 direction)`. The tasks `DisplayUpdate` and `CameraProcessor` both access the same global variables `speed` and `direction`, these accesses are protected by a resource `DataResource`. Depending on the define `VIOLATE_RES_LOCK` this resource is locked for 20 us or 120 us to give an example of resource locking time protection, which is set to 100 us.

A background task is executed whenever no other task needs to run and consumes all the rest of the CPU time to do some memory check (simulated by call of function `usrCheckMemory(uint32 address)`). It needs to disable interrupts for a short time, which is not monitored by timing protection, but we include the locking time in timing analysis.

Priorities of ISRs:

CameraDataAvailable	High

HighTemp	Medium
SystemTimer	Low

Priorites of Tasks:

CameraProcessor	40 (high)
CoolingControl	30
DisplayUpdate	20
BackgroundTask	10 (low)

The implementation of the user hook functions is empty and may be extended by the user. If the user code takes substantial computation time, the busy-loops in the `main.c` file have to be adjusted such that the given timing constraints are still fulfilled.

10.2.2 Timing Analysis

To ensure that the system is schedulable and each task reaches its timing constraints the TimingAnalyzer can be used. The example includes a file `DriverlessVehicle.otf`, which can be used for the analysis. For this example the OIL file cannot be used in the TimingAnalyzer as we extend the analysis with resources and the SystemTimer.

The following values are extracted from the text above:

Task	ExecutionTime	Period	Deadline
CameraProcessor	50 ms	100 ms	100 ms
CoolingControl	10 ms	1000 ms	1000 ms
DisplayUpdate	10 ms	500 ms	500 ms
BackgroundTask	1000 ms	1000 ms	-

Table 10-1 Example: Driverless Vehicle – Timing Analysis; Values-1

For `BackgroundTask` no deadline is set, because it doesn't have a time critical job. It will use all available CPU time, not used by the other tasks and ISRs. Actually `BackgroundTask` is not periodic, but activated only once and running for ever. `ComputationTime` and `Period` are just set to allow a timing analysis. For the `BackgroundTask` only interrupt blocking times are relevant, as this also effects tasks and ISRs with higher priority. The interrupt blocking time of this task has to be checked manually by code inspection. Make sure to include the runtime of the API functions to disable and enable interrupts in the calculation for the runtime of the code in between.

ISR	AnalysisPriority	AnalysisPriority	Period	Deadline
CameraDataAvailable	70	200 us	100 ms	10 ms
HighTemp	60	100 us	5 s	1 s

Table 10-2 Example: Driverless Vehicle – Timing Analysis; Values-2

The computation times are calculated by adding the runtime of the API functions they call and the ISRcat2-overhead from the hardware specific documentation. The given values here are just examples. Values for Period are given by external hardware constraints.

Extended tasks with multiple events can only be analyzed and monitored by timing protection if they follow a specific pattern:

A cyclic alarm sets an event for the extended task. The task uses an endless loop of the form:

```
for(;;)
{
EventMaskType ev;

   WaitEvent(CyclicEvent);
   GetEvent(&ev);
   ClearEvent(ev);
   if(ev & Event1)
   {
        /* handle event 1 */
   }
   if(ev & Event2)
   {
        /* handle event 2 */
   }
}
```

This means, that the extended task waits only on one single event, which is set periodically. All other events, are only handled at these points in time. The period and deadline for the task are given by the period of the cyclic alarm. The execution time for the analysis is given by the time the task requires to handle all events. I.e. the worst case runtime of one iteration of the loop, between leaving and entering `WaitEvent`.

To include interrupt lock times in the analysis, a resource names `InterruptLock` is added. This resource is shared by all tasks which disable interrupts and by the ISR with highest priority. For the tasks the locking times of this resource are set to the longest interrupt lock time of this task. The occupation time of the high priority ISR does not matter. It just have to meet the constraint `0 < occupation time < executiontime` of the ISR. With this trick we ensure to include the interrupt lock times in the analysis for the possible delay of ISRs.

The resource `InterruptLock` can now also be used to model non-preemptive ISRs, i. e. ISRs with the OIL attribute `EnableNesting` set to `FALSE`. Add the resource to all these ISRs with the locking time equal to the complete execution time of the ISR.

To analyze the behavior of non-preemptive tasks we apply the same trick as for ISRs without nesting: A resource `TaskLock` is added, which is occupied by all non-preemptive tasks during their complete runtime and by the task with highest priority. Again the locking time of this high priority task does not matter.

Now there is one issue left: the system timer. To include it in the timing analysis, an ISR is created. This is a periodical ISR, with the period given by the OIL attribute `TickTime`. The system timer interrupt is required to be handled before the next timer tick occurs. I.e. the deadline for the system timer ISR is also given by `TickTime`. The computation time of the `SystemTimer` depends on the number of alarms and the configured alarm actions. To estimate the runtime add the values for ISRcat2-Overhead and the runtimes of the API functions which do the action configured for each alarm. In this example one alarm is configured which sets an event (including a task switch), therefore the runtimes for `SetEvent` (with taskswitch) and ISRcat2-Overhead have to be used.

The priority of the `SystemTimer` ISR also depends on the platform. Normally the system timer uses an interrupt level with lowest possible priority.

Caution
On some platforms it is also possible to choose whether the `SystemTimer` shall enable nesting or not. Make sure to include this correctly in the analysis.

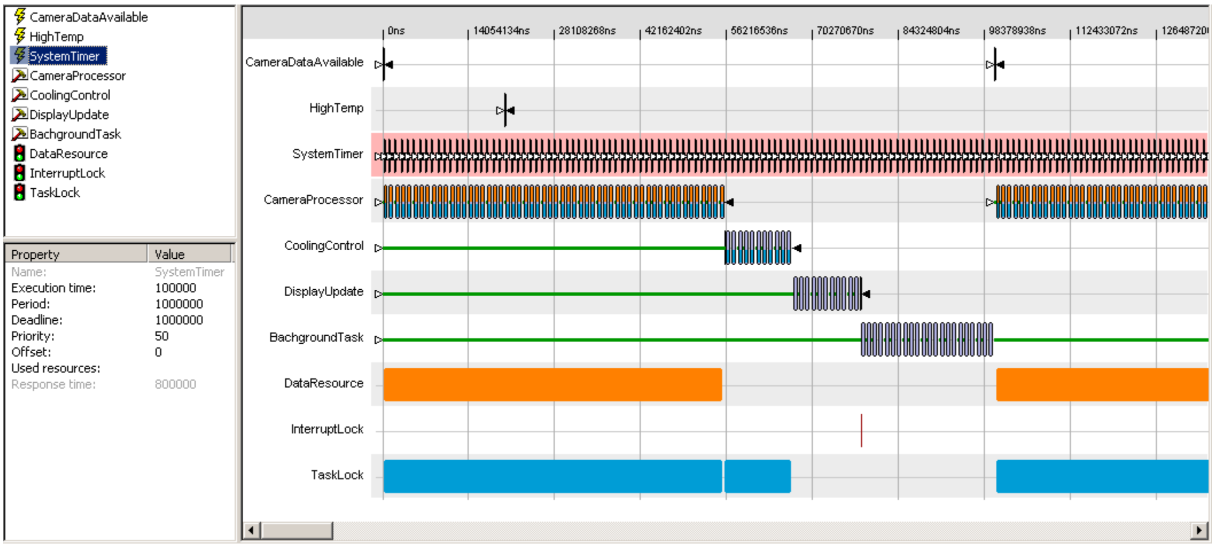

Figure 10-1 Example: Driverless Vehicle - Timing Analysis; TimingAnalyzer

The TimingAnalysis can be shown by opening the file `DriverlessVehicle.otf` in the example directory using the TimingAnalyzer.

10.2.3 Settings for Timing Protection

By the timing analysis we know that the system is schedulable under the given timing constraints. To check that the application fulfills these constraints, the timing protection feature of scalability classes 2 and 4 can be used.

Timing protection settings for tasks:

Task	EXECUTIONBUDGET	TIMEFRAME	MAXOSINTERRUPTLOCKTIME	MAXALLINTERRUPTLOCKTIME
CameraProcessor	49 ms	99 ms	0 us	0 us
CoolingControl	9 ms	999 ms	40 us	40 us
DisplayUpdate	10 ms	100 ms	0 us	0 us

Table 10-3 Example: Driverless Vehicle – Settings for Timing Protection; Values-1

BackgroundTask cannot be observed by timing protection, because it is not released periodically and never terminates.

The resources which are occupied are configured with the attribute LOCKINGTIME:

Task	RESOURCE	RESOURCELOCKTIME
CameraProcessor	DataResource	49 ms
DisplayUpdate	DataResource	100 us

Table 10-4 Example: Driverless Vehicle – Settings for Timing Protection; Values-2

The resources InterruptLock and TaskLock are just introduced for the TimingAnalysis and not configured in the OS configuration.

ISR	EXECUTION-TIME	TIMEFRAME	MAXOS-INTERRUPT-LOCKTIME	MAXALL-INTERRUPT-LOCKTIME
CameraDataAvailable	200 us	99 ms	0 us	0 us
HighTemp	100 us	5000 ms	0 us	0 us

Table 10-5 Example: Driverless Vehicle – Settings for Timing Protection; Values-3

The ISR SystemTimer is also not added in the configuration of the OS. It is automatically added during the code generation for the OS. In the above example we just added the ISR for the analysis. The SystemTimer is not monitored by timing protection.

10.2.4 Adaption of the Example

To try out timing protection the source code uses the preprocessor defines preprocessor VIOLATE_INT_LOCK, VIOLATE_RES_LOCK, VIOLATE_EXECUTION_TIME and VIOLATE_ARRIVAL. These may be defined as “1” in the file main.c to configure the source code to violate the corresponding timing protections. You can use a breakpoint in the function ProtectionHook, to analyze the protection violation and see an example how to react on them.

In the file UserSpec.c there are a few callback functions which may be filled with user specifc code. This can be used to make the example more interactive, for example by switching LEDs on an evaluation board.

10.3 ScheduleTables

This example shows the usage of Schedule Tables. The example can also be used to visually view the ScheduleTables in the TimingAnalyzer.

The TimingAnalyzer can be started from the OIL Configurator. The ScheduleTable can be activated in the TimingAnalyzer and a simulation can be started that shows a graphical representation of Schedule Tables.

10.4 ECG

The example ECG uses OSEK COM messages for intertask communication. This example shows how the heart rate monitoring function of an electrocardiogram (ECG) works. The ECG has an alarm display which is activated if abnormal cardiac events are detected. All measurements are sent to the ECG's chart recorder. One or more ECGs can be connected to a remote analyzer workstation which queries the measured heart rate data from the ECG. If a query message is received by the ECG hardware an interrupt is raised. The activities of both the patient and the remote monitor are simulated.

The system consists of the following software components:

> `SimulationControl`

  `BasicTask` which is periodically activated by the `SimulationTimer` (cyclic timer with 1s cycle). The cardiac event to be simulated is sent via an unqueued message to the `HeartRateSensor`. The `HeartRateSensor` task is notified by an event associated with the message.

> `HeartRateSensor`

  `ExtendedTask` which waits for events in an endless loop. If a `NotifyHeartRate` event is received which is triggered by the `NotifyTimer` (cyclic timer with 100ms cycle) a `HeartRateMsg` is sent to the `ChartRecorder` and the `HeartRateMonitor`. A `HeartRateMsgRemote` (unqueued message) is sent to the `RemoteAnalyzer`. `HeartRateMsg` is a queued message (with queue size 3) which notifies the `HeartRateMonitor` task by an event that new heart rate data is available. `HeartRateMsg` contains the current rate generated out of the simulated abnormal cardiac event condition and a time-stamp. If a `SimulateAbnormalEvent` event is received, the current heart rate is recalculated.

> `HeartRateMonitor`

  `Extended Task` which waits for events in an endless loop. If a `NotifyHeartRate` event is received the heart rate in the message is used to detect abnormal cardiac events. If the heart rate is below 40 beats per second, a bradycardia alarm is declared; if it is above 120 bps, a tachycardia alarm is declared. In either case an `AbnormalEventMsg` is sent to the `AlarmDisplay` which is activated by the message transmission and the `ChartRecorder`. If the measured rate reaches a normal value again, an `AbnormalEventMsg` is sent with event condition `NORMAL`. `AbnormalEventMsg` is a queued message with queue size 1.

> `AlarmDisplay`

  `Basic Task` which is activated if an `AbnormalEventMsg` was sent by the `HeartRateMonitor`. The `AlarmDisplay` evaluates the message and updates its display accordingly.

> `ChartRecorder`

  `Basic Task` running in an endless loop (background task). The task is polling `HeartRateMsg` and `AbnormalEventMsg` messages while it is continuously writing its chart.

> `RemoteAnalyzer`

  ISR triggered by the reception of data queries from the remote analyzer. It looks at whether a `HeartRateMsgRemote` is available and sends the message contents to the remote analyzer.

The following figure shows the tasks, ISRs, messages, alarms and events.

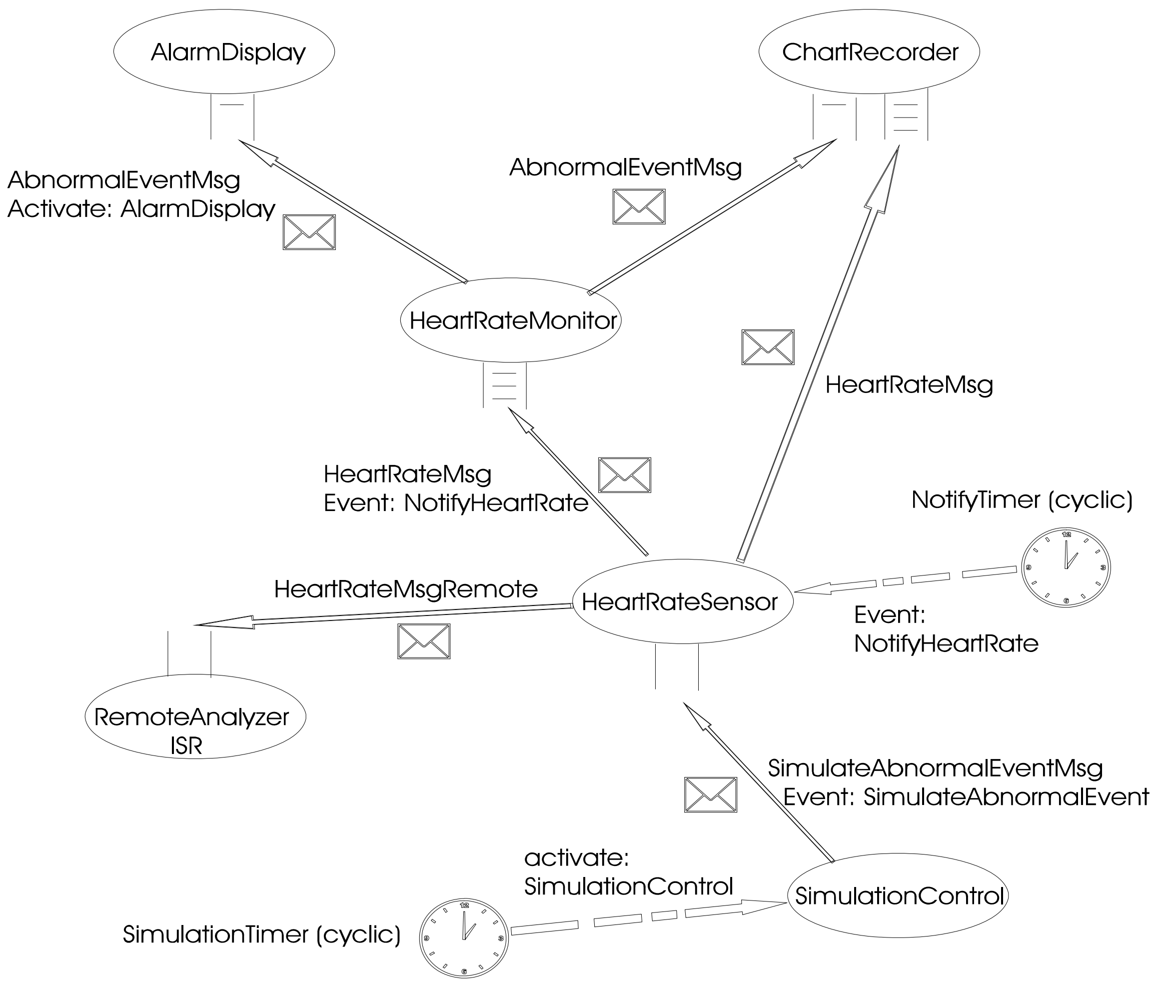

Figure 10-2 ECG example

A variety of OSEK COM's features are covered by this example:

> unqueued (`SimulateAbnormalEventMsg`, `HeartRateMsgRemote`), queued with FIFO size = 1 (`AbnormalEventMessage`) and queued with FIFO size > 1 (`HeartRateMsg`) messages

> broadcasts (`HeartRateMsg`, `AbnormalEventMessage`) and one-to-one communication (`SimulateAbnormal- EventMsg`, `HeartRateMsgRemote`)

> tasks and ISRs as message receivers

> asynchronous notification mechanisms: task activation (`AbnormalEventMessage`) and event setting (`HeartRateMessage`, `SimulateAbnormalEventMsg`)

> API calls `StartCOM`, `SendMessage`, `ReceiveMessage` and `GetMessageStatus`

10.5 Dlcomp

This example program shows the usage of the component management. A data logger component is integrated in a test system.

The data logger component consists of the following files:

> `DLCOMP_component.oil`(OIL file of the component)

> `Datalogg.c` (C source code of the component)

> `Datalogg.h` (header file of the component)

> `usrostyp.h` (Message data structures of the component)

The test system consists of the following files:

> `DLCOMP_test.oil` (OIL file of the test system)

> `Main.c` (test program)

> `Userspec.c` (user specific hook routines – may be modified)

To get the example running, the following steps are required:

1. Open `DLCOMP_test.oil` in OIL Configurator
2. In the 'manage components' window, import the file `DLCOMP_component.oil`.
   The system now consists of objects belonging to 'Main Component', which is the test system, and of objects belonging to the data logger component 'DLCOM_component'.
3. It is recommended to save the merged OIL file under a new name e.g. `DLCOMP_system.oil`.
4. Generate code
5. Build the example program

Functionality of the Data Logger Component:

The data logger component collects data using different logging modes which are set when starting the data logger:

> Event or time-triggered logging

> FIFO queue or circular buffer

To store the data a queued message is used. All API functions of the data logger component are called from the test task's context. The data logger component has an additional input and output buffer where the test task can access the data which is written or read. Synchronization of the data logger task and the test task is done by events.

10.6 Traffic

This sample application simulates the traffic on an intersection of two streets. There are four traffic lights and one control unit for all traffic lights. Each light has a sensor which monitors the traffic. The following figure shows the topology:

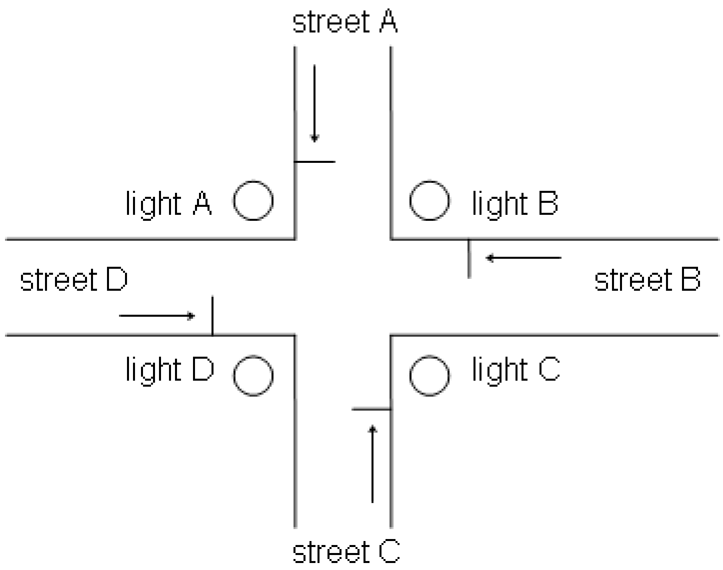

Figure 10-3 Traffic light examle

The system has the following components:

1. TrafficIn
   Simulation of the incoming traffic on each street. To avoid the generation of random numbers, equidistant arrivals are assumed.
2. Sensor
   Each traffic light has a sensor which monitors the incoming and outgoing traffic on each street. If more than `n` cars are waiting at a traffic light or at least one car is waiting longer than the time `T`, the control unit (Control) shall be notified.
3. Control
   The control unit services all streets in a round-robin fashion. If the sensor receives no notification for one street, the street will be skipped. Changes in the setting of the traffic lights are initialized in TrafficOut.
4. TrafficOut
   Simulation of the outgoing traffic on each street. Depending on the traffic light settings, departures are initialized for the appropriate sensor. Equidistant departures shall be assumed.

This problem can be solved by a realtime simulation using the OSEK operating system. In the following graph one possible solution is shown. Note that the graph has been drawn for two streets and therefore has two sensors. Due to the symmetry of the problem, a directed graph for four streets can be obtained by duplicating the sensors and their associated events and alarms.

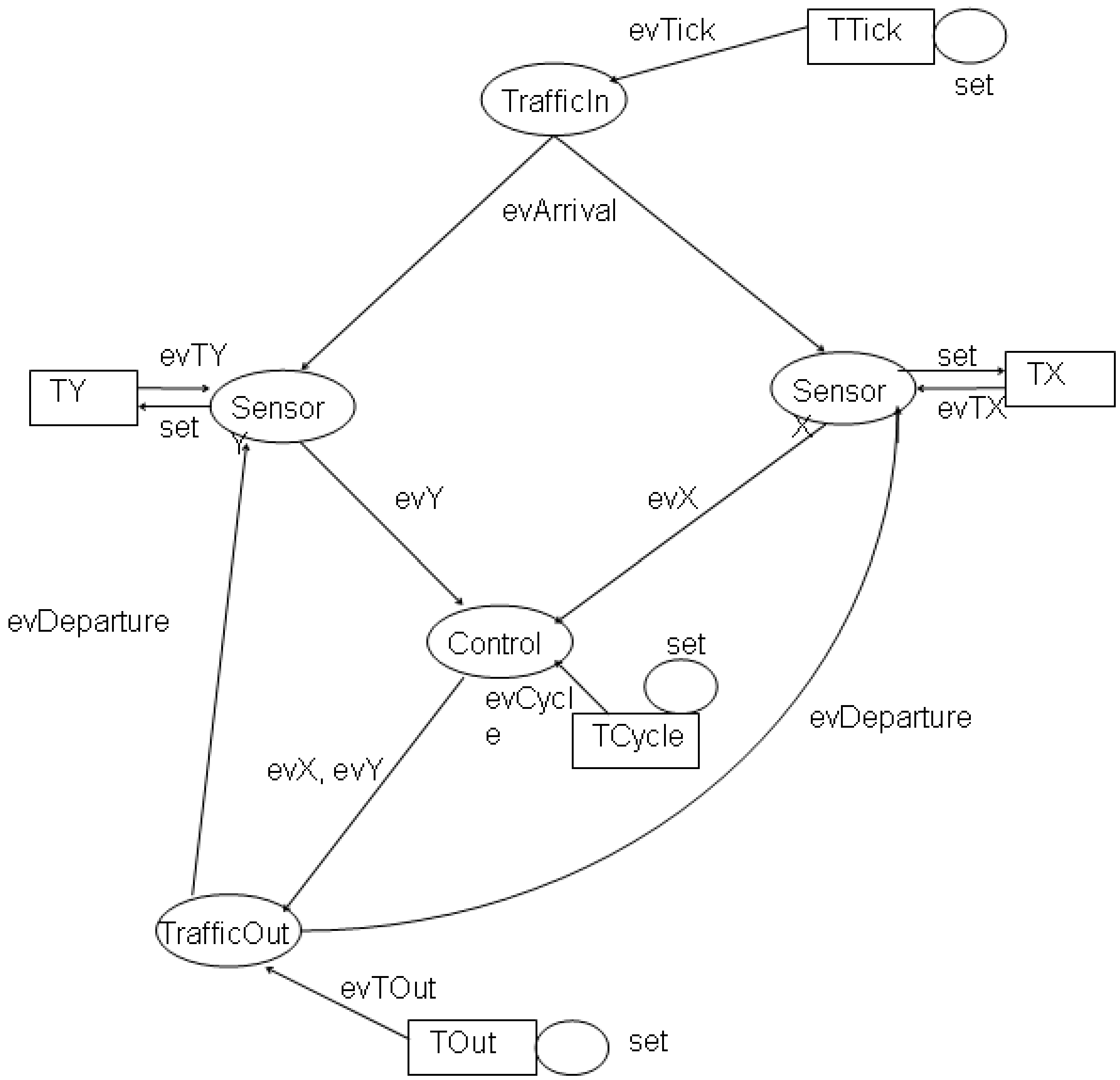

Figure 10-4 Possible solution for traffic light example

The implementation can be found in the ‘traffic’ directory. All tasks have been implemented as extended tasks waiting infinitely for incoming events. This example shows how state machines can be implemented by means of extended tasks, events, and alarms.

10.7 SimpleTrace

SimpleTrace provides an example for an SC3 application using generated stubs for trusted functions.

Application `TraceAppl` implements a simple trace using a ring buffer. The interface is provided by 4 trusted functions:

> `void TraceInit(void)`

  reset and initialize trace

- `void TraceInsert(uint32 item)`

  insert item into trace buffer

- `int TraceGetNumberOfEntries(void)`

  get number of entries in trace buffer

- `int TraceReadItem(int position, uint32 *pItem)`

  read entry at position (0 is oldest entry)

Each of these trusted functions must be configured in the OIL file. For stub generation, the generator needs two (non-standard) attributes `Params` and `ReturnType` with the function arguments and the function return type.

The generated call stubs will have the prefix `Call_<name>` (<name> is the name of the trusted function), e.g. to initialize the module, function `Call_TraceInit()` must be called.

Each trusted function with address arguments for return values must check the access rights of the caller before writing results through an address argument. Function `TraceReadItem` provides an example for this check.

11 Debugging Support

11.1 Kernel aware Debugging

All implementations of MICROSAR OS support kernel-aware debugging according to the ORTI specification. To use this feature, 'ORTIDebugSupport' must be enabled in the OIL Configurator. On some platforms, proprietary solutions are available.

Refer to the hardware specific documentation [4] for details.

11.2 Internal Trace

The internal trace of the operating system provides a trace capability at system call level. It is useful if no other trace mechanisms are available or if it is required to analyze the history of kernel activities.

11.2.1 Configuration

The trace is configured in the OIL Configurator by the attribute `InternalTrace` of the OS object. If enabled, the following settings are available:

> Size of the trace buffer (sub-attribute `TraceDepth`)

> Usage of a time stamp (sub-attribute `TimeStamp`)

Per default, the time stamp is generated by means of the MICROSAR OS system counter. If more accurate time stamps are required, e.g. to track fast task switches a user defined time stamp source can be used. In that case the current value of the time-stamp must be provided by the application with the function `osGetUserTimeStamp`. This function has the following prototype:

```
uint16 osGetUserTimeStamp(void);
```

11.2.2 Initialization

To initialize the trace the function `osInitTrace` is called automatically by the OS in `StartOS` if the trace is enabled. It is possible to reinitialize the trace later with the function `osInitTrace`:

Prototype : `void osInitTrace(void);`

11.2.3 Evaluation

The trace is evaluated by inspecting the global circular buffer `osTraceBuffer` with the structure `tOsTraceBuffer`. This structure has the following elements:

> `uint8 stateNo`: Identifier of the trace event. Possible trace events are defined in the file testmac2.h (`osdTraceXXX`).

> `uint8 taskNo`: Identifier of the task which is currently active. Possible values are defined in the file tcb.h in the 'Tasks' section.

- `uint16 timeStamp`: Time stamp of the trace event depending on the used time-stamp source.

The global variable `osTraceBufferIndex` contains the next trace buffer index which will be used to store the next trace event.

Example gen (10.1):

```
osTraceBufferIndex=5
```

The last valid entry in the trace buffer is:

```
osTraceBuffer[4]={18,4,120}
```

Interpretation:

last system call: 18 = osdTraceSetEvent -> SetEvent

current task: 4 = basicTaskFirst

time-stamp: 120 system ticks

11.2.4 User defined Trace Events

In addition to the kernel activity it is possible to track other events. In the file `testmac2.h`, 20 user events are available (`osdTraceUser00 – osdTraceUser19`) which can be used in the application by means of the trace macro `osTrace`.

Example:

```
void myFunc(void)
{
   if(error)
   {
      osTrace(osdTraceUser07);
   }
}
```

11.2.5 Example code for printing the Trace Buffer

The following example code iterates over the trace buffer in order to print it via a printf call. The algorithm starts with the oldest entry (the next to be overridden; pointed to by osTraceBufferIndex) and continues printing to the newest entry. Since the Trace Buffer is a circular buffer, after reaching the last element of the buffer array the algorithm continues with the first element of the buffer array.

```
void PrintTrace(void)
{
   int i;

   for(i=osTraceBufferIndex;i<osdTraceDepth;i++)
   {
#ifdef osdTraceUseTimestamp
      (void)printf("%3d %3d %6d\n"      ,
                   osTraceBuffer[i].taskNo,
                   osTraceBuffer[i].stateNo,
                   osTraceBuffer[i].timeStamp);
#else
      (void)printf("%3d %3d\n",
                  osTraceBuffer[i].taskNo,
                  osTraceBuffer[i].stateNo);
#endif
   }

   for(i=0;i<osTraceBufferIndex;i++)
   {
#ifdef osdTraceUseTimestamp
      (void)printf("%3d %3d %6d\n",
                    osTraceBuffer[i].taskNo,
                    osTraceBuffer[i].stateNo,
                    osTraceBuffer[i].timeStamp);
#else
      (void)printf("%3d %3d\n",
                   osTraceBuffer[i].taskNo,
                   osTraceBuffer[i].stateNo);
#endif
   }
}
```

11.3 Version and Variant Coding

The version and the variant is coded into the generated binary or HEX file. The user has the possibility to read version and variant using an emulator, or if the electronic control unit is accessible via the CCP protocol via the CAN bus.

The generator writes version and variant information into a structure, defined in `osek.h`.

```
typedef struct
{
   osuint8 ucMagicNumber1;      /* magic number:                        */
   osuint8 ucMagicNumber2;      /* defined as uint8 for independency of */
   osuint8 ucMagicNumber3;      /* byte order                           */
   osuint8 ucMagicNumber4;

   osuint8 ucSysVersionMaj;     /* version of operating system, Major   */
   osuint8 ucSysVersionMin;     /* version of operating system, Minor   */
   osuint8 ucGenVersionMaj;     /* version of code generator            */
   osuint8 ucGenVersionMin;     /* version of code generator            */
   osuint8 ucSysVariant1;       /* general variant coding 1             */
   osuint8 ucSysVariant2;       /* general variant coding 2             */
   osuint8 ucOrtiVariant;       /* ORTI version and variant             */

   …                        /* implementation specific variant coding */

} osVersionVariantCodingType;
```

The structure contains the version of the operating system (major and minor version number), the version of the code generator used (major and minor version number), information about the OS configuration bit-encoded into 8-bit values (`ucSysVariantX`) and information about usage of the OSEK runtime interface (ORTI):

The magic number is defined as 0xAFFEDEAD and may be used for an identification of the version in hex or binary files.

Bits	Meaning	Possible Values
0..1	Conformance Class	0: BCC1 1: BCC2 2: ECC1 3: ECC2
2	Status Level	0: STANDARD STATUS 1: EXTENDED STATUS
3..4	Scheduling policy	0: non preemptive 1: full preemptive 2: mixed preemptive
5	Stack Check	0: disabled 1: enabled
6	Error information level	0 STANDARD 1 Modulenames
7	OS internal checks	0 STANDARD 1 Additional

Table 11-1 Bit-definitions of the variant coding, ucSysVariant1

Bits	Meaning	Possible Values
0..1	Scalability Class	0: SC1 1: SC2 2: SC3 3: SC4
2	Usage of Schedule tables	0: no schedule tables in system 1: schedule tables are used
3	Usage of high resolution schedule tables	0: no high resolution tables in system 1: high resolution schedule tables are used
4	Schedule table synchronization	0: synchronization is not used 1: synchronization is used
5	Timing protection	0: timing protection is used 1: timing protection is switched off

Table 11-2 Bit-definitions of the variant coding, osSysVariant2

Bits	Meaning	Possible Values
0..6	ORTI version	0x00: No ORTI used 0x20: ORTI 2.0 used 0x21: ORTI 2.1 used 0x22: ORTI 2.2 used
7	ORTI additional information	0: The ORTI information is restricted to those parts that do not cause runtime or memory overhead 1: The full set of ORTI information is provided by the OS

Table 11-3 Bit definitions of the variant coding, osOrtiVariant

The data for the structure is located in the constant oskVersionVariant and specified in the OS module osek.c.

The structure also contains implementation specific variant coding which is described in the separate documentation [4].

12 Glossary and Abbreviations

12.1 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CCP	CAN Calibration Protocol
COM	Communication (= module COM in AUTOSAR/MICROSAR)
CPU	Central Processing Unit
ECU	Electronic Control Unit
EPROM	Erasable Programmable Read Only Memory
EEPROM	Electrically Erasable Programmable Read Only Memory
HIS	Hersteller Initiative Software
IRQ	
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NMI	Non Maskable Interrupt
OIL	OSEK Implementation Language
ORTI	OSEK RunTime Debugging Interface
OS	Operating System
OSEK	Abbreviation of the German term "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" - Open Systems and the Corresponding Interfaces for Automotive Electronics
RAM	Random Access Memory
ROM	Read-Only Memory
SC1, SC2, SC3, SC4	Scalability Class 1, -2, -3, -4
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification
WCET	Worst Case Execution Time
XML	Extensible Markup Language

Table 12-1 Abbreviations

13 Contact

Visit our website for more information on

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

www.vector.com

For support requests you may write to **osek-support@vector.com**