OSEK realtime operating system

osCAN

Technical Reference

Author:	Dipl.-Ing. Winfried Janz, Dipl.-Inform. Joachim Stehle, Dipl.-Ing.(FH) Torsten Schmidt, Dipl.-Ing. Rainer Künnemeyer
Date:	2007-04-04
Version:	3.12
State:	Released

Contents

1 Overview

This documentation of the OSEK[1] operating system describes the general part of all *osCAN* implementations. For each implementation the hardware specific part is described in a separate document /osCAN_HW/.

The implementation is based on the OSEK OS specification 2.2 described in the document „OSEK/VDX Operating System“ Version 2.2 /OSEK OS/. This documentation assumes that the reader is familiar with the OSEK OS specification.

This documentation describes only the operating system and the code generation tool. The following table shows the related documents.

OSEK is a registered trademark of Siemens AG.

[1] (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug, Open Systems and their interfaces to the Electronic in motor vehicles)

2 More Information

2.1 Related Documents

Abbreviation	File Name	Description
/OSEK OS/		OSEK/VDX Operating System Specification 2.2 [2]
/osCAN/	OsCAN3.pdf	User manual of Vector *osCAN* (This document)
/osCAN_HW/	Osekxxxx.pdf	User manual of Vector *osCAN*; Hardware specific part
/OSEK COM/		OSEK/VDX Communication – Version: 2.2.2 [2]
/OIL/		OIL: OSEK Implementation Language – Version: 2.3 [2]

Table 1: Documents

2.2 Support

In case of problems with osCAN or with this documentation, please contact the osCAN support. He e-mail address is: osek-support@vector-informatik.de.

[2] This document is available in PDF-format on the Internet at the OSEK/VDX homepage (http://www.osek-vdx.org)

3 System Overview

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

As a requirement from ECU’s there is no dynamic task handling, 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 will be compiled and linked together to one file which is loaded into an emulator or is burned into an EPROM or FlashEPROM.

3.1 Functional parts

The operating system is started by the application. The startup module calls the function main. In main 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.

osCAN does not rely on a special startup code. The only two requirements are that *StartOS* is called on a valid stack and that all interrupts are disabled until this call. If in special cases the application has to use interrupts before *StartOS* (this indeed is allowed for interrupts of category 1 - see chapter ‘Interrupt handling’), the application is responsible that still none of the category 2 interrupts is enabled. To allow those category 1 ISRs to call the API function *SuspendAllInterrupts*, it is important to use an ANSI conform startup code that initializes all variables to zero.

The function of the scheduler is to evaluate the task with the highest priority in the *READY* state and to call this task. If the task was previously preempted by an other higher priority task, the scheduler resumes the task.

The operating system is controlled by external events. External events can be events from interrupt routines or from the alarm management (which is also driven by an interrupt routine). So any external event will result in the change of task states.

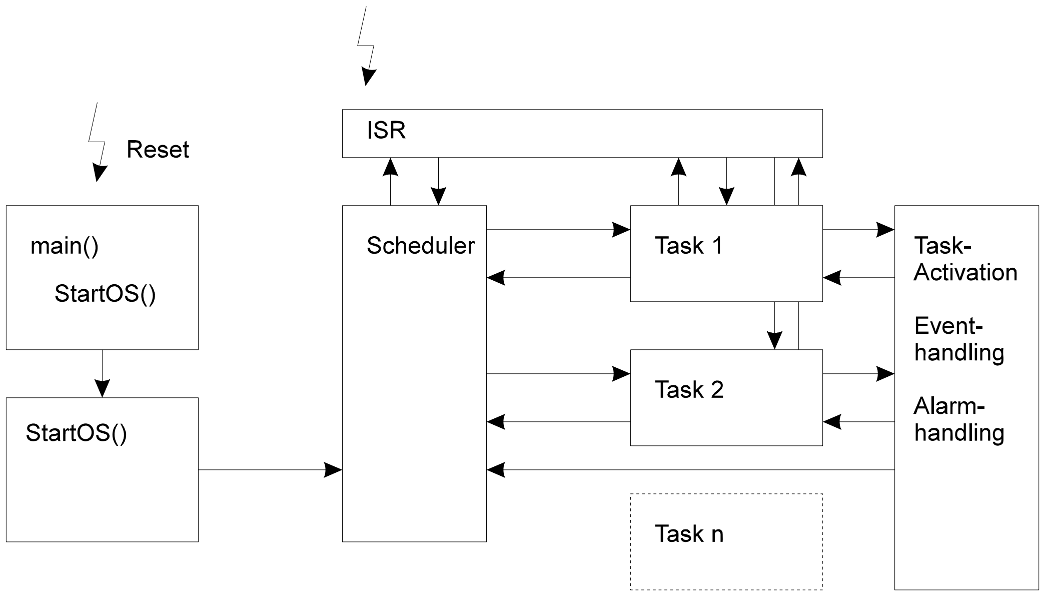


Figure 1: Functional parts

Interrupt routines are under the control of the application programmer, an OSEK operating system allows an efficient and fast 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 the existing interrupt routines.

3.2 Software parts

All operating system objects which have to be defined before the compilation are defined in OIL-definition files. OIL (OSEK Implementation language) is a grammar for defining all static OSEK objects and their attributes. The OIL specification is described in the document “OIL: OSEK Implementation Language – Version: 2.3” /OIL/. The OIL-files are read and written by the OIL-configurator. The OIL-configurator can be considered as an editor for OIL-files. The OIL-configurator is also described in the online help of the OIL-configurator.

If the OIL-file is set up the codegenerator can be started from the OIL-configurator. The codegenerator generates header files with references to all defined OSEK objects and with control switches of the operating system. The generated parts have to be linked together to the operating system modules and the application modules.

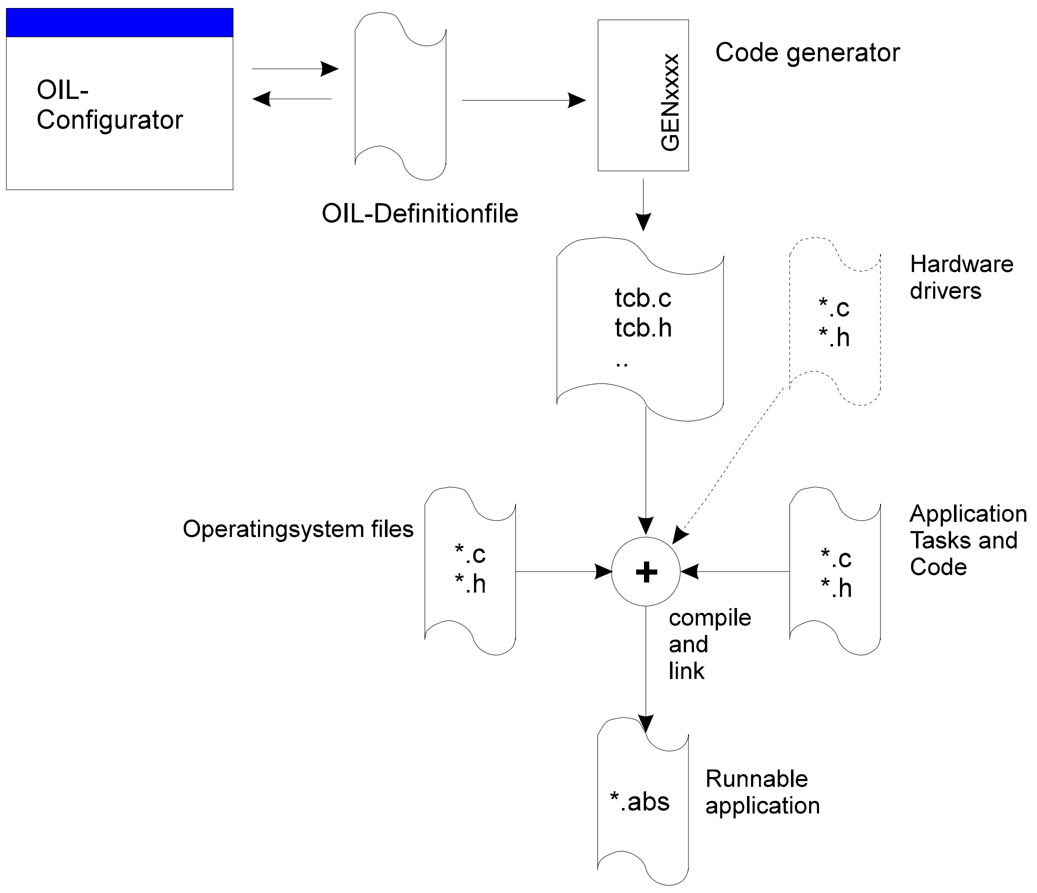


Figure 2: System overview of software parts

3.3 Implementation

The OSEK specification leaves many points open for an implementation. Every OSEK implementation for a specific microcontroller has to fix the open points to achieve an optimal solution for the processor. The operating system has to fit for the target microprocessor and for the C-compiler. The programming model of the C-compiler is as important as the hardware of the processor.

3.4 System sources

The *osCAN* system is 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 also a library version of the operating system is supplied. For different configurations different libraries have to be linked to the application.

3.5 Data protection

It is recommended to make backups of the OIL files which are edited by the OIL configurator on a regular basis. All other parts of the osCAN installation can be restored easily by re-installing the osCAN package as described in chapter 4.

4 Installation

4.1 Installation requirements

The OIL-configurator is a 32-bit Windows program.

Requirements:

- Windows95, Windows98, Windows NT, Windows 2000
- 16 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 will be delivered with a Windows installation. The installation copies all files onto the local harddisk 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 in the following referred as *root*. The delivered installation uses the path C:\OSEK as the default *root* path.

The installed components are:

• OIL-Configurator:	*root*\OILTOOL
• OSEK-system:	*root\HwPlattform*

4.3 OIL-Configurator

The OIL-configurator is a general tool for different OSEK-implementations. The implementation specifc parts are the codegenerator and the OIL-implementation files for the codegenerator.

• OIL-Configurator	*root*\OILTOOL
• OIL-Implementation files	*root*\OILTOOL\GEN
• Codegenerator	*root*\OILTOOL\GEN

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 specific files will be copied onto the local hard disk. 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 specific files are described in the hardware specific part of this manual /osCAN_HW/.

4.3.3 Codegenerator

The codegenerator GENxxxx.EXE will be copied onto the local hard disk. The codegenerator 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 this structure.

• *root*\HwPlatform\APPL\Compiler\Derivative:	Sample applications
• *root*\HwPlatform\BIN:	executable files (e.g. make tool)
• *root*\HwPlatform\DOC:	Documentation
• *root*\HwPlatform\DRV:	Driver (e.g. CAN Driver, CCP Driver)
• *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)

4.5 Applications

The APPL-directory contains five subdirectories with application examples:

• APPL\Compiler\Derivative\GEN:	Example with a variety of OSEK objects.
• APPL\Compiler\ *Derivative*\ECG:	Demo application using the intertask communication of OSEK-COM.
• APPL\Compiler\ *Derivative*\TRAFFIC:	Example with simulated traffic.
• APPL\Compiler\ *Derivative*\DLCOMP:	Example for component management.
• APPL\Compiler\ *Derivative*\WEATCTRL	Example for procedures.

5 OIL-Attributes

Before an application can be compiled all static OSEK objects have to be generated. This is done by defining all objects in the OIL-configurator and by starting the codegenerator. Some of the attributes of an OSEK objects are standard for all OSEK implementation, some are specific for the each implementation.

5.1 OIL-configurator

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 sheets for each OSEK object. 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 /osCAN_HW/.

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

5.2 OS

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

Attribute Name	Description
Name	Free selectable name, not used by the codegenerator.
Comment	Any comment.
CC	Conformance class. Possible values: BCC1, BCC2, ECC1, ECC2, AUTO
STATUS	Level for status (error) messages. Possible values: STANDARD, EXTENDED
SCHEDULE	Scheduling policy. Possible values: NON, FULL, MIXED, AUTO.
STARTUPHOOK	Selects if the hook routine *StartupHook* is called at system startup.
ERRORHOOK	Selects if the hook routine *ErrorHook* is called if an error occurs.
SHUTDOWNHOOK	Selects if the hook routine ShutdownHook is called at system shutdown.

PRETASKHOOK	Selects if the hook routine *PreTaskHook* is called at every task switch.
POSTTASKHOOK	Selects if the hook routine *PostTaskHook* is called at every task switch.
USEGETSERVICEID	If selected, the usage of the access macros to the service ID information in the error hook is enabled.
USEPARAMETER-ACCESS	If selected, the usage of the access macros to the context related information in the error hook is enabled.
WithStackCheck	If set to true at each task switch a stack check is performed. This attribute is only available if the implementation supports stacks.
UseGeneratedFastAlarm	If set to true, code is generated for each alarm, if set to false, alarms are handled with sorted lists. Enabling generated alarms leads to a better performance if only few alarms are used.
ErrorInfoLevel	If set to STANDARD, the operating system will report standard OSEK error codes and additional unique errornumbers. If set to Modulenames, additional information about the error location will be reported. Setting to Modulenames increases ROM size.
OSInternalChecks	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	The used compiler can be choosen. If there is only one compiler this attribute is also set by default.
TickTime	Duration of the system tick in µs.
SupportOfProcedures	If set to STANDARD procedure support is enabled. For tasks which have procedures assigned a C task body with procedure calls will be generated. If set to StackOptimisation optimised assembler code will be generated which allows for stack sharing if non-preemptive tasks or cooperative task groups with internal resources are used. For the availability of this option refer to the platform specific documentation.
ORTIDebugSupport	Enables kernel aware debugging with the ORTI interface.
APIOptimization	See subattributes for more information.
InternalTrace	See subattributes for more information.
EnumeratedUnhandle-dISRs	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.*

Table 2: Attributes of OS

5.2.1 Subattributes

APIOptimization = Manual:

Attribute Name	Description
APIName	If the API function corresponding to the attribute name is used in the application, the attribute has to be selected.

Table 3: Subattributes of APIOptimization = Manual

APIOptimization = Automatic:

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

InternalTrace = TRUE:

Attribute Name	Description
TraceDepth	Size of the trace buffer.
TimeStamp	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.
UsePrintout	If the platform supports a printf the contents of the trace buffer can be printed if this attribute is switched on.

Table 4: Subattributes of InternalTrace = TRUE

5.3 Task

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

Attribute Name	Description
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	Any comment.
TYPE	BASIC task, EXTENDED task or AUTO
SCHEDULE	Schedule policy for this task. Possible values: NON or FULL.
PRIORITY	The priority of the task. A higher number represents a higher priority

	(according to the OSEK specification). The priority may be set with gaps, the gaps will be eliminated by the codegenerator. 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.
ACTIVATION	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	If set to true, the task will be activated at startup of the system depending on the application modes which are defined in a list of sub-attributes.
EVENT	Reference to an event which is used by this task in case of a extended task. This attribute can be used multiple if more than one EVENT has to be assigned. If events are used with this task, the ACTIVATION cannot be greater than 1.
RESOURCE	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.
ACCESSOR	This attribute can be used multiply if more than one ACCESSOR has to be assigned. See subattributes for more information.
Procedure	This attribute is used to assign procedures to a task. See subattributes for more information. This attribute can be used multiple if more than one Procedure has to be assigned.
StackSize	Task stack size in byte. This attribute is only available if the implementation supports configurable task stacks.
NotUsingSchedule	In certain cases stacks my be shared between different non-preemptive basic-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 he may enable the stack sharing by the attribute *NotUsingSchedule*. This attribute is only available if the implementation supports stacks.

Table 5: Attributes of TASK

5.3.1 Subattributes

ACCESSOR = SENT:

Attribute Name	Description
MESSAGE	The MESSAGE reference parameter defines the message to be sent by the task.
WITHOUTCOPY	The WITHOUTCOPY parameter specifies if a local copy of the message is used.
ACCESSNAME	The ACCESSNAME parameter defines the reference which can be used by the application to access the message data.
DoNotGenerateBuffer	It is possible to use a local message buffer instead of the global generated buffer. If *DoNotGenerateBuffer* is set to TRUE no global data buffer is generated and the ACCESS-NAME can't be used. The name of the local message buffer has to be used as access name. Default: FALSE

Table 6: Subattributes of ACCESSOR = SENT

ACCESSOR = RECEIVED:

Attribute Name	Description
MESSAGE	The MESSAGE reference parameter defines the message to be received by the task.
WITHOUTCOPY	The WITHOUTCOPY parameter specifies if a local copy of the message is used.
ACCESSNAME	The ACCESSNAME parameter defines the reference which can be used by the application to access the message data.
DoNotGenerateBuffer	It is possible to use a local message buffer instead of the global generated buffer. If *DoNotGenerateBuffer* is set to TRUE no global data buffer is generated and the ACCESS-NAME can't be used. The name of the local message buffer has to be used as access name. Default: FALSE

Table 7: Subattributes of ACCESSOR = RECEIVED

Procedure = InitProcedure:

Attribute Name	Description
Name	Name of the procedure.

Table 8: Subattributes of Procedure = InitProcedure

Procedure = EventProcedure:

Attribute Name	Description
Name	Name of the procedure. This attribute can be used multiple if more than one Procedure is assigned to the EVENT.
EVENT	Reference to an event on which the procedure is called in case of an extended task.

Table 9: Subattributes of Procedure = EventProcedure

5.4 Counter

The Vector *osCAN* implementation provides always 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 an can not be changed by the user.

New counters can be added in the OIL configurator.

Attribute Name	Description
Name	Name of the counter. This name is used for the Alarm configuration.
Comment	Any comment.
MINCYCLE	The MINCYCLE attribute specifies the minimum allowed number of ticks for a cyclic alarm linked to the counter.
MAXALLOWEDVALUE	The MAXALLOWEDVALUE attribute defines the maximum allowed counter value.
TICKSPERBASE	The TICKSPERBASE attribute specifies the number of ticks required to reach a counter specific unit. The interpretation is application specific.

Table 10: Attributes of COUNTER

5.5 Alarm

The action of an alarm has to be defined statically.

Attribute Name	Description
Name	Name of the alarm. This name is used as an argument to all alarm related OSEK-API-functions (e.g. *SetRelAlarm*).
Comment	Any comment
COUNTER	This is per default the *SystemTimer*
ACTION	Possible values: SETEVENT or ACTIVATETASK or ALARMCALLBACK. See subattributes for more information.
AUTOSTART	If set to true, the alarm will be activated at startup of the system. See subattributes and Chapter Static Alarms for more information. Default: FALSE

Table 11: Attributes of Alarm

5.5.1 Subattributes

ACTION = ACTIVATETASK:

Attribute Name	Description
TASK	Task to be activated.

Table 12: Subattributes of ACTION = ACTIVATETASK

ACTION = SETEVENT:

Attribute Name	Description
TASK	Task to which the event should be send
EVENT	Event to be sent to the specified task

Table 13: Subattributes of ACTION = SETEVENT

ACTION = ALARMCALLBACK:

Attribute Name	Description
ALARMCALLBACKNAME	Name of the callback function to be activated.

Table 14: Subattributes of ACTION = ALARMCALLBACK

AUTOSTART = TRUE:

Attribute Name	Description
ALARMTIME	Static alarm time in units specified by the attribute *AlarmUnit*
CYCLETIME	Static cycle time in units specified by the attribute *AlarmUnit*
APPMODE	List of application modes where the alarm is started automatically.
AlarmUnit	USEC, MSEC, SEC or Ticks.
StaticAlarm	A static alarm is used, i.e. the alarm-time and cycle-time settings cannot be changed at run-time.

Table 15: Subattributes of AUTOSTART = TRUE

AUTOSTART = FALSE:

Attribute Name	Description
StaticAlarm	A static alarm is used, i.e. the alarm-time and cycle-time settings cannot be changed at run-time.

Table 16: Subattributes of AUTOSTART = FALSE

StaticAlarm = TRUE:

Attribute Name	Description
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 or Ticks.

Table 17: Subattributes of AUTOSTART = FALSE and StaticAlarm = TRUE

5.6 Resource

Resources have to be defined with the following attributes.

Attribute Name	Description
Name	Name of the resource. This name is used as an argument to all resource related OSEK-API-functions (e.g. *GetResource*).
Comment	Any comment
RESOURCEPROPERTY	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.

Table 18: Attributes of Resource

5.7 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 implementation he should use event names which are mapped by the codegenerator to the defined bits.

Attribute Name	Description
Name	Name of the event. This name is used as an argument to all event related OSEK-API-functions (e.g. SetEvent).
Comment	Any comment
MASK	Eventmask or AUTO

Table 19: Attributes of Event

Remark: If the user selects AUTO for the mask, the codegenerator 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 codegenerator will generate wrong bit-masks.

5.8 ISR

Attribute Name	Description
Name	Name of the interrupt service routine.
Comment	Any comment
CATEGORY	Number of category for the interrupt service routine (1-2)
RESOURCE	Reference to a resource which is occupied by this task. This attribute can be used multiple if more than one RESOURCE has to be assigned. **Remark**: This attribute is only available if the implementation supports resource management for ISRs. Details are described in the plattform specific part of this documentaion /osCAN_HW/.
ACCESSOR	This attribute can be used multiply if more than one ACCESSOR has to be assigned. See subattributes for more information.
UseSpecialFunctionName	This attribute allows to change the ISR function name. (The default function name equals the ISR name.) This attribute makes it possible to handle the interrupt requests of different interrupt sources in one single ISR function. *This feature is optional. Please refer to /osCAN_HW/ to find out whether a specific implementation of osCAN supports this feature.*

Table 20: Attributes of ISR

5.8.1 Subattributes

ACCESSOR = SENT:

Attribute Name	Description
MESSAGE	The MESSAGE reference parameter defines the message to be sent by the ISR.
ACCESSNAME	The ACCESSNAME parameter defines the reference which can be used by the application to access the message data.
DoNotGenerateBuffer	It is possible to use a local message buffer instead of the global generated buffer. If *DoNotGenerateBuffer* is set to TRUE no global data buffer is generated and the ACCESSNAME can't be used. The name of the local message buffer has to be used as the access name. Default: FALSE

Table 21: Subattributes of ACCESSOR = SENT

ACCESSOR = RECEIVED:

Attribute Name	Description
MESSAGE	The MESSAGE reference parameter defines the message to be received by the ISR.
ACCESSNAME	The ACCESSNAME parameter defines the reference which can be used by the application to access the message data.
DoNotGenerateBuffer	It is possible to use a local message buffer instead of the global generated buffer. If *DoNotGenerateBuffer* is set to TRUE no global data buffer is generated and the ACCESS-NAME can't be used. The name of the local message buffer has to be used as the access name. Default: FALSE

Table 22: Subattributes of ACCESSOR = RECEIVED

UseSpecialFunctionName = TRUE:

Attribute Name	Description
FunctionName	The name of the function that provides the ISR functionality.

Table 23: Subattributes of UseSpecialFunctionName = TRUE

5.9 Message

Messages have the following attributes:

Attribute Name	Description
Name	Name of the message. This name is used as an argument to all message related OSEK-API-functions (e.g. *SendMessage*).
Comment	Any comment
TYPE	UNQUEUED, QUEUED. See subattributes for more information.
CDATATYPE	C data type (STRING).
ACTION	Asynchronous notification mechanism: NONE, ACTIVATETASK, SETEVENT, CALLBACK or FLAG See subattributes for more information.

Table 24: Attributes of Message

5.9.1 Subattributes

TYPE = QUEUED:

Attribute Name	Description
QUEUEDEPTH	Number of message objects that go in queue (INT).

Table 25: Subattributes of TYPE = QUEUED

ACTION = ATIVATETASK:

Attribute Name	Description
TASK	Task to be activated.

Table 26: Subattributes of ACTION = ACTIVATETASK

ACTION = SETEVENT:

Attribute Name	Description
TASK	Task which an event should be send
EVENT	Event to be sent to the specified task

Table 27: Subattributes of ACTION = SETEVENT

ACTION = CALLBACK:

Attribute Name	Description
CALLBACKNAME	The CALLBACKNAME parameter defines the function which is called when the message is sent.

Table 28: Subattributes of ACTION = CALLBACK

ACTION = FLAG:

Attribute Name	Description
FLAGNAME	The FLAGNAME parameter defines the name of the flag which is set when the message is sent.

Table 29: Subattributes of ACTION = FLAG

5.10 COM

Attribute Name	Description
USEMESSAGERE-SOURCE	The USEMESSAGERESOURCE attribute specifies if the message resource mechanism is used. Setting this attribute to TRUE will lead to the COM conformance class will be CCCB. If this attribute is set the functions GetMessageResource and ReleaseMessageResource are enabled.
USEMESSAGESTATUS	The USEMESSAGESTATUS attribute specifies if the message status is available. Setting this attribute to TRUE will lead to the COM conformance class will be CCCB. If this attribute is set the function GetMessageStatus is enabled.

Table 30: Attributes of COM

Note: These settings have priority over the settings made for the *APIOptimisation* attribute in the OS object.

5.11 NM

The section NM is not used with the actual osCAN implementation.

5.12 APPMODE

Application modes have to be defined with the following attributes.

Attribute Name	Description
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	Any comment

Table 31: Attributes of Appmode

6 System generation

This chapter describes the generation of the executable program. The definition of the OIL-file was described in the previous chapter. The general steps programming an application using the OSEK operating systems are:

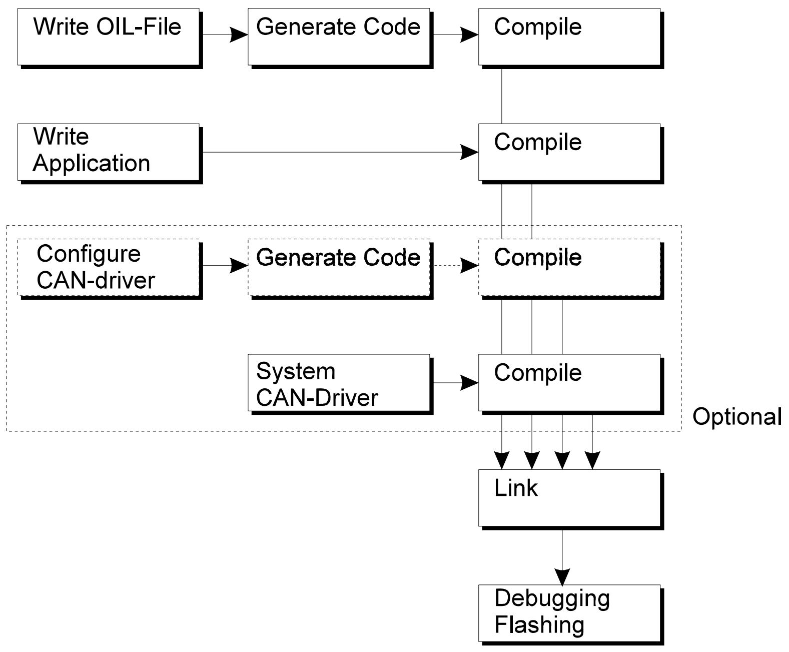


Figure 3: Systemgeneration

6.1 Codegenerator

With the *osCAN* package is delivered the codegenerator GENxxxx.EXE. (xxxx depends on the hardware platform and has to be replaced). The codegenerator is implemented as a 32-Bit Windows console application and can be started from the OIL-configurator or directly from the command line

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

```
GENxxxx.EXE, Version: 3.00, Vector Informatik GmbH, 2002
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
```

Called with the parameter –m a list of known implementations of the codegenerator is printed.

6.1.1 Generated Files

The code generator generates several of files for each *osCAN* implementation. The following 5 files are always generated:

- tcb.h
- tcb.c
- msg.h
- msg.c
- proctask.c

The files have always the same name and will be written into the generation path specified in the OIL-configurator or with the command line option -d.

The header-file 'tcb.h' is included into the file 'osek.h'. The user must incude 'osek.h' in every module of his application. The header 'tcb.h' is included automatically. Recompile all files after generation of tcb.h..

The module 'tcb.c' has to be compiled and has to be linked to the application.

The module 'msg.c' has to be linked to the application if the user uses the OSEK message communication.

The header 'msg.h' is included into 'osekcom.h'. The header file 'osekcom.h' is included automatically by 'osek.h' if messages are used in the OIL file.

The module proctask.c contains the task bodies of the tasks using procedures.

If kernel aware debugging with the ORTI interface is enabled an ORTI file is generated:

• <OILFileName>.ort	(ORTI file)

In addition some other files are generated which are described in the hardware specific part of this manual /osCAN_HW/.

6.1.2 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.

6.2 Application Template Generator

With this OSEK implementation is delivered the application template generator GENTMPL.EXE. The codegenerator 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:

```
GENTMPL.EXE, Version: 3.00, Vector Informatik GmbH, 2001
Usage: GENTMPL.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.

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

6.2.1 Generated Files

The template generator generates 1 file:

- main.c

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

The module ‘main.c’ has to be compiled and has to be linked to the application.

6.2.2 Generated Code

Templates are generated for the following OSEK objects:

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 Messages

Generated code for Messages:

SendMessage and *ReceiveMessage* calls are generated in Tasks and ISRs if a corresponding ACCESSOR attribute exists. The calls show the usage of the API-functions. *StartCOM* is generated into the StartupHook if messages exist.

6.2.2.4 Hook routines

Templates for hook routines are generated as required.

6.2.2.5 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 */
```

6.2.2.6 main function

The generated main function calls *StartOS*.

6.3 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.

6.3.1 Include paths

The operating system is delivered with include-files in the subdirectory *root\HwPlatform*\include. The delivered examples includes the generated header files tcb.h and msg.h, which are stored in the subdirectory .\tcb of each example. Compiler and assembler have to be passed the search paths for this files.

7 Timer and alarms

All time based actions are performed in OSEK using counters and alarms. Counters are part of the kernel and are incremented by a specific hardware resource. In case of a time based counter the counter is incremented periodically. An alarm has a fix reference to a counter. An alarm, if activated, has a certain value. If the referred counter reaches the given value a defined action will be 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*.

7.1 Timebase

Usually the timebase 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 timebase unit is µs.

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 /osCAN_HW/.

7.1.1 User defined SystemTimer

In all implementations it is possible to configure *UserDefined* for the *SystemTimer*. If *UserDefined* is selected the counter API 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 a ISR of category 2 for Timer7
- Select UseGeneratedFastAlarm
- Select your TickTime (e.g.1000)

The following functions are generated by the Codegenerator:

- TickType GetCounterValueSystemTimer(void);
- StatusType InitCounterSystemTimer(TickType ticks);
- void CounterTriggerSystemTimer(void);

Insert the following code in your application:

```
void StartupHook(void)
{
   /* my initialisation 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 */
      CounterTriggerSystemTimer();
      n=0;
   }
}
```

Remark: The configurated *TickTime* in the OIL-Configurator must be the same as the cyclic call time of the generated function *CounterTriggerSystemTimer*.

7.1.2 Timer-Macros

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

```
SetRelAlarm(Alarm1, USEC(1200), USEC(1200));  /* micro seconds */

SetRelAlarm(Alarm1, MSEC(10), MSEC(10));      /* milli seconds */

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-codegenerator.

REMARK: The macros are performing several multiplication and division operations. If large numbers are passed to these macros this may lead to data type overflow and/or truncation of the result. Thus the alarm expiry points will be wrong. This error is not detected by the operating system.

7.1.3 Range of alarms

The range of alarms depend 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 /osCAN_HW/.

7.2 Timer interrupt routine

The timer interrupt routine is of category 2. The ISR 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*).

7.3 Static Alarms

The *osCAN* 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.

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.

Remark: The implementation of static alarms are an extended feature of the *osCAN* implementation. The usage saves RAM, but the application code might not be portable between different OSEK implementations.

Remark: 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 “osSaveSetRelAlarm_increment()”, “osSaveSetRelAlarm_cycle()”, “osSaveSetAbsAlarm_start()” and “osSaveSetAbsAlarm_cycle()” return an undefined value.

7.4 Additional Counters

Per default *osCAN* supports one counter – the *SystemTimer*. 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*. This applies also for the static alarms.

Additional counters are an extended feature of *osCAN* and might not be portable between other OSEK implementations. Up to 256 counters are supported.

7.4.1 Counter API

To obtain the counter specific limits (e.g. `maxallowedvalue`) the function *GetAlarm-Base* can be used.

7.4.1.1 Initialisation

`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.

7.4.1.2 Read Counter

`TickType GetCounterValue<CounterName>(void);`

Return value: current counter value.

Called by the user to read the current counter value.

7.4.1.3 Trigger Counter

`void CounterTrigger<CounterName>(void);`

Called by the user usually in the ISR which is raised by the trigger source. This function increments the counter and checks the associated alarms.

7.4.2 Example

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

```
ISR(MyCounterISR)
{
   CounterTriggerMyCounter();
}

TASK(Init)
{
   InitCounterMyCounter(180);
   TerminateTask();
}
```

8 Hook routines

All hook routines defined in the OSEK specification are called and have, if enabled, to be provided by the user. The hook routines are called with disabled interrupts. The prototypes of the hook routines are described in Chapter 11: Implementation specific behavior.

8.1 Call of hook routines

StartupHook: The *StartupHook* routine is called while the operating system is initialized. The interrupts are disabled. The user may call the initialization routines for hardware drivers, activate tasks or start alarms (events may not be set).

PreTaskHook: *PreTaskHook* is called after a task is set into the RUNNING state (not into the READY state). The user can use the API-function *GetTaskID* to determine the new task.

PostTaskHook: *PostTaskHook* is called before a task is taken out of the RUNNING state. The user can use the API-function *GetTaskID* to determine the currently left task.

ErrorHook: *ErrorHook* is called every time an API-funcition is called with wrong parameters or if the system detects an error (e.g. stack overflow). *ErrorHook* is called with the parameter of type *StatusType*. The user may use the passed error number 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 /OSEK OS/. All possible access macros are supported by osCAN.

If the 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 the API-function was called. Inspecting these two variables allows the user to locate the source code which caused the error message.

ShutdownHook: The system calls the Shutdown hook routine if the function *ShutdownOS* was called.

9 Stack handling

9.1 Task stack

In general, each task has its own stack. The task stack holds all temporal 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.

9.2 Interrupt stack

The implementation of interrupt stacks depends on the hardware and is described in /osCAN_HW/.

9.3 Stack checking

Stack checking can be enabled independent of the status level (EXTENDED or STANDARD). If enabled, the system fills all stacks with the pattern 0xAA.

Each call of the task dispatcher checks the stack. The lowest address in each stack has to contain the stack pattern, if not, the system will be shut down by calling *ShutdownOS*.

9.4 Stack Sharing

One of the basic assumptions of osCAN on stacks is, that a task always starts on an empty stack. Afterwards the stack stays valid until the task terminates. So, 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 as their stack region, when none of them can start, while another one has started and not yet terminated.

The following subchapters present the configurational circumstances, detected by osCAN 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 tasks 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 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 for these tasks.

9.4.1 Basic tasks on the same priority

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

9.4.2 Basic non-preemptive tasks that do not call *Schedule*

Basic non-preemptive tasks can only be preempted, 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 osCAN, 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*. With 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 have all set the attribute *NotUsingSchedule* to TRUE, shares the same memory region for its stacks.

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

9.4.3 Basic tasks, that share an internal resource and do not call *Schedule*

This feature is optional. Please refer to /osCAN_HW/ to find out whether a specific implementation of osCAN supports this feature.

Basic tasks using an internal resource can only be pre-empted 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 osCAN, 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*. With 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 have all set the attribute *NotUsingSchedule* to TRUE, shares the same memory region for its stacks.

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

9.4.4 Stack sharing by usage of the Procedure Model

If the procedure programming model is used extended stack sharing is possible even for groups of extended tasks. Refer to chapter 14 for details.

9.5 Stack Usage

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

uint16 osGetStackUsage(TaskType taskId)

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

Note: It is required to set *WithStackCheck* = True to use this functions.

10 Interrupt handling

Implementation specific details about the interrupt handling are described in the hardware specific part of this implementation /osCAN_HW/.

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

10.1 Interrupt categories

The OSEK-OS specification defines three groups of interrupts.

10.1.1 Category 1:

Interrupts of category 1 are not allowed to use API-functions, vice versa these routines can be programmed without restrictions and are totally independent from the kernel. The programming conventions depends on the used compiler / 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.

10.1.2 Category 2:

Interrupts of category 2 may use, certain restricted, API-functions. Interrupts of category 2 can be programmed as a normal C-function 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 */

}
```

Note: 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.

11 Implementation specific behavior

11.1 API functions

The implementation of several API functions are implementation specific and are described in the document /osCAN_HW/. These functions are:

API-function	Particularities
DisableAllInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
EnableAllInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
SuspendAllInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
ResumeAllInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
SuspendOSInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
ResumeOSInterrupts	This function might be implemented using a global interrupt flag or an interrupt level register
GetResource	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 behaviour and functionality is always the same (according to the specification).
ReleaseResource	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 behaviour and functionality is always the same (according to the specification).
SetRelAlarm	According to the OSEK OS specification /OSEK OS/ the behavior of the first parameter <increment> is up to the implementation if the value is equal to 0. In the *osCAN* operating system the number of ticks is until the alarm expires is between 0 and 1. Example: If the *SystemTimer* is used for an alarm which is set with an <increment> equal to 0 and the *TickTime* is for example 1000µs, the time is between 0 and 1000µs until the alarm expires.

StartOS	After calling *StartOS* the program never returns to the call level of *StartOS*.
ShutdownOS	After the call of *ShutdownHook osCAN* disables all interrupts and will never return to the call level. The *ShutdownHook* is called with disabled interrupts.

11.2 Hook routines

The Prototypes of the hook routines are described below.

The context where hook routines are called are implementation specific and are described in /osCAN_HW/.

11.2.1 ErrorHook

```
void ErrorHook(StatusType ErrorCode);
```

See also in Chapter 8: Hook routines.

11.2.2 PreTaskHook

```
void PreTaskHook(void);
```

11.2.3 PostTaskHook

```
void PostTaskHook(void);
```

11.2.4 StartupHook

```
void StartupHook(void);
```

11.2.5 ShutdownHook

```
void StartupHook(StatusType ErrorCode);
```

12 API macros

For each API function, four macros are defined:

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)

For each hook routine, two macros are defined:

OS_Hookabbreviation_ENTRY	Called at start of hook function
OS_ Hookabbreviation_EXIT	Called at exit of hook function

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

OS_START_DISPATCH	Called at start of dispatcher

In standard configuration, these macros are empty by including the file EMPTYMAC.H. By defining *osdTestMacros* to a value of 1 - 4, the include file TESTMAC1 - TESTMAC4 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.

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

13 OSEK COM Intertask Communication

Important notes:

- Before any message related API function such as *SendMessage* or *ReceiveMessage* can be used, the OSEK COM intertask communication *must* be initialized by calling the service *StartCOM*. This is *not* done automatically in *StartOS*!
- To initialize the message objects of unqueued messages the function *MessageInit* must be provided by the application.

13.1 Overview of properties

Conformance classes:	CCCA, CCCB
Maximum number of messages:	limited by available memory
Maximum size of messages:	65535 Bytes
Maximum FIFO depth:	255 Messages
Maximum number of receivers per message:	255 tasks or ISRs
Maximum number of notifications per message	255
Status levels:	STANDARD and EXTENDED
Message types:	unqueued and queued
Asynchronous notification mechanisms:	task activation, event setting, callback and flag
Connections:	1:1, 1:N
Configurations:	WithCopy, WithoutCopy
	Selectable per sender and receiver

13.2 Buffering of Messages

If the *WithCopy* configuration is selected OSEK COM has to assure that the receiving entity gets a consistent copy of data. It has to keep a copy of the message to be red. This implies that 3 objects are required, one at the sender, one at the receiver and one in OSEK COM's internal buffer. As a consequence the data has to be copied twice.

In case of *WithoutCopy* configuration both on sender and receiver side only one global object exists which is directly accessed by the application. Data consistency has to be guaranteed by the application which accesses the message object e. g. by means of the two API calls *GetMessageResource* and *ReleaseMessageResource*.

As an addition to COM 2.2.2 and for backward compatibility to COM 2.1 this implementation supports 1:N communication for queued messages. In that case each receiver task of a queued message must have its own queue.

13.3 Synchronization

In a full preemptive system it is possible that tasks are preempted at any time. If this happens while the OSEK COM internal message object is updated and the receiving task gets active the receiver will get corrupt data if it issues a *ReceiveMessage*. Therefore the OSEK COM internal message object must be locked while it is updated by the sender. The most efficient way to achieve this goal is to disable interrupts while the object is written or read. Therefore interrupt latency increases with message length.

13.4 Data Type of Messages

For each message a user defined C-type must be defined. The type can be a standard type like *uint32* or a structure type. The typedefs of all referenced C-types except for standard types must be specified in the include file `umsgtype.h`.

13.5 Initialization of Messages

To initialize the message objects the user has to implement the function `MessageInit`. Since the queues are empty when the system starts, only unqueued messages have to be initialized. The initialization of an unqueued message can be done by direct access to the message object using the generated externals *<MessageName>UsrData*.

13.6 Message Processing

If a queued message is transmitted by the API call *SendMessage* it is first copied into the OSEK COM internal message buffer. Sending queued messages to more than one receiver means that the data must be copied to each receiver's queue. Then all specified asynchronous notifications are performed for the message.

The receiver - task or ISR - of the message calls *ReceiveMessage* to get the data. OSEK COM first determines the queue of the receiver if the message is of type QUEUED. Then the data is copied from the OSEK COM internal message buffer the application's message copy at the receiver side.

13.7 Access Name

In the *WithCopy* configuration the term access name relates to a copy of the message object. In the *WithoutCopy* configuration access name represents a pointer to the message object.

In case of a *WithCopy* accessor a global buffer with the name of the ACCESSNAME attribute is generated. If an *WithoutCopy* accessor is used this name is mapped to the message object.

13.7.1 Memory allocation options

Generation of the global buffer for a message copy in a *WithCopy* configuration can be disabled by setting the attribute *DoNotGenerateBuffer* to TRUE. In that case the message copy has to be provided by the user. This may lower the usage of global memory and may also lead to increased performance since local register variables can be used as message copies. Note that a seamless configuration change is not possible then.

13.8 API

13.8.1 Services

The following services are available as defined in the OSEK COM standard:

- StartCOM
- StopCOM
- InitCOM (call of InitCOM and CloseCOM is not necessary for intertask communication)
- CloseCOM
- *MessageInit* (performs initial initialization of unqueued messages - to be provided by the application programmer, called by StartCOM)
- SendMessage
- ReceiveMessage
- GetMessageResource
- ReleaseMessageResource
- GetMessageStatus
- ReadFlag
- ResetFlag

13.8.2 Examples

13.8.2.1 StartCOM

StartCOM must be called to initialise the OSEK COM intertask communication. This initialisation is not done in *StartOS*!

In addition *StartCOM* can be called in a startup task which is the first task running in a system or later to perform re-initialisation. In the following source code fragment the return value is omitted.

```
TASK(StartupTask)
{
   StartCOM();
   TerminateTask();
}
```

13.8.2.2 SendMessage

The following source code fragment shows how the message with name „msg2“ is transmitted. The user defined data type msgUserType2 is used. If the transmission fails an error handling function is called. In this example a local variable is used as message copy, i.e. the attribute *DoNotGenerateBuffer* is set to TRUE.

```
msgUserType2 msgSnd = {1011,2022,{6,7}};

if(SendMessage(msg2,&msgSnd)!=E_OK)
{
   ErrorHandler();
}
```

Transmission of standard data type data with message „msg3“ looks like this:

```
uint32 m3 = 333;

if(SendMessage(msg3,&m3)!=E_OK)
{
   ErrorHandler();
}
```

13.8.2.3 ReceiveMessage

The following source code fragment shows how the message with name „msg2“ is received. The user defined data type msgUserType2 is used. If the reception fails an error handling function is called. In this example a local variable is used as message copy, i.e. the attribute *DoNotGenerateBuffer* is set to TRUE.

```
msgUserType2 msgRcv;

if(ReceiveMessage(msg2,&msgRcv)!=E_OK)
{
   ErrorHandler();
}
```

Reception of standard data type data with message „msg3“ looks like this:

```
uint32 m3=0;

if(ReceiveMessage(msg3,&m3)!=E_OK)
{
   ErrorHandler();
}
```

13.8.2.4 GetMessageResource and ReleaseMessageResource

Transmission of unqueued messages using the *WithoutCopy* configuration is done by direct access of the message object. If the message object is subject to change by other tasks it can be protected by *GetMessageResource* and *ReleaseMessageResource*.

```
TASK(task1)
{
   ...

   if(GetMessageResource(msg1) == E_OK)
   {
      /* message allocated - enter critical section */
      msg1Accessor1 = 0x12345678;
      SendMessage(msg1, &msg1Accessor1); /* needed for notifications */
      /* free message - leave critical section */
```

```
            ReleaseMessageResource(msg1);
         }

         ...
      }

      TASK(task2)
      {
         ...

         if(GetMessageResource(msg1) == E_OK)
         {
            /* message allocated - enter critical section */
            if(msg1Accessor2 != 0)
            {
               ...
            }
            /* free message - leave critical section */
            ReleaseMessageResource(msg1);
         }

         ...
      }
```

Note:

- *ReceiveMessage* can be omitted since it returns only the message status
- the message object is accessed by `msg1Accessor1` and `msg1Accessor2` which are both mapped onto the message object, i.e. *<message name>UsrData.*
- deadlocks must be resolved by the application because the message resource is not an operating system resource and therefore no priority ceiling is applied to the task calling *GetMessageResource*

13.8.2.5 GetMessageStatus

The following code fragment shows the usage of *GetMessageStatus*.

```
      if(GetMessageStatus(msg2)!=E_OK)
      {
         ErrorHandler();
      }
```

13.9 Resource Requirements

Memory and timing requirements grow with the size of messages and - for queued messages - with the number of receivers. Therefore it is recommended to keep messages short. The definition of the term ‘short message’ depends on the individual environment. Larger amounts of data should be transferred by more efficient shared memory mechanisms that avoid copying of data or by means of the *WithoutCopy* configuration.

13.9.1 RAM-Requirements

The following list shows the RAM requirements of OSEK-COM for STANDARD STATUS and CCCB (for systems without message resources, callback or flag notifications) in bytes for one message. Note that the message copies required at the sender and the receiver are not taken into account. ‘message’ is the message object itself.

- unqueued message: sizeof(message)
- queued message:
  NumberOfReceivers * (2 + 2*sizeof(PTR) + FifoSize * sizeof(message))

Note: Values measured on different architectures may vary slightly due to alignment. PTR represents a pointer.

13.9.2 ROM-Requirements

The following list shows the ROM requirement of OSEK-COM for STANDARD STATUS and CCCB (for systems without message resources, callback or flag notifications) in Bytes for one message.

- 6 + 3*sizeof(PTR)
- queued message: add 3*sizeof(PTR) * NumberOfReceivers
- per notification: add 1 + sizeof(TaskType) + sizeof(EventMaskType)

Note: Values measured on different architectures may vary slightly due to alignment. PTR represents a pointer.

13.9.3 Code size

Refer to the platform specific *osCAN* documentation.

13.9.4 Time-Requirements

Refer to the platform specific *osCAN* documentation.

14 Procedures

14.1 Concept

Usually a task contains a set of functionality which is implemented in different C functions which are called by this task. In osCAN these functions are referred to as *procedures*. Two programming models are provided – one for Basic and one for Extended tasks. For these models the structure and the names of the procedures can be specified in the OIL file which enables the code generator to generate the task bodies of the tasks containing procedures automatically. Therefore the user will only have to implement the procedures for such a task. Furthermore special optimisations are possible if non-preemptive tasks or task groups with internal resources are used.

14.2 Programming models

14.2.1 Basic task

For a basic task the code which is generated for a task using procedures has the following structure:

```
TASK(myBasicTask)
{
   Procedure_0();
   Schedule();
   Procedure_1();
   Schedule();
   Procedure_2();
   TerminateTask();
}
```

Each task can consist of multiple procedures which are executed subsequently.

14.2.2 Extended task

The programming model of an extended task consists of an initialisation phase and an event dispatcher.

```
EventMaskType osevmyExtTask;
TASK(myExtTask)
{
   Procedure_0();
   Schedule();
   Procedure_1();
   Schedule();
   Procedure_2();

   for(;;)
   {
      WaitEvent(event0 | event1);
      GetEvent(myExtTask,& osevmyExtTask);
      ClearEvent(osevmyExtTask);

      if(osevmyExtTask & event0)
      {
         Procedure_3();
         Schedule();
         Procedure_4();
         Schedule();
      }
```

```
            if(osevmyExtTask & event1)
            {
               Procedure_5();
               Schedule();
            }
         }
      }
```

Both in the initialisation phase and after reception of any event multiple procedures can be called.

14.2.3 Preemption

Note that the *Schedule* calls will be generated for a non-preemptive task or a cooperative task which belongs to a task group using an internal resource. This means, non-preemptive tasks containing procedures can be preempted after each procedure and procedures of different non-preemptive tasks can never interrupt each others. Also tasks of one cooperative task group will never interrupt each others. In contrast procedures of full preemptive tasks can be interrupted and can interrupt each others at any time. This has to be taken into account when considering data consistency issues.

In procedures the call of API-functions which can lead to a preemption of the task, i.e. *Schedule* and *WaitEvent* is not allowed. This is asserted by the operating system if the *OS-InternalCheck* attribute is set to *Additional*.

14.2.4 Combination of tasks with and without procedures

Tasks with and without procedures can be combined. The task bodies as described above will only be generated for task which have at least one procedure assigned.

14.2.5 Prototype of a procedure

All procedures have the following C language prototype:

```
void <ProcedureName>(void);
```

14.3 Development process

To enable procedure support the attribute *SupportOfProcedures* has to be set to *STANDARD* or *StackOptimisation* in the OS section of the OIL configurator.

For each task which shall use procedures the names of the required procedures are entered in the OIL configurator in the *Procedure* section.

The code generator generates one additional file *proctask.c* containing task bodies which comply with the programming models described above.

Now the procedures have to be implemented.

Note that the file *proctask.c* is generated at every run of the code generator.

14.4 Stack optimisation

As stated earlier non-preemptive tasks or tasks of one cooperative group using procedures can interrupt each others only when *Schedule* or *WaitEvent* is called. If the code generator generates assembler code instead of C code the register usage and the stack usage inside the generated task bodies is well known by the operating system. Therefore it is possible to use one stack for all non-preemptive tasks and for each cooperative group since the context can be reduced to the return address to the generated task.

osCAN will use one stack for all non-preemptive Basic tasks which do not call *Schedule* and all non-preemptive tasks using procedures.

Note that this optimisation is possible both for Basic and Extended tasks. Therefore it is possible to implement non-preemptive and cooperative event driven systems with multiple scheduling points inside a task with an extremely low stack usage which comes close to the stack usage of a system without operating system.

To enable stack optimisation the attribute *SupportOfProcedures* has to be set to *StackOptimisation* in the OS section of the OIL configurator.

Note that this optional feature is not available on all platforms.

Also note that the restrictions concerning the usage of *Schedule* and *WaitEvent* only applies to non-preemptive and cooperative tasks which are using the procedure model. There are no restrictions for other tasks.

15 Error handling

15.1 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 .

15.2 OSEK error numbers

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

E_OK	0
E_OS_ACCESS	1
E_OS_CALLEVEL	2
E_OS_ID	3
E_OS_LIMIT	4
E_OS_NOFUNC	5
E_OS_RESOURCE	6
E_OS_STATE	7
E_OS_VALUE	8

Table 32: OSEK error numbers

The additional, implementation specific error numbers are defined as:

E_OS_SYS_ASSERTION	9
E_OS_SYS_ABORT	10
E_OS_SYS_DIS_INT	11
E_OS_SYS_API_ERROR	12
E_OS_SYS_ALARM_MANAGEMENT	13

Table 33: Implementation specific error numbers

E_OS_SYS_ASSERTION (9)

This error is generated if the kernel detects an internal inconsistency. The reason and an exact explanation is described below.

E_OS_SYS_ABORT (10)

This error is generated if the kernel has to shut down the system but the reason was not an API-function.

E_OS_SYS_DIS_INT (11)

This error is generated if the user has called an API-function which may lead to a task switch with disabled interrupts (either with a disabled global interrupt flag or with a too high interrupt level).

E_OS_SYS_API_ERROR (12)

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.

More implementation specific errors may be described in /osCAN_HW/.

15.3 OSEK COM error numbers

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

E_OK	0
E_COM_BUSY	32
E_COM_ID	33
E_COM_LIMIT	34
E_COM_LOCKED	35
E_COM_NOMSG	36
E_COM_RX_ON[3]	37

Table 34: OSEK COM error numbers

[3] E_COM_RX_ON is not used in this implementation.

The additional, implementation specific error numbers are defined as:

E_COM_SYS_NOT_SUPPORTED	48
E_COM_SYS_NOTIFY_FAILED	49
E_COM_SYS_QUEUE_NOT_FOUND	50

Table 35: Implementation specific error numbers

E_COM_SYS_NOT_SUPPORTED (48)

This error is generated if an unsupported feature was requested.

E_COM_SYS_NOTIFY_FAILED (49)

This error is generated if an asynchronous notification (ActivateTask or SetEvent) could not be performed.

E_COM_SYS_QUEUE_NOT_FOUND (50)

This error is generated if the receiver queue of a queued message could not be determined (internal error).

15.4 osCAN error numbers

In addition to the OSEK error numbers all *osCAN* 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 *osCAN* implementations are described below. The implementation specific error numbers have a function group number >= 0xA000 and are described in the document /osCAN_HW/.

To access these error numbers the ERRORHOOK has to be enabled. Then the numbers are accessible via the macro "OSErrorGetosCANError()".

Error types:

Type	Description
OSEK	OSEK error. After calling *ErrorHook*, the program is continued.
Assertion	System assertion error. After calling *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.
Syscheck	System error: After calling *ErrorHook* the operating system is shut down. Refer to the specific error for a description how to enable or disable error checking.

15.4.1 Error numbers of group: Task Management

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

Error numbers of the group:

Name of error	Error	Type	Reason
OsdErrATWrongTaskID	0x1101	OSEK	Called with invalid task ID
OsdErrATWrongTaskPrio	0x1102	assertion	task has wrong priority level
OsdErrATMultipleActivation	0x1103	OSEK	number of activation of activated task exceeds limit
OsdErrATIntAPIDisabled	0x1104	OSEK	Interrupts are disabled with functions provided by OSEK
OsdErrATAlarmMultipleActivation	0x1105	OSEK	Number of activation of activated task exceeds limit (task activation is performed by alarm-expiration)
OsdErrTTDisabledInterrupts	0x1201	OSEK	TerminateTask called with dis-

			abled interrupts
OsdErrTTResourcesOccupied	0x1202	OSEK	TerminateTask called with occupied resources
OsdErrTTNotActivated	0x1203	assertion	TerminateTask attempted for a task with activation counter == 0 (not activated)
OsdErrTTOnInterruptLevel	0x1204	OSEK	TerminateTask called from an interrupt service routine
OsdErrHTInterruptsDisabled	0x1301	OSEK	ChainTask called with disabled interrupts
OsdErrHTResourcesOccupied	0x1302	OSEK	ChainTask called with occupied resources
OsdErrHTWrongTaskID	0x1303	OSEK	new task has invalid ID
OsdErrHTNotActivated	0x1304	assertion	tried to terminate a task which have an activation counter which is zero
OsdErrHTMultipleActivation	0x1305	OSEK	Number of activation of new task exceeds limit
OsdErrHTOnInterruptLevel	0x1306	OSEK	ChainTask called on interrupt level
OsdErrHTWrongTaskPrio	0x1307	assertion	task has wrong priority level
OsdErrSHInterruptsDisabled	0x1401	OSEK	Schedule called with disabled interrupts
OsdErrSHOnInterruptLevel	0x1402	OSEK	Schedule called on interrupt level
OsdErrSHScheduleNotAllowed	0x1403	assertion	Schedule called from task with enabled stack sharing by setting *NotUsingSchedule* in the OIL-configurator
OsdErrSHInContextOverlayProc	0x1404	assertion	Schedule called inside a procedure
osdErrSHResourcesOccupied	0x1405	OSEK	Called with an occupied resource
OsdErrGSWrongTaskID	0x1501	OSEK	Called with invalid task ID
OsdErrGSIntAPIDisabled	0x1502	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrGIIntAPIDisabled	0x1601	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrMTMissingTerminateTask	0x1701	syscheck	Exit of task without the call of *TerminateTask* or *ChainTask.* This error is detected in *EXTENDED*

			STATUS only.

Table 36: Error numbers of group: Task Management

15.4.2 Error numbers of group: Interrupt handling

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

Error numbers of the group:

Name of error	Error	Type	Reason
osdErrEAIntAPIWrongSequence	0x2401	assertion	DisableAllInterrupts not called before
osdErrDAIntAPIDisabled	0x2501	assertion	Interrupts are disabled with functions provided by OSEK
osdErrUEUnhandledException	0x2801	syscheck	An unhandled exception or interrupt was detected. This error check is always enabled.
osdErrSDWrongCounter	0x2901	assertion	Wrong counter value detected
osdErrREWrongCounter	0x2A01	assertion	Wrong counter value detected
osdErrSGWrongCounter	0x2B01	assertion	Wrong counter value detected
osdErrRGWrongCounter	0x2C01	assertion	Wrong counter value detected

Table 37: Error numbers of group: Interrupt Handling

15.4.3 Error numbers of group: Resource Management

Group (3) contains the functions:

API-function	Abbreviation	function number
GetResource	GR	1
ReleaseResource	RR	2

Error numbers of the group:

Name of error	Error	Type	Reason
osdErrGRWrongResourceID	0x3101	OSEK	invalid resource ID
osdErrGRPriorityOccupied	0x3102	assertion	ceiling priority of the specified resource already in use
osdErrGRResourceOccupied	0x3103	OSEK	resource already occupied
osdErrGRNoAccessRights	0x3104	OSEK	task has no access to the specified resource
osdErrGRWrongPrio	0x3105	OSEK	Specified resource has a wrong priority. Possible reason: the task has no access rights to this resource.
osdErrGRIntAPIDisabled	0x3106	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrRRWrongResourceID	0x3201	OSEK	invalid resource ID
osdErrRRCeilingPriorityNotSet	0x3202	assertion	Ceiling priority of the resource not found in the ready bit field
osdErrRRWrongTask	0x3203	assertion	Resource occupied by a different task
osdErrRRWrongPrio	0x3204	OSEK	Specified resource has a wrong priority. Possible reason: the task has no access rights to this resource.
osdErrRRNotOccupied	0x3206	OSEK	The specified resource is not occupied by the task
osdErrRRWrongSequence	0x3207	OSEK	at least one other resource must be released before
osdErrRRIntAPIDisabled	0x3208	OSEK	Interrupts are disabled with functions provided by OSEK

Table 38: Error numbers of group: Resource Management

15.4.4 Error numbers of group: Event Control

Group (4) contains the functions:

API-function	Abbreviation	function number
SetEvent	SE	1
ClearEvent	CE	2
GetEvent	GE	3
WaitEvent	WE	4

Error numbers of the group:

Name of error	Error	Type	Reason
osdErrSEWrongTaskID	0x4101	OSEK	invalid task ID
osdErrSENotExtendedTask	0x4102	OSEK	cannot SetEvent to basic task
osdErrSETaskSuspended	0x4103	OSEK	cannot SetEvent to task in SUS-PENDED state
osdErrSEWrongTaskPrio	0x4104	assertion	Wrong task priority detected
osdErrSEIntAPIDisabled	0x4105	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrCENotExtendedTask	0x4201	OSEK	a basic task cannot clear an event
osdErrCEOnInterruptLevel	0x4202	OSEK	ClearEvent called on interrupt level
osdErrCEIntAPIDisabled	0x4203	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrGEWrongTaskID	0x4301	OSEK	invalid task ID
osdErrGENotExtendedTask	0x4302	OSEK	cannot GetEvent from basic task
osdErrGETaskSuspended	0x4303	OSEK	Cannot GetEvent from a task in SUS-PENDED state
osdErrGEIntAPIDisabled	0x4304	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrWENotExtendedTask	0x4401	OSEK	WaitEvent called by basic task
osdErrWEResourcesOccu-pied	0x4402	OSEK	WaitEvent called with occupied re-sources
osdErrWEInterruptsDis-abled	0x4403	OSEK	WaitEvent called with disabled inter-rupts
osdErrWEOnInterruptLevel	0x4404	OSEK	WaitEvent called on interrupt level
osdErrWEInContextOver-	0x4405	assertion	WaitEvent called inside procedure

layProc			

Table 39: Error numbers of group: Event control

15.4.5 Error numbers of group: Alarm Management

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

Error numbers of the group:

Name of error	Error	Type	Reason
osdErrGBWrongAlarmID	0x5101	OSEK	invalid alarm ID
osdErrGBIntAPIDisabled	0x5102	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrGAWrongAlarmID	0x5201	OSEK	invalid alarm ID
osdErrGANotActive	0x5202	OSEK	alarm not active
osdErrGAIntAPIDisabled	0x5203	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrSAWrongAlarmID	0x5301	OSEK	invalid alarm id
osdErrSAAlreadyActive	0x5302	OSEK	alarm already active
osdErrSAWrongCycle	0x5303	OSEK	specified cycle is out of range
osdErrSAWrongDelta	0x5304	OSEK	specified delta is out of range
osdErrSAIntAPIDisabled	0x5305	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrSLWrongAlarmID	0x5401	OSEK	Invalid alarm ID
osdErrSLAlreadyActive	0x5402	OSEK	alarm already active
osdErrSLWrongCycle	0x5403	OSEK	Specified cycle is out of range
osdErrSLWrongStart	0x5404	OSEK	Specified start is out of range

osdErrSLIntAPIDisabled	0x5405	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrCAWrongAlarmID	0x5501	OSEK	invalid alarm ID
osdErrCANotActive	0x5502	OSEK	alarm not active
osdErrCAIntAPIDisabled	0x5503	OSEK	Interrupts are disabled with functions provided by OSEK
osdErrCAAlarmInternal	0x5504	syscheck	Internal error detected while alarm was cancelled. This error is only detected when *OSInternalChecks* is set to *Additional.*

Table 40: Error numbers of group: Alarm Management

15.4.6 Error numbers of group Operating system execution control

Group (6) contains the functions:

API-function	abbreviation	function number
osCheckStackOverflow	SO	1
osSchedulePrio	SP	2
osGetStackUsage	SU	3
osCheckLibraryVersionAndVariant	CL	4
osErrorHook	EH	5
StartOS	ST	6

Error numbers of the group:

Name of error	Error	Type	Reason
osdErrSOStackOverflow	0x6101	syscheck	Task stack overflow detected. This error is only detected when the OIL attribute *WithStackCheck* ist set to *TRUE*.
osdErrSPInterruptsEnabled	0x6201	assertion	Scheduler called with enabled interrupts
osdErrSUWrongTaskID	0x6301	assertion	Called with invalid task ID
osdErrCLWrongLibrary	0x6401	syscheck	Wrong library linked to application. This error check is always enabled.
osdErrEHInterruptsEnabled	0x6501	assertion	ErrorHook called with enabled interrupts

osdErrSTMemoryError	0x6601	assertion	StartOS failed while initializing memory.

Table 41: Error numbers of group: operating system execution control

Remark: implementation specific error numbers are described in the document /osCAN_HW/.

15.5 ErrorInfoLevel

Every osCAN implementation offers an additional service for error treatment. To use this feature the OS properties EXTENDED_STATUS and ERRORHOOK has 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.

15.6 Reactions on error situations

Depending on discovered errors different reactions on errors 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).

16 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 version and variant information is written into structure which is defined in OSEK.H.

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

   uint8 ucSysVersionMaj;    /* version of operating system, Major */
   uint8 ucSysVersionMin;    /* version of operating system, Minor */
   uint8 ucGenVersionMaj;    /* version of code generator */
   uint8 ucGenVersionMin;    /* version of code generator */
   uint8 ucSysVariant;       /* general variant coding    */

   ...                       /* implementation specific variant coding */

} osVersionVariantCodingType;
```

The structure contains the version of the operating system (major and minor version number), the version of the used codegenerator (major and minor version number) and in a 8-bit-value (ucSysVariant) bit-coded the following informations:

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

Table 42: Bit-definitions of the variant coding

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 /osCAN_HW/.

17 Examples

The delivery contains five examples:

1.	Gen	(shows the usage of the OSEK API)
2.	Traffic	(shows the usage of the OSEK API)
3.	EGC	(shows the usage of message and ISRs)
4.	Dlcomp	(shows the usage of the component management)
5.	Weatctrl	(shows the usage of procedures)

17.1 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 and
- resources.

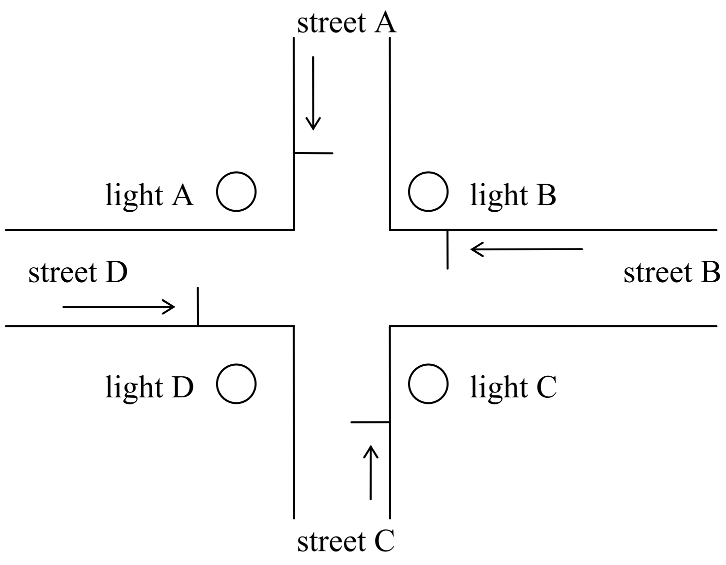
The program works as follows:

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. s. 1.
9. s. 2.
10. s. 3.
11. s. 4.
12. s. 5.

13. extendedTaskFirst activates extendedTaskSecond
14. goto 1.

17.2 Traffic

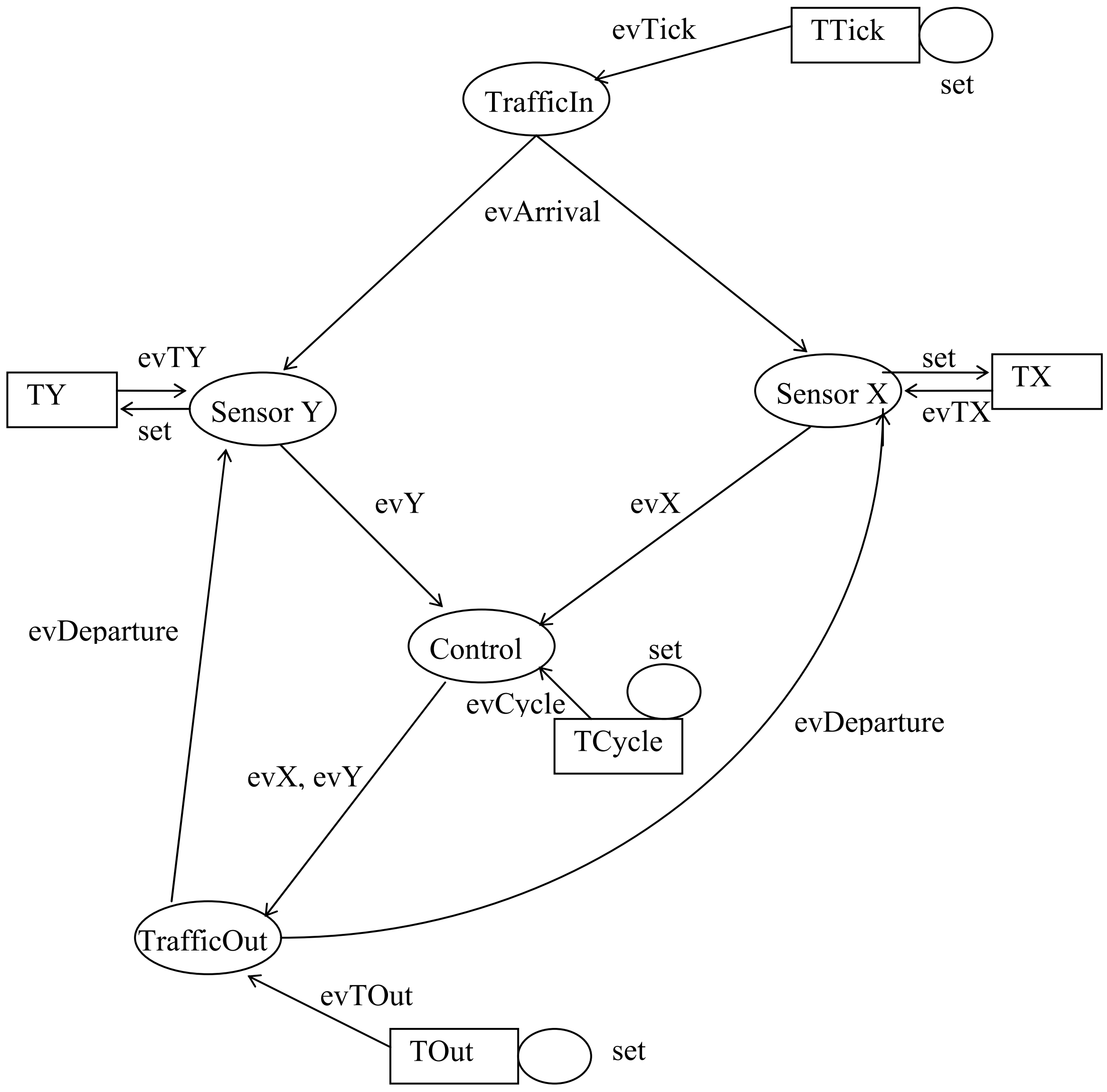
This sample application simulates the traffic on an intersection of two streets. There are four traffic lights and one control unit for all traffic lights. Each light has a sensor which monitors the traffic. The following figure shows the topology:



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 round-robin. If for one street no notification from the sensor has been received it shall be skipped. Changes in the setting of the traffic lights are signaled to TrafficOut.
4. TrafficOut
   Simulation of the outgoing traffic on each street. Depending on the traffic light setting departures are signaled to 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 but due to the symmetry of the problem a graph for four streets can be obtained by duplicating the sensors and their associated events and alarms.



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

17.3 ECG

The example ECG uses OSEK COM messages for intertask communication. This example shows how the heart rate monitoring function of an 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
  Basic Task 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
  Extended Task waiting 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. In a HeartRateMsg the current rate which is generated out of the simulated abnormal cardiac event condition and a time-stamp is conveyed. If a SimulateAbnormalEvent event is received the current heart rate is re-calculated.

- HeartRateMonitor
  Extended Task waiting 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 bps 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 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 continously writing its chart.

- RemoteAnalyzer
  ISR triggered by the reception of data queries from the remote analyzer. It looks 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.

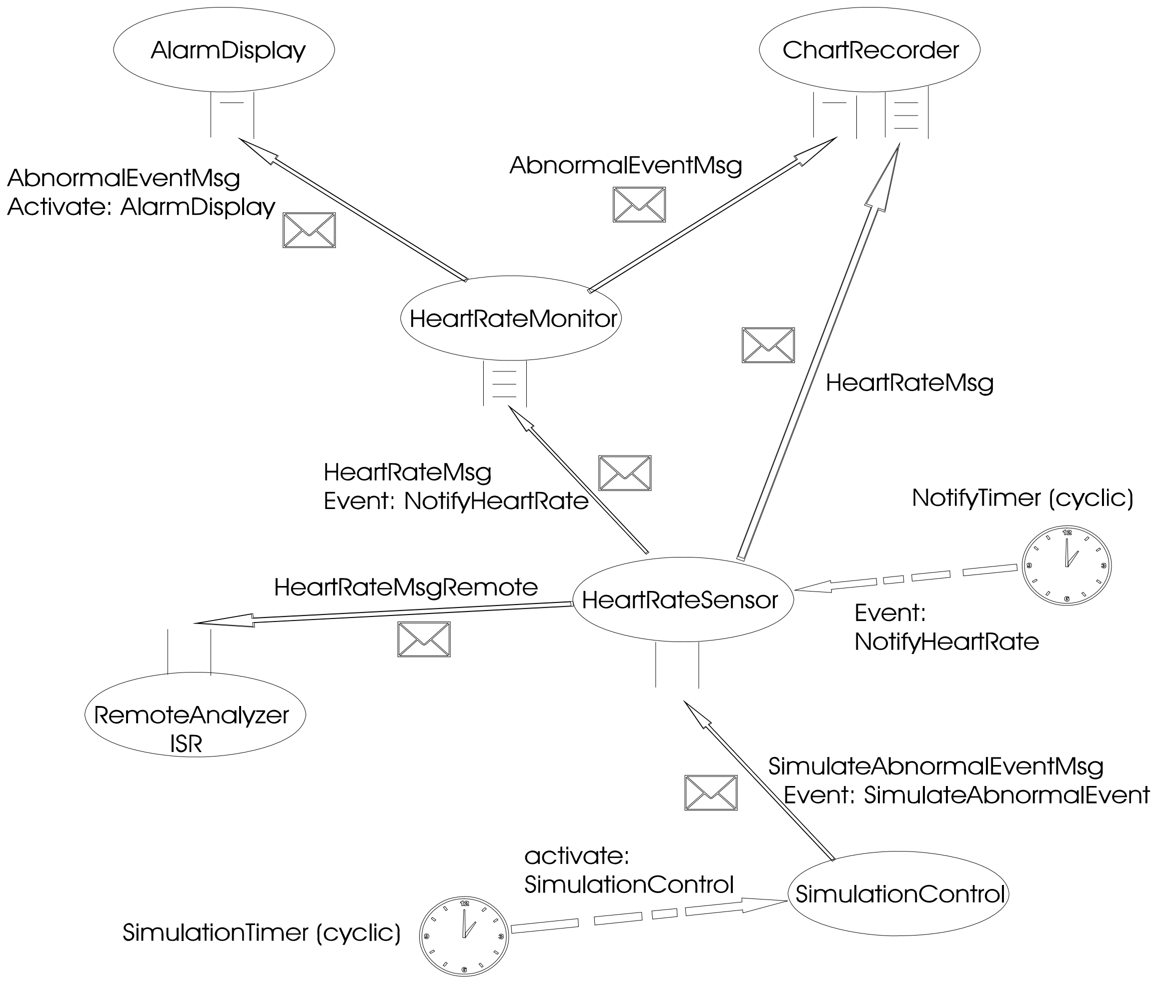


Figure 5: ECG example

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

- unqueued (SimulateAbnormalEventMsg, HeartRateMsgRemote), queued with FIFO size = 1 (AbnormalEvent- Message) and queued with FIFO size > 1 (HeartRateMsg) messages
- broadcasts (HeartRateMsg, AbnormalEventMessage) and uni-casts (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

17.4 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)
• umsgtype.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. Synchronisation of the data logger task and the test task is done by events.

17.5 Weatctrl

This example shows the usage of procedures. The control unit consists of two functional entities: A weather station which collects input data such as temperature and wind speed and two control blocks which control the operation of an outside awning and a window.

Both the awning and the window are subject to temperature, wind and rain and have to be operated accordingly.

The following input parameters are measured by an analog input channel:

- Temperature inside
- Temperature outside
- Light intensity

In addition some information are available as a binary input signal at a port:

- Rain detector
- Up button
- Down button

The third type of input channel is a hardware counter which contains the

- Wind speed

The control for the awning has the following functions:

1. If rain is detected the awning is always disabled
2. If it is dark the awning is always disabled
3. If the wind speed is at storm level the awning is always disabled
4. The awning is enabled if the sun is shining and conditions 1-3 are false
5. The awning is enabled if requested by the user (button pressed) and conditions 1-3 are false

The control for the window has the following functions:

6. If rain is detected the window is always closed
7. If the wind speed is at storm level the window is always closed
8. The window is opened if the temperature inside is greater than a limit and conditions 6-7 are false

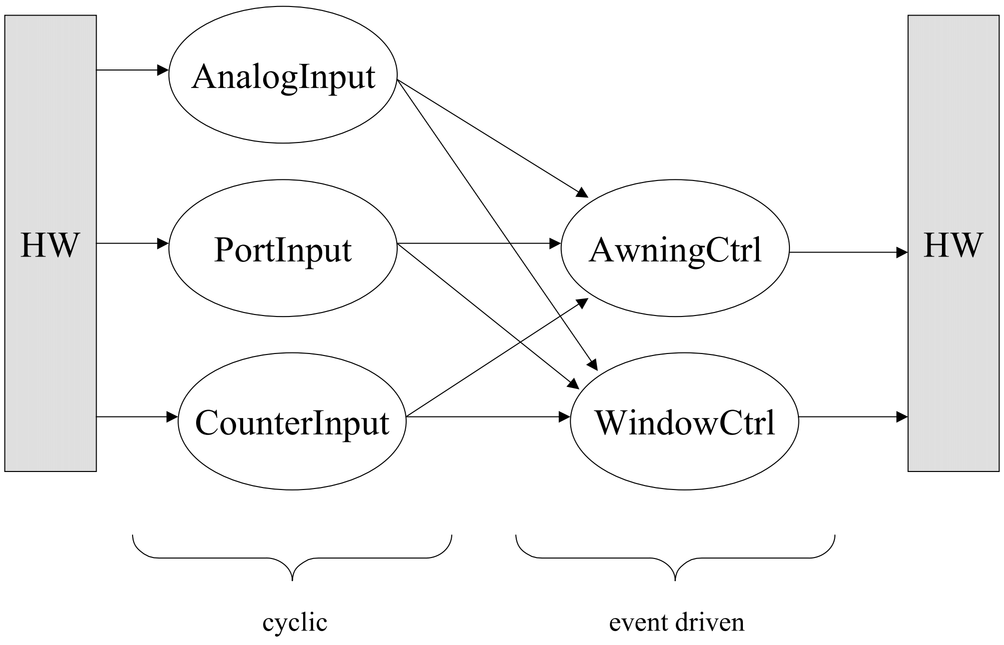
For both controls the number of enable-disable cycles per time period must be limited.

The input sub-system consists of three tasks, one for input of analog data, one for port and one for counter input. Each task contains procedures to acquire the data from the hardware by polling, process the data and detect relevant changes. E.g. for the temperatures the average of n samples has to be calculated and for the port input a debouncing is required.

The event-driven control sub-system consists of two tasks. Both awning and window control only become active if a relevant change is detected at the input side.

All tasks were implemented using the event procedure programming model, i.e. they have an initialization phase and an event handler. Since there are no requirements for short response times in the system a non-preemptive design was chosen which allows for the stack optimization as described in the procedures chapter.

In the following figure the tasks and the data flow is shown:



18 Debug support

18.1 Kernel aware debugging

All implementations of osCAN 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 /osCAN_HW/ for details.

18.2 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 analyse the history of kernel activities.

18.2.1 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 (subattribute *TraceDepth*)
- Usage of a time-stamp (subattribute *TimeStamp*)
- Usage of trace printing (subattribute *UsePrintout*)

Per default the time-stamp is generated by means of the osCAN 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);
```

If the debugger supports printing to windows or files with a printf command the contents of the trace buffer can be printed. This simplifies the evaluation of the trace.

18.2.2 Initialisation

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

Prototype: `void osInitTrace(void);`

18.2.3 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 (17.1):

`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

18.2.4 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);
   }
}
```

19 Index

20 History

Version	Date	Author	Description
1.00	08.05.2000	Si	First release
1.01	02.06.2000	Se	Update to COM 2.2
1.02	08.08.2000	HH	Support for multiple counters
1.03	14.11.2000	Se	OS_TESTMACROS replaced by osdTestMacros
2.00	15.11.2000	HH	Merged oscan.doc and comdoc.doc
	21.12.2000	Jz	Review and corrections
	10.01.2001	Si	Some corrections
2.01	15.02.2001	HH	Resource requirements of messages updated
2.02	20.02.2001	HH	Added data logger component example
2.03	14.03.2001	Si	Some corrections
2.04	31.05.2001	HH	Procedures added
2.05	2001-10-17	Si	Error message osdErrATAlarmMultipleActivation added
3.00	2001-12-18	HH	Update to OS 2.2
3.01	2002-02-06	Si	Some corrections
3.02	2002-06-10	Si	Errorcode added
3.03	2002-07-03	HH	Changed hook routines description
3.04	2002-09-09	To	Additional remark in chapter "Static alarms"
3.05	2002-09-13	To	Added chapter "ErrorInfoLevel"
3.06	2003-04-25	Rds	Correction of error types
3.07	2003-07-08	Se	Added description of errorcode osdErrSTMemoryError
3.08	2003-12-12	Se	Extended description of interrupts Added OSEK trademark Added osdErrSTMemoryError to index Some changes in description of "syscheck" errors
3.09	2007-01-29	Lam	More information about startup.
3.10	2007-02-16	To	more information about usage of the USEC, MSEC, SEC macros. New OS attribute EnumeratedUnhandledISRs.
3.11	2007-03-29	Rk Lam	Added a new stack optimization to chapter 9.4. New ISR attribute UseSpecialFunctionName.
3.12	2007-04-04	Rk	Support e-mail address added in chapter 2.2.