NXP AUTOSAR OS/S32K v.4.0

Technical Reference

Document Number: TR2OSASR4.0R1.0.0
Rev. 1.3

Contents

Section number **Title** **Page**

Chapter 1
Introduction

1.1 Introduction....17

1.2 AUTOSAR OS Overview....18

1.3 Typographical Conventions....20

1.4 References....21

1.5 Definitions, Acronyms and Abbreviations....21

Chapter 2
Operating System Architecture

2.1 Operating System Architecture....23

2.2 Processing Levels....23

2.3 Scalability Classes....24

2.4 Conformance Classes....25

2.5 OS and Application Stacks....27

2.6 AUTOSAR OS Overall Architecture....28

2.7 Application Program Interface....29

2.8 AUTOSAR OS Protection features....30

2.8.1 Service Protection....30

2.8.2 Task or ISRs with Incorrect Behavior....32

Chapter 3
Task Management

3.1 Task Management....33

3.2 Task Concept....33

3.2.1 Basic Tasks....34

3.2.2 Extended Tasks....36

3.3 Task Priorities....39

3.4 Tasks Stacks....39

3.4.1 Stack Allocation....39

3.5 Programming Issues....40
3.5.1 Configuration Options....40
3.5.2 Data Types....41
3.5.3 Task Definition....41
3.5.4 Run-time Services....42
3.5.5 Constants....43
3.5.6 Conventions....43

Chapter 4
Interrupt Processing

4.1 Interrupt Processing....45
4.2 General....45
4.3 ISR Categories....46
4.3.1 ISR Category 1....46
4.3.2 ISR Category 2....47
4.3.3 Nonmaskable Interrupt (NMI)....47
4.4 Interrupt Level Manipulation....47
4.5 ISR Stack....48
4.6 Interrupt Dispatcher....48
4.7 Programming Issues....49
4.7.1 Run-time Services....49
4.7.2 Constants....49
4.7.3 Conventions....50
4.7.4 ISR Definition....50

Chapter 5
Scheduler

5.1 Scheduler....53
5.2 General....53
5.3 Scheduling Policy....54
5.3.1 Non-preemptive Scheduling....54

5.3.2 Full-preemptive Scheduling....55
5.3.3 Mixed-preemptive Scheduling....56
5.3.4 Groups of Tasks....57
5.4 Programming Issues....57
5.4.1 Configuration Options....57
5.4.2 Run-time Services....57

Chapter 6
Resource Management

6.1 Resource Management....59
6.2 General....59
6.3 Access to Resources....60
6.3.1 Restrictions when using Resources....61
6.3.2 Priority Ceiling Protocol....61
6.3.3 Scheduler as a Resource....62
6.4 Internal Resources....63
6.5 Programming Issues....64
6.5.1 Configuration Options....64
6.5.2 Data Types....64
6.5.3 Run-time Services....64
6.5.4 Resource Definition....65

Chapter 7
Counters, Alarms, Schedule Tables

7.1 General....67
7.2 Counters....68
7.3 Alarms....70
7.4 Alarm Callback....72
7.5 Schedule Table....72
7.6 Programming Issues....73
7.6.1 Configuration Options....73

7.6.2 Data Types ... 73
7.6.3 Counters and Alarm Generation ... 75
7.6.4 Run-time Services ... 76
7.6.5 Constants ... 77

Chapter 8
Events

8.1 Events ... 79
8.2 General ... 79
8.3 Events And Scheduling ... 80
8.4 Programming Issues ... 81
8.4.1 Configuration Options ... 81
8.4.2 Data Types ... 82
8.4.3 Events Definition ... 82
8.4.4 Run-time Services ... 82

Chapter 9
Communication

9.1 IOC Concept ... 85
9.2 Queued and Unqueued Communication ... 86
9.3 Programming Issues ... 87
9.3.1 IOC Definition ... 87
9.3.2 Run-time Services ... 87
9.3.3 Callback Function ... 88
9.3.4 Usage of IOC ... 88

Chapter 10
Error Handling and Special Routines

10.1 Error Handling and Special Routines ... 91
10.2 General ... 91
10.3 Hook Routines ... 91
10.4 Error Handling ... 93
10.4.1 Error Interface ... 93

10.4.1.1 Macros for Errorhook....95
10.4.2 Extended Status....95
10.4.3 Possible Error Reasons....96
10.5 Start-up Routine....96
10.6 Application Modes....97
10.7 System Shutdown....97
10.8 Programming Issues....98
10.8.1 Configuration Options....98

Chapter 11
System Configuration

11.1 System Configuration....101
11.2 General....101
11.3 Application Configuration File....102
11.4 OILConcept....102
11.4.1 OIL File....102
11.4.2 OIL Format....103
11.4.3 Implementation Definition....103
11.4.3.1 Implementation Definition Grammar....103
11.4.4 Application Definition....105
11.4.4.1 Object Definition....105
11.4.4.2 Include Directive....106
11.4.4.3 Comments....107
11.4.4.4 File Structure....107
11.4.5 Configuration File Rules....107

Chapter 12
System Objects Definition

12.1 System Objects Definition....109
12.2 General....109
12.2.1 OIL Attributes Names Mapping to Xml Configuration....110

12.3 OS Definition....110
12.3.1 Global System Attributes....111
12.3.2 Multi-Core specific Attributes....113
12.3.3 Memory Related Attributes....113
12.3.4 CPU Related Attributes....115
12.3.5 Stack Related Attributes....120
12.3.6 Hook Routines Related Attributes....121
12.4 OS-Application Definition....123
12.4.1 Attributes....123
12.4.1.1 Attributes....126
12.4.2 Task Definition....128
12.4.2.1 Attributes....129
12.4.3 ISR Definition....132
12.4.3.1 Attributes....132
12.4.4 Schedule Table Definition....133
12.4.4.1 Attributes....134
12.4.5 Resource Definition....136
12.4.6 Event Definition....137
12.4.6.1 Attributes....137
12.4.7 Counter Definition....138
12.4.7.1 Attributes....138
12.4.8 Alarm Definition....140
12.4.8.1 Attributes....141
12.4.9 Inter-OS-application Communicator (IOC) Definition....143
12.4.9.1 Attributes....143
12.4.10 Application Modes Definition....146

Chapter 13
Building of Application

13.1 Building of Application....149

13.2 Application Structure 149
13.3 Action Sequence to Build Application 150
13.3.1 Application Configuration 151
13.3.2 Source Files 154
13.3.3 Compiling and Linking 155
13.3.4 OS Object Files Dependency 156

Chapter 14
ARM Architecture Platform-Specific Features

14.1 ARM Architecture Platform-Specific Features 159
14.2 Compiler-specific Features 159
14.2.1 Used Library Functions 159
14.2.2 Compiler Issues 159
14.2.3 Options for GreenHills MULTI 161
14.2.3.1 Assembler Options 162
14.2.3.2 Linker Options 163
14.2.4 Options for Linaro ARM EABI 163
14.2.4.1 Assembler Options 164
14.2.4.2 Linker Options 165
14.2.5 Options for IAR ARM EABI 165
14.2.5.1 Assembler Options 165
14.2.5.2 Linker Options 166
14.2.6 Linker Command File 166
14.2.7 Limitations 167
14.2.7.1 General Limitations 167
14.2.7.2 S32K Specific Limitations 167
14.2.8 S32K Specific Features 167
14.2.8.1 Programming Model 168
14.2.8.2 Target Hardware Initialization 168
14.2.8.3 Interrupt Handling and Vector Table 168

14.2.8.4 Using Floating Point ... 168
14.2.8.5 Nested Interrupts ... 169
14.2.8.6 Interrupt Dispatcher ... 170
14.2.9 Timer Hardware ... 170

Chapter 15
Application Troubleshooting

15.1 Application Troubleshooting ... 173
15.2 System Generation ... 173
15.3 Known Problems ... 173
15.3.1 Troubleshooting ... 173

Chapter 16
System Services

16.1 System Services ... 175
16.2 General ... 175
16.3 Autosar OS-Application Services ... 176
16.3.1 Data Types ... 176
16.3.2 System Macros for Memory Access ... 177
16.3.3 Constants ... 177
16.3.4 States of OS-Applications ... 177
16.3.5 GetApplicationID ... 177
16.4 CheckObjectAccess ... 178
16.5 CheckObjectOwnership ... 178
16.6 TerminateApplication ... 178
16.7 AllowAccess ... 179
16.8 GetApplicationState ... 180
16.9 Task Management Services ... 180
16.9.1 Data Types ... 180
16.9.2 Constants ... 181
16.9.3 Conventions ... 181

16.9.4 Task Declaration ... 181
16.9.5 ActivateTask ... 181
16.9.6 TerminateTask ... 182
16.9.7 ChainTask ... 182
16.9.8 Schedule ... 183
16.9.9 GetTaskID ... 183
16.9.10 GetTaskState ... 184
16.10 ISR Management Services ... 184
16.10.1 Data Types ... 185
16.10.2 Constants ... 185
16.10.3 Conventions ... 185
16.10.4 ISR Declaration ... 185
16.10.5 EnableAllInterrupts ... 185
16.10.6 DisableAllInterrupts ... 186
16.10.7 ResumeAllInterrupts ... 186
16.10.8 SuspendAllInterrupts ... 187
16.10.9 ResumeOSInterrupts ... 187
16.10.10 SuspendOSInterrupts ... 188
16.10.11 GetISRID ... 188
16.10.12 DisableInterruptSource ... 189
16.10.13 EnableInterruptSource ... 189
16.11 Resource Management Services ... 189
16.11.1 Data Types ... 190
16.11.2 Constants ... 190
16.11.3 Resource Declaration ... 190
16.11.4 GetResource ... 190
16.11.5 ReleaseResource ... 191
16.12 Event Management Services ... 191
16.12.1 Data Types ... 192

16.12.2 Event Declaration....192
16.12.3 SetEvent....192
16.12.4 ClearEvent....193
16.12.5 GetEvent....193
16.12.6 WaitEvent....194
16.13 Counter Management Services....194
16.13.1 Data Types And Identifiers....194
16.13.2 System Macros for Time Conversion....195
16.13.3 Constants....195
16.13.4 Counter Declaration....196
16.13.5 InitCounter....197
16.13.6 IncrementCounter....197
16.13.7 GetCounterValue....198
16.13.8 GetElapsedValue....198
16.13.9 GetCounterInfo....199
16.14 Alarm Management Services....199
16.14.1 Data Types and Identifiers....199
16.14.2 Constants....200
16.14.3 Alarm Declaration....200
16.14.4 GetAlarmBase....200
16.14.5 GetAlarm....201
16.14.6 SetRelAlarm....201
16.14.7 SetAbsAlarm....202
16.14.8 CancelAlarm....203
16.15 Scheduletable Management Services....203
16.15.1 Data Types....204
16.15.2 Constants....204
16.15.3 StartScheduleTableRel....204
16.15.4 StartScheduleTableAbs....205

16.15.5 StartScheduleTableSynchron....205
16.15.6 StopScheduleTable....205
16.15.7 NextScheduleTable....206
16.15.8 SyncScheduleTable....206
16.15.9 SetScheduleTableAsync....207
16.15.10 GetScheduleTableStatus....207
16.16 Communication Management Services....208
16.16.1 Data Types....208
16.16.2 Constants....208
16.16.3 IocSend....209
16.16.4 IocWrite....209
16.16.5 IocReceive....210
16.16.6 IocRead....210
16.16.7 IocEmptyQueue....211
16.17 Debugging Services....211
16.17.1 GetRunningStackUsage....211
16.17.2 GetStackUsage....212
16.18 Operating System Execution Control....212
16.18.1 Data Types....212
16.18.2 Constants....213
16.18.3 GetActiveApplicationMode....214
16.18.4 StartOS....214
16.18.5 ShutdownOS....214
16.18.6 OS System Timers control....215
16.18.7 Hook Routines....216
16.18.7.1 ProtectionHook....216
16.18.7.2 ErrorHook....216
16.18.7.3 PreTaskHook....217
16.18.7.4 PostTaskHook....217

16.18.7.5 PreIsrHook 218
16.18.7.6 PostIsrHook 218
16.18.7.7 StartupHook 218
16.18.7.8 ShutdownHook 219

Chapter 17
Debugging Application

17.1 Debugging Application 221
17.2 General 221
17.2.1 Extended Status 221
17.2.2 ORTI 221
17.3 Using OS Extended Status for Debugging 223
17.4 Context Switch Monitoring 224
17.5 ORTI Features 224
17.5.1 ORTI Trace Interfaces 224
17.5.1.1 Hardware-specific implementation 226
17.5.2 ORTI Breakpoint Interface 226
17.6 Stack Debugging Support 230
17.6.1 Stack Overflow Checking 230

Chapter 18
Sample Application

18.1 Sample Application 231
18.2 Description 231
18.3 Source Files 233

Chapter 19
System Service Timing

19.1 General Notes 235

Chapter 20
Memory Requirements

20.1 Memory Requirements 237
20.2 NXP Autosar OS Memory Usage 237

Chapter 21
System Generation Error Messages

21.1 General Notes....239
21.2 Severity Level....239
21.3 Error Message Format....239
21.4 List of Messages....240

Chapter 1
Introduction

1.1 Introduction

This Technical Reference describes NXP AUTOSAR OS/S32K, the implementation of the AUTOSAR[1] Operating System (AUTOSAR OS) for ARM Architecture providing high speed performance and low RAM usage. The AUTOSAR OS specification is based on the OSEK OS specification with addition of more protection facilities and ability to integrate into overall AUTOSAR framework. The NXP AUTOSAR OS/S32K is intended to be used inside AUTOSAR framework, however it can be used without it to build the applications in the traditional OSEK manner.

Operating System Architecture chapter gives a high level description of the OS architecture and presents OS Conformance Classes.

Task Management chapter explains the task concept in AUTOSAR and all other questions related to tasks.

Scheduler chapter provides a description of scheduling policies in AUTOSAR OS.

Interrupt Processing chapter highlights the AUTOSAR approach to interrupt handling.

Resource Management chapter describes resource management and task coordination by resources.

Counters, Alarms, Schedule Tables chapter describes usage of these control mechanisms in AUTOSAR OS.

Events chapter is devoted to event management and task coordination by events.

Communication chapter describes a IOC concept in AUTOSAR OS and its usage.

Error Handling and Special Routines chapter describes the support provided for the user to debug an application and handle errors.

1. The term AUTOSAR means 'Automotive Open System Architecture'

System Configuration chapter describes possible AUTOSAR OS versions, configuration options and the configuration mechanism.

System Objects Definition chapter describes the objects controlled by the Operating System: tasks, resources, alarms, IOCs, counters, ISRs and even the OS itself are considered as system objects.

Building of Application chapter contains information on how to build a user's application using the AUTOSAR OS.

ARM Architecture Platform-Specific Features chapter describes special AUTOSAR OS features for different MCU types and issues connected with porting applications to these MCUs.

Application Troubleshooting chapter describes special AUTOSAR OS features for different MCU types and issues connected with porting applications to these MCUs.

System Services chapter provides a detailed description for all AUTOSAR Operating System run-time services, with appropriate examples.

Debugging Application chapter provides information about preparation of all data required for the OSEK aware debugger to display information about an application in AUTOSAR terms.

Sample Application appendix contains the text and listing of a sample customer application developed using the AUTOSAR OS.

System Services appendix provides information about OS services execution time.

The Introduction chapter consists of the following sections:

- AUTOSAR OS Overview
- Typographical Conventions
- References
- Definitions, Acronyms and Abbreviations

1.2 AUTOSAR OS Overview

The AUTOSAR Operating System is a real-time operating system which conforms to the *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification.

The AUTOSAR OS meets the following requirements:

- The OS is fully configured and statically scaled;
- The OS is intended to be used as integrated part of AUTOSAR;

- The OS performance parameters are well known;
- The OS is written in correspondence with MISRA C Guidelines, all deviations from MISRA are documented.

A wide range of scalability, a set of system services, various scheduling mechanisms, and convenient configuration features make the AUTOSAR Operating System feasible for a broad spectrum of applications and hardware platforms.

The AUTOSAR OS provides a pool of different services and processing mechanisms for task management and synchronization, data exchange, resource management, and interrupt handling. The following features are provided for the user:

Task Management

- Activation and termination of tasks;
- Management of task states, task switch.

Scheduling Policies

- Full-, non-, and mixed-preemptive scheduling techniques.

Event Control

Event Control for task synchronization.

Interrupt Management

- Services for disabling/enabling all interrupts;
- Services for disabling/enabling interrupts of category 2;

Resource Management

- Control of mutually exclusive access to jointly used resources or devices, or for control of a program flow.

Communication

- Data exchange between tasks and/or ISRs.

Counter and Alarm Management

- The counter management provides services for execution of recurring events;

- The alarm management is based on the counter management. The alarm management allows the user to link task activation or event setting to a certain counter value. These alarms can be defined as either single (one-shoot) or cyclic alarms. Expiration of a preset relative counter value, or the fact that a preset absolute counter value is reached, results in activation of a task, or setting a task event;
- ScheduleTable enables periodic activations of tasks in accordance with a static defined schedule. NXP AUTOSAR OS/S32K supports Global Time Synchronization in all Scalability Classes.

Error Treatment

- Mechanisms supporting the user in case of various errors, such as calling OS services with wrong parameters and in the wrong context.

ORTI Subsystem

- The ORTI provides an interface to Operating System run-time data for 'OSEK aware' debuggers.

The AUTOSAR Operating System is scaled in two ways: either by changing the set of system services or through the so-called Conformance and Scalability Classes. They are available to meet different requirements concerning the OS functionality and capability. The Conformance Classes differ in the number of services they provide and OS capabilities. The Scalability Classes differ in the protection mechanisms provided by the OS (see Scalability Classes). The classes are based on one another in upwardly compatible fashion. (see Conformance Classes)

The AUTOSAR OS is built according to the user's configuration instructions while the system is generated. Both system and application parameters are configured statically. Therefore, the special tool called the System Generator is used for this purpose. Special statements are designed to tune any parameter. The user should only edit the definition file, run the System Generator and then assemble the resulting files and the application ones. Thus, the user can adapt the Operating System for the desired task and the target hardware. The OS cannot be modified later at the run time.

1.3 Typographical Conventions

This Technical Reference employs the following typographical conventions:

Boldface type

Bold is used for important terms, notes and warnings.

Italics

Italics are used for all AUTOSAR names of directives, macros, constants, routines and variables.

`Courier` font

The courier typeface is used for code examples in the text.

1.4 References

Table 1-1. References

#	Title	Version
1	AUTOSAR Specification of Operating System	V5.0.0 R4.0 Rev 3
2	Specification of ECU Configuration Parameters (AUTOSAR_EcucParamDef.arxml)	V3.2.0 R4.0 Rev 3
3	OSEK/VDX System Generation OIL: OSEK Implementation Language	v.2.5, 01 July 2004
4	OSEK/VDX Operating System	v.2.2.3, 17 Feb 2005
5	ORTI: OSEK Run Time Interface	v.2.0 (c), 23 June 1999
6	OSEK/VDX OSEK Run Time Interface (ORTI), Part A: Language Specification	v. 2.2, 14 Nov 2005
7	OSEK/VDX OSEK Run Time Interface (ORTI), Part B: OSEK Objects and Attributes	v. 2.2, 25 Nov 2005
8	AUTOSAR Specification of Memory Mapping	v.1.4.0 R.4.0 rev. 3
9	Specification of Standard Types	v.1.3.0 R4.0 rev. 1

1.5 Definitions, Acronyms and Abbreviations

The following acronyms and abbreviation are used in this Technical Reference.

Table 1-2. Definitions, Acronyms and Abbreviations

API	Application Program Interface (a set of data types and functions)
AUTOSAR	Automotive Open System Architecture
BCC	Basic Conformance Class, a defined set of functionality in OSEK, for which waiting state of tasks is not permitted
BT	Basic task (the task which has no waiting state)
CCCB	OSEK Conformance Communication Class B
CPU	Central Processor Unit
ECC	Extended Conformance Class, a defined set of functionality in OSEK, for which waiting state of tasks is permitted
ET	Extended Task (the task which may have waiting state)

Table continues on the next page...

Table 1-2. Definitions, Acronyms and Abbreviations (continued)

HW	Hardware
ID	Identifier, an abstract identifier of a system object
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MCU	Microcontroller Unit
MPU	Memory Protection Unit
N/A	Not applicable
OIL	OSEK Implementation Language
ORTI	OSEK Run Time Interface
OS	Operating System
OSEK	Open systems and the corresponding interfaces for automotive electronics (in German)
RAM	Random Access Memory
ROM	Read Only Memory
SC	Scalability Class, defines OS protection level
SG, SysGen	System Generator Utility
SW	Software

Chapter 2
Operating System Architecture

2.1 Operating System Architecture

This chapter gives a high level description of the OS architecture and presents the OS Conformance Classes.

This chapter consists of the following sections:

- Processing Levels
- Conformance Classes
- OS and Application Stacks
- AUTOSAR OS Overall Architecture
- Application Program Interface

2.2 Processing Levels

The AUTOSAR Operating System provides a pool of different services and processing mechanisms. It serves as a basis for application programs which are independent of each other, and provides their environment on a processor. The AUTOSAR OS enables controlled real-time execution of several processes which virtually run in parallel.

The AUTOSAR Operating System provides a defined set of interfaces for the user. These interfaces are used by entities competing for the CPU.

There are two types of entities:

- Interrupts (service routines managed by the Operating System);
- Tasks (basic tasks and extended tasks).

The highest processing priority is assigned to the interrupt level, where interrupt service routines (ISR) are executed. The interrupt services may call a number of operating system services. The processing level of the operating system has the priority immediately below

the former one. This is the level on which the operating system works: task management procedures, scheduler and system services. Immediately below there is the task level on which the application software is executed. Tasks are executed according to their user assigned priority. A distinction is made between the management of tasks with and without *waiting* state (*Extended* and *Basic Tasks*, see Task Concept).

The following set of priority rules has been established:

- interrupts have precedence over tasks;
- the interrupt priority is defined by specific hardware conditions;
- for the items handled by the OS, bigger numbers refer to higher priorities;
- the task's priority is statically assigned by the user.

The Operating System provides services and ensures compliance with the set of priority rules mentioned above.

2.3 Scalability Classes

In order to customize the operating system to the needs of the user and to take full advantage of the processor features the operating system can be scaled according to the 4 scalability classes SC1 .. SC4.

Table 2-1. AUTOSAR OS Scalability Classes

Feature	SC1	SC2	SC3	SC4
OSEK OS (all conformance classes)	*	*	*	*
Counter Interface	*	*	*	*
Schedule Tables	*	*	*	*
Stack Monitoring	*	*	*	*
Timing Protection		*		*
Global Time and Synchronization Support		*		*
Memory Protection			*	*
OS-Applications	*	*	*	*
Service Protection			*	*
CallTrustedFunction			*	*

The NXP AUTOSAR OS/S32K v.4.0.3 supports SC1 class with addition of Service Protection in *Extended Status* for SC1 and Global Time Synchronization for ScheduleTable.

2.4 Conformance Classes

Various requirements of the application software for the system, and various capabilities of a specific system (e.g. processor type, amount of memory) require different features of the operating system. These operating system features are described as *Conformance Classes* (CC). They differ in the number of services provided, their capabilities and different types of tasks.

The Conformance classes were created to support the following objectives:

- providing convenient groups of operating system features for easier understanding and discussion of the AUTOSAR operating system.
- allowing partial implementations along pre-defined lines. These partial implementations may be certified as AUTOSAR compliant.
- creating an upgrade path from the classes of less functionality to the classes of higher functionality with no changes to the application using AUTOSAR related features.

The required Conformance Class is selected by the user at the system generation time and cannot be changed during execution.

Definition of the functionalities provided by each Conformance Class depends on the properties of the tasks and the scheduling behavior. As the task properties (Basic or Extended, see Task Concept) have a distinct influence on CC, they also assume part of their names. There are Basic-CC and Extended-CC, and each group can have various 'derivatives'.

The Conformance classes are determined by the following attributes:

- Multiply requesting of task activation - not supported by NXP AUTOSAR OS;
- Task types (see Task Concept);
- Number of tasks per priority.

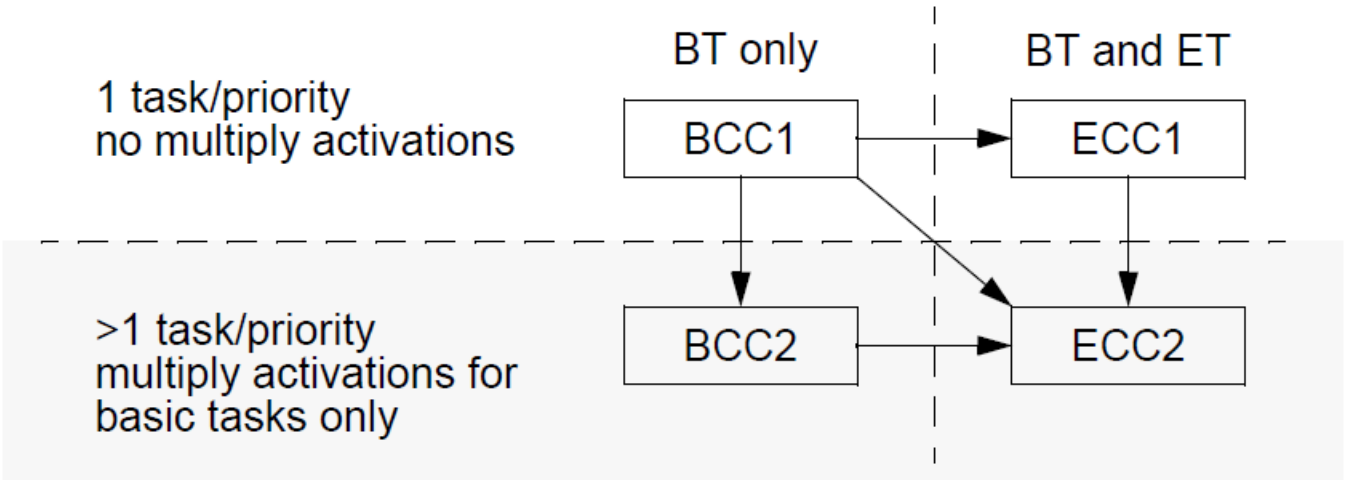

Figure 2-1. Restricted Upward Compatibility for Conformance Classes

The OSEK OS specification defines the following Conformance Classes: BCC1, BCC2, ECC1, ECC2. The NXP AUTOSAR OS does not support multiple activation and therefore it doesn't have BCC2 and ECC2 classes.

The NXP AUTOSAR OS supports the following Conformance Classes:

- BCC1 - only Basic tasks, limited to one activation request per task and one task per priority, and all tasks have different priorities;
- ECC1 - like BCC1, plus Extended tasks.

Table 2-2 indicates the minimum resources to which an application may resort, determined for each Conformance Class in the OSEK OS.

Table 2-2. OSEK OS Conformance Classes

	BCC1	BCC2	ECC1	ECC2
Multiple activation of tasks	no	yes	BT: no ET: no	BT: yes ET: no
Number of tasks which are not in *suspended* state	>=8		>= 16, any combination of BT/ET	
Number of tasks per priority	1	>1	1 (both BT/ ET)	>1 (both BT/ ET)
Number of events per task	-		BT: no ET: >= 8	
Number of task priorities	>=8		>=16	
Resources	only Scheduler	>= 8 resources (including Scheduler)		
Internal Resources	>=2			
Alarm	>= 1 single or cyclic alarm			

Table continues on the next page...

Table 2-2. OSEK OS Conformance Classes (continued)

	BCC1	BCC2	ECC1	ECC2
Messages	possible			

The system configuration option *CC* (specified by the user) defines the class of the overall system. In the NXP AUTOSAR OS this option can have the values *BCC1* and *ECC1* or it can be set to *AUTO* (see OS Definition).

Maximal numbers of OS system objects are indicated in the table below.

Table 2-3. Maximal System Resources

Number of OS-Applications	8
Number of task's priorities	64
Number of tasks which are not in suspended state	64
Number of events per task	32
Number of resources (including RES_SCHEDULER)	2047
Number of Application Modes	8
Number of all other OS objects	2047

Note that actual number of OS objects might be limited by amount of available memory.

2.5 OS and Application Stacks

The NXP AUTOSAR OS uses different approaches to stack definition and usage depending on configuration.

In BCC1 within SC1 class the OS uses the "Single Stack" (the main application stack) for all Tasks and all ISRs, i.e. OS never switches the stack and everything is executed on the stack where StartOS was called.

NOTE

The *main* application stack is configured by the User, its size is defined in the linker command file. In the linker command file specific linker variables **OS_MAIN_STACK_MAX** and **OS_MAIN_STACK_MIN** shall be defined for main stack. Linker variables are described in Linker Command File.

NOTE

If stack monitoring is configured by the user and measurement of MAIN stack usage is needed the user needs to fill the MAIN stack with the default pattern (0x55555555) or the user defined pattern. For correct measurement this must be done before main() is called. The reason the MAIN stack is not filled by OS is because this stack is not under OS control. The OS Task and ISR stacks are filled by OS before any Task/ISR are executed. In the same way the user must fill the main stack before main() is called.

In ECC1 within SC1 class the OS uses the "Single Stack" for all Basic Tasks, common ISR stack for all ISRs of category 2 and OS Timers ISRs and separate stacks for each Extended Task. ISR stack size is configured by "IsrStackSize", Extended Task stack size is configured by "STACKSIZE" for each task. These stacks are located in .osstack section.

NOTE

Stacks for ISRs category 2 and Extended Tasks stacks are configured statically in all classes.

In SC1 class NXP AUTOSAR OS uses the main application stack for a 'Single Stack'.

The context of preempted task is saved on the task stack. The User shall reserve the additional space for the task context on the Extended task stack and on the main application stack for Basic Tasks.

The ISRs of category 1 are always executed on the stack of the preempted Task/ISR.

2.6 AUTOSAR OS Overall Architecture

The AUTOSAR OS is a real-time operating system which is executed within a single electronic control unit. It provides local services for the user's tasks. The AUTOSAR OS consists of the following components:

- ***Scheduler*** controls the allocation of the CPU for different tasks;
- ***Task management*** provides operations with tasks;
- ***ISR management*** provides entry/exit frames for interrupt service routines and supports CPU interrupt level manipulation;
- ***Resource management*** supports a special kind of semaphore for mutually exclusive access to shared resources;
- ***Local communication*** provides message exchange between tasks;

- ***Counter management*** provides operations on objects like timers and incremental counters;
- ***Alarm management*** links tasks and counters;
- ***Service protection*** subsystems;
- ***Error handlers*** handle the user's application errors and internal errors, and provide recovery from the error conditions;
- ***Hook routines*** provide additional debugging features;
- ***System start-up*** initializes data and starts the execution of applications;
- ***System timers*** provides implementation-independent time management.

As it is shown in Table 2-2, the Conformance Classes, in general, differ in the degree of services provided for the task management and scheduling (the number of tasks per priority, multiple requesting, Basic/Extended Tasks). In higher CC an advanced functionality is added for the resource management and event management only. But even in BCC1 the user is provided with almost all the AUTOSAR OS service mechanisms.

The AUTOSAR Operating System is not scaled through the Classes only, but it also has various extentions which can be used in any Scalability/ Conformance Classes. These extentions affect memory requirements and overall system performance. The extentions can be turned on or off with the help of the corresponding system configuration options. They all are described in System Objects Definition.

Since the AUTOSAR Operating System is fully statically configured, the configuration process is supported by the System Generator (SG). This is a command-line utility, which processes system generation statements defined by the user in a special file. These statements fully describe the required system features and application object's parameters. The SG produces a header file (Os_prop.h) which is used for system compilation and C-code files to be compiled together with the other user's source code. The produced code consists of C-language definitions and declarations of data as well as C-preprocessor directives. See System Configuration and Building of Application for details on system generation.

2.7 Application Program Interface

The AUTOSAR Operating System establishes the Application Program Interface (API) which must be used for all the user actions connected with system calls and system objects. This API defines the data types used by the system, the syntax of all run-time service calls, declarations and definitions of the system.

The AUTOSAR OS data types are described in the subsections dedicated to the corresponding mechanisms. The syntax of system calls and system configuration statements is described briefly in the corresponding subsections and in detail in System Objects Definition and System Services.

NOTE

The user's source code shall strictly correspond to the rules stated in this Technical Reference.

The AUTOSAR OS may be compiled in *Extended Status*. It means that an additional check is made within all OS activities and extended return codes are returned by all the OS services to indicate errors, if any. See System Services and Error Handling about Extended Status return values. In order to provide the Extended Status in the system, the configuration option *STATUS* must be set to *EXTENDED* at the configuration stage.

The NXP AUTOSAR OS provides support for the 'OSEK aware' debuggers by means of the OSEK Run Time Interface (ORTI). See Debugging Application and Global System Attributes for details.

2.8 AUTOSAR OS Protection features

2.8.1 Service Protection

As OS-Applications interact with the OS through services, it is essential that the service calls will not corrupt the OS itself. Service Protection guards against such corruption at runtime. The AUTOSAR OS provides it also in SC1 when configured with *EXTENDED* Status.

There are a number of cases considered within Service Protection: An OSApplication makes an API call

1. with an invalid handle or out of range value.
2. in the wrong context, e.g. calling ActivateTask() in the StartupHook.
3. or fails to make an API call that results in the OSEK OS being left in an undefined state, e.g. it terminates without a ReleaseResource() call
4. that impacts on the behavior of every other OS-Application in the system, e.g. ShutdownOS()
5. to manipulate OS objects that belong to another OS-Application (to which it does not have the necessary permissions), e.g. an OS-Application tries to execute ActivateTask() on a task it does not own.

All this cases are handled by AUTOSAR OS Service protection. The codes returned in case of error are defined in each service description in the chapter System Services.

The table below indicates the allowed context for the OS service calls.

Table 2-4. Allowed Services Contexts

System service	Task	ISR Cat.1	ISR Cat.2	ErrorHook	PreTaskHook	PostTaskHook	StartupHook	ShutdownHook	Alarm Callback	ProtectionHook
ActivateTask	*		*							
TerminateTask	*									
ChainTask	*									
Schedule	*									
GetTaskID	*		*	*	*	*				*
GetTaskState	*		*	*	*	*				
EnableAllInterrupts	*	*	*	*	*	*	*	*	*	*
DisableAllInterrupts	*	*	*	*	*	*	*	*	*	*
ResumeAllInterrupts	*	*	*	*	*	*	*	*	*	*
SuspendAllInterrupts	*	*	*	*	*	*	*	*	*	*
ResumeOSInterrupts	*	*	*	*	*	*	*	*	*	*
SuspendOSInterrupts	*	*	*	*	*	*	*	*	*	*
GetResource	*		*							
ReleaseResource	*		*							
SetEvent	*		*							
ClearEvent	*									
GetEvent	*		*	*	*	*				
WaitEvent	*									
GetAlarmBase	*		*	*	*	*				
GetAlarm	*		*	*	*	*				
SetRelAlarm	*		*							
SetAbsAlarm	*		*							
CancelAlarm	*		*							
GetActiveApplicationMode	*		*	*	*	*	*	*		
StartOS										
ShutdownOS	*		*	*			*			
GetISRID	*		*	*						*
GetApplicationID	*		*	*	*	*	*	*		*
GetApplicationState	*		*	*	*	*	*	*		*
CheckObjectAccess	*		*	*						*
CheckObjectOwnership	*		*	*						*

Table continues on the next page...

Table 2-4. Allowed Services Contexts (continued)

System service	Task	ISR Cat.1	ISR Cat.2	ErrorHook	PreTaskHook	PostTaskHook	StartupHook	ShutdownHook	Alarm Callback	ProtectionHook
TerminateApplication	*		*	*						
StartScheduleTableRel	*		*							
StartScheduleTableAbs	*		*							
StopScheduleTable	*		*							
NextScheduleTable	*		*							
SyncScheduleTable	*		*							
StartScheduleTableSynchron	*		*							
GetScheduleTableStatus	*		*							
SetScheduleTableAsync	*		*							
IncrementCounter	*		*							
DisableInterruptSource	*		*							
EnableInterruptSource	*		*							
GetCounterInfo [1]	*		*							
InitCounter [1]	*		*							
GetElapsedValue	*		*							
GetCounterValue	*		*							
GetRunningStackUsage	*		*	*	*	*				
GetStackUsage	*		*	*	*	*				
IocSend_<IocId>[_<SenderId>]	*		*							
IocWrite_<IocId>[_<SenderId>]	*		*							
IocReceive_<IocId>i	*		*							
IocRead_<IocId>	*		*							
IocEmptyQueue_<IocId>	*		*							

1. this service is not specified inAUTOSAR OS v.4.0 specification, it is NXP OS extension of the AUTOSAR OS.

The NXP AUTOSAR OS checks the allowed context in EXTENDED status for all Scalability Classes.

2.8.2 Task or ISRs with Incorrect Behavior

The OS dispatcher handles the cases when the Task returns w/o call to TerminateTask and when Task/ISR does not release Resource or fails to call Resume*Interrupts() after Suspend*Interrupts(). In this cases the OS does required cleanup, releases locked objects and calls ErrorHook if it is configured.

Chapter 3
Task Management

3.1 Task Management

This chapter describes the task concept of AUTOSAR and all other questions related to tasks.

This chapter consists of the following sections:

- Task Concept
- Task Priorities
- Tasks Stacks
- Programming Issues

3.2 Task Concept

Complex control software can be conveniently subdivided into parts executed according to their real-time requirements. These parts can be implemented by means of tasks. The task provides the framework for execution of functions. The Operating System provides parallel and asynchronous task execution organization by the scheduler.

Two different task concepts are provided by the AUTOSAR OS:

- Basic Tasks (BT);
- Extended Tasks (ET).

The Basic Tasks release the processor only if:

- they are being terminated,
- the AUTOSAR OS is executing higher-priority tasks, or
- interrupt occurred.

The Extended Tasks differ from the Basic Tasks by being allowed using additional operating system services which may result in *waiting* state. *Waiting* state allows the processor to be freed and reassigned to a lowerpriority task without the necessity to terminate the Extended Task.

The task type is determined automatically. If a TASK object has a reference to EVENT, the task is considered to be Extended.

Both types of tasks have their advantages which must be compared in context of application requirements. They both are justified and supported by the AUTOSAR operating system.

Every task has a set of related data: task description data located in ROM and task state variables in RAM. Also, every extended task has its own stack assigned.

Every running task is represented by its run-time context. This refers to the CPU general purpose registers and some special registers. When the task is interrupted or preempted by another task, the run-time context is saved.

The task has several states since the processor can execute only one instruction of the task at any time, but at the same time several tasks may compete for the processor. The AUTOSAR OS is responsible for saving and restoring the task context in conjunction with state transitions whenever necessary.

3.2.1 Basic Tasks

The state model for the Basic Tasks is nearly identical to the one for the Extended Tasks. The only exception is the absence of *waiting* state.

- *running*

  In running state the CPU is assigned to the task so that its instructions can be executed. Only one task can be in this state at the same point in time, while all the other states can be adopted simultaneously by several tasks.

- *ready*

  All functional prerequisites for transition into running state are met, and the task only waits for allocation of the processor. The scheduler decides which of the ready tasks is executed next.

- *suspended*

In suspended state the task is passive and does not occupy any resources, merely ROM.

Table 3-1. States and Status Transitions for a Basic Task

Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the *ready* list by a system service.
start	ready	running	A *ready* task selected by the scheduler is executed.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into *ready* state.
terminate	running	suspended	The *running* task causes its transition into *suspended* state by a system service.

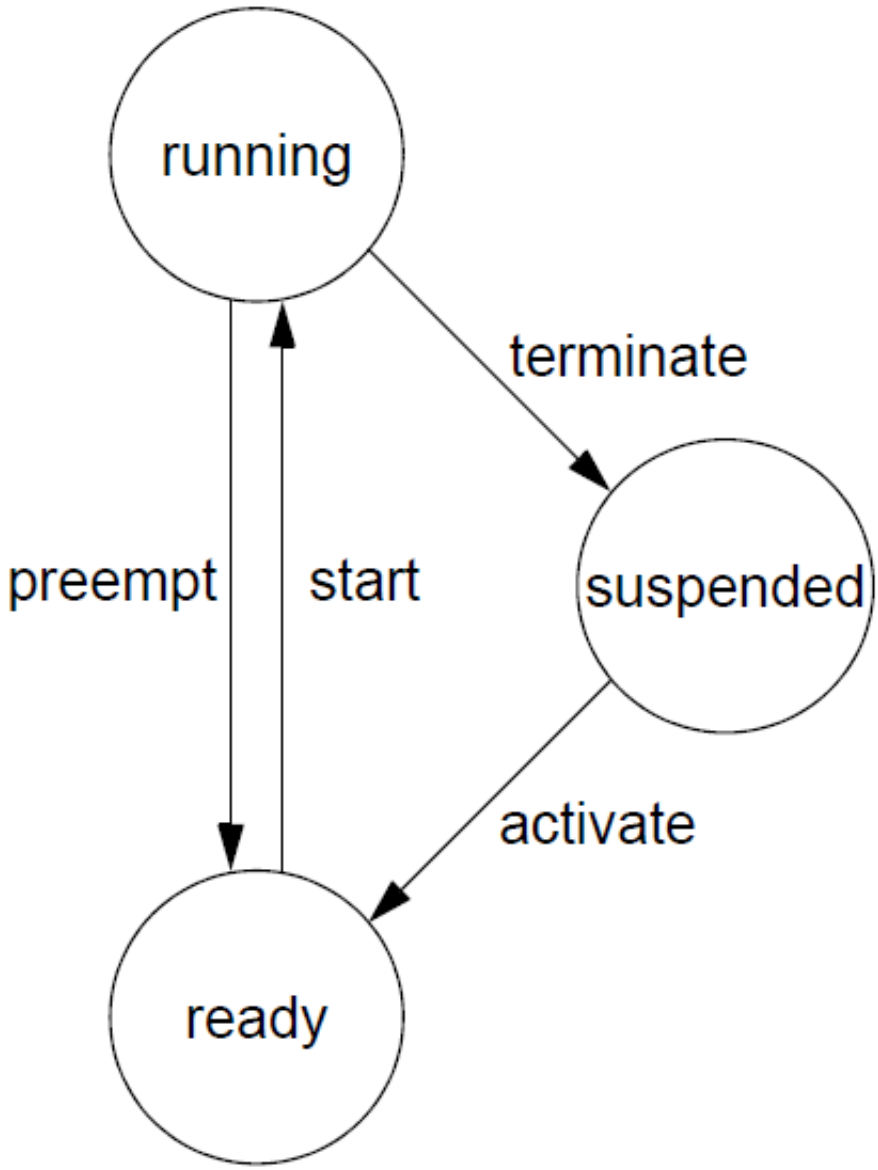

Figure 3-1. Status Model with Task Transitions for a Basic Task

3.2.2 Extended Tasks

The Extended Tasks have four task states:

- *running*

  In running state the CPU is assigned to the task so that its instructions can be executed. Only one task can be in this state at the same point in time, while all the other states can be adopted simultaneously by several tasks.

- *ready*

  All functional prerequisites for transition into running state are met, and the task only waits for allocation of the processor. The scheduler decides which of the ready tasks is executed next.

- *waiting*

  A task cannot be executed (any longer), because it has to wait for at least one event (see Events).

- *suspended*

  In suspended state the task is passive and does not occupy any resources, merely ROM.

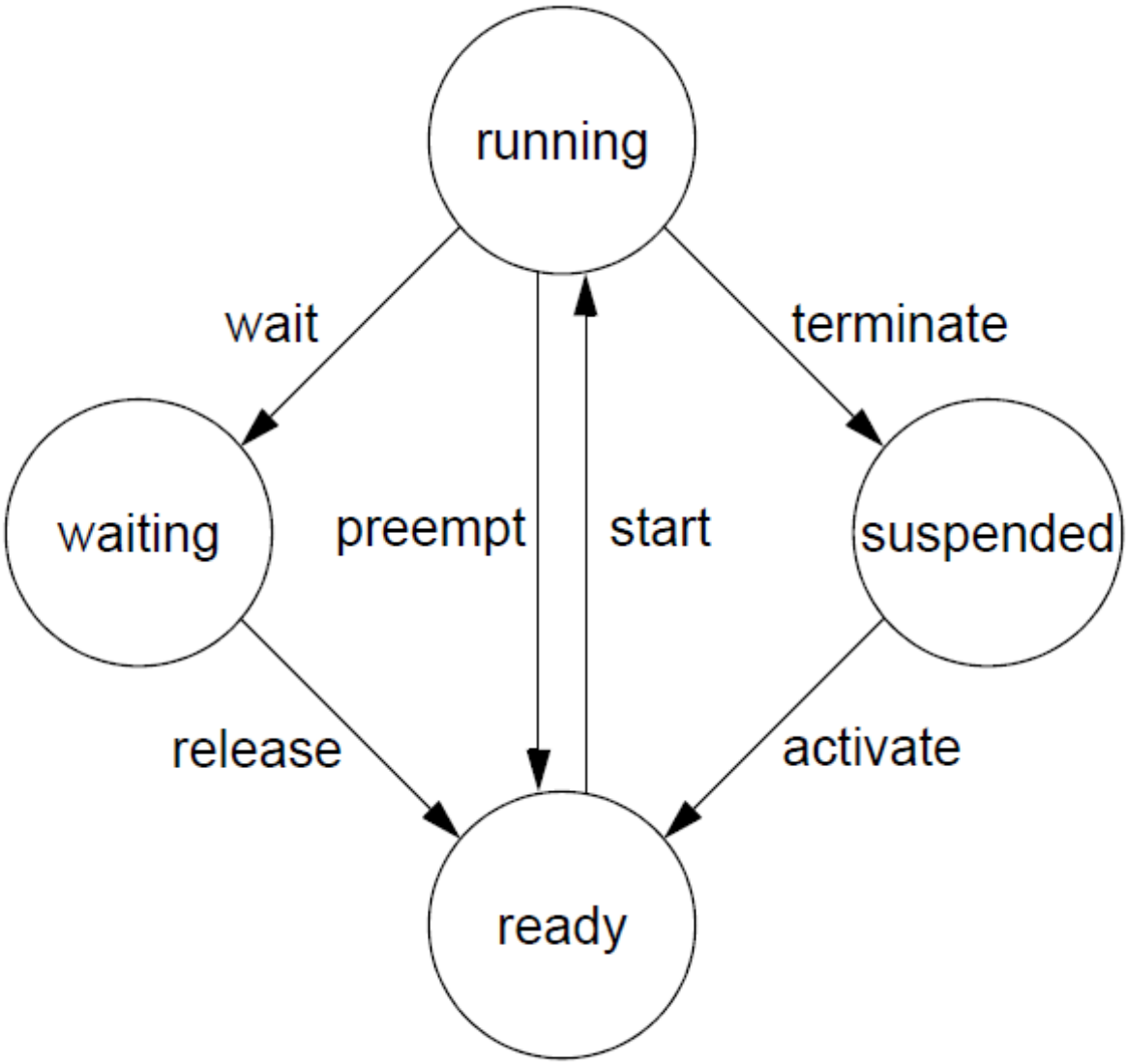

Figure 3-2. Status Model with Task Transitions for an Extended Task

Table 3-2. States and Status Transitions for an Extended Task

Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the *ready* list by a system service.
start	ready	running	A *ready* task selected by the scheduler is executed.
wait	running	waiting	To be able to continue an operation, the *running* task requires an event. It causes its transition into *waiting* state by using a system service.

Table continues on the next page...

Table 3-2. States and Status Transitions for an Extended Task (continued)

Transition	Former state	New state	Description
release	waiting	ready	Events have occurred which a task has been waiting for.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into *ready* state.
terminate	running	suspended	The *running* task causes its transition into suspended state by a system service.

Termination of tasks is possible only if the task terminates itself ('selftermination').

There is no provision for a direct transition from *suspended* into *waiting* state. This transition is redundant and would make the scheduler more complicated. *Waiting* state is not directly entered from *suspended* state since the task starts and explicitly enters *waiting* state on its own.

3.3 Task Priorities

The AUTOSAR OS specifies the value 0 as the lowest task priority in the operating system. Accordingly bigger numbers define higher task priorities.

The task priorities defined in OIL are not intersected with ISR priorities. Interrupts have a separate priority scale. All task priorities are lower than any ISR priorities and the scheduler priority.

In the AUTOSAR OS the priority is statically assigned to each task and it cannot be changed at run-time. A dynamic priority management is not supported. However, in particular cases the operating system can change task priority. In this context, please refer to Priority Ceiling Protocol.

When rescheduling is performed, the scheduler always switches to the task with the highest priority among the *ready* tasks and the *running* one.

3.4 Tasks Stacks

3.4.1 Stack Allocation

In BCC1 within SC1..2 class the OS uses the 'Single Stack' (the *main* application stack) for all Tasks.

In ECC1 within SC1 class the OS uses the 'Single Stack' for all Basic Tasks and separate statically allocated stacks for Extended Tasks.

In all classes each Extended Task has separate statically allocated stack.

The minimal size of the task stack depends on:

- the scheduling policy (non-preemptable or preemptable task);
- the services used by the task;
- the interrupt and error handling policy;
- the processor type.

For more details see OS and Application Stacks.

3.5 Programming Issues

3.5.1 Configuration Options

The following system configuration options affect the task management:

- *STATUS*

  Specifies error checking at run-time.

- *STACKMONITORING*

  Turns on stack overflow runtime checking and stack usage services.

- *Pattern*

  Defines the pattern used to fill stack memory. Possible values are 0..0xFFFFFFFF

- *PatternSize*

  Defines the size of the area to be checked in 32-bit words.

- *CC*

  Specifies the conformance class. If AUTO, the conformance class is defined according to the Tasks definitions.

3.5.2 Data Types

The AUTOSAR OS establishes the following data types for the task management:

- *TaskType*

  The abstract data type for task identification;

- *TaskRefType*

  The data type to refer the variables of the *TaskType* data type. Reference to the *TaskType* variable can be used instead of the *TaskRefType* variable;

- *TaskStateType*

  The data type for the variables for storage of the task state;

- *TaskStateRefType*

  The data type to refer the variables of the *TaskStateType* data type. Reference to the *TaskStateType* variable can be used instead of the *TaskStateRefType* variable.

Only these data types may be used for operations with tasks.

3.5.3 Task Definition

Every task in an application is generated using the *TASK* system generation object with the set of properties in OIL file. These properties define the task behavior and the resource allocation method. Each task property has its own name, and the user defines the task's features by setting the corresponding properties in the task definition. See also System Objects Definition.

The task definition looks like the following:

```
TASK TASKSENDER {
    PRIORITY = 5;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    RESOURCE = MYRESOURCE;
    RESOURCE = SECONDRESOURCE;
    EVENT = MYEVENT;
    STACKSIZE = 256;
    TIMING_PROTECTION = FALSE;
};
```

A description of possible task properties is indicated in Table 3-3

Table 3-3. Task Properties

Object Parameters	Possible Values	Description
Standard Attributes		
PRIORITY	integer [0...0xFFFFFFFF]	Defines the task priority. The lowest priority has the value 0
SCHEDULE	FULL, NON	Defines the run-time behavior of the task
AUTOSTART	TRUE, FALSE	Defines whether the task is activated during the system start-up procedure or not
APPMODE	name of APPMODE	Defines the application mode in which the task is autostarted
ACTIVATION	1	Specifies the maximum number of queued activation requests for the task (NXP AUTOSAR OS does not allow multiply activations)
RESOURCE	name of RESOURCE	Resources accessed by the task. There can be several resource references
EVENT	name of EVENT	Event owned by the task. There can be several event references
STACKSIZE	integer	Defines the size of the task stack in bytes
TIMING_PROTECTION	TRUE, FALSE	Specifies is the Timing Protection applied to the TASK

The application definition file contains one such statement per task. The task generation statement is described in detail in System Objects Definition.

3.5.4 Run-time Services

The AUTOSAR OS provides a set of services for the user to manage tasks. A detailed description of these services is provided in System Services. Below is only a brief list of them.

Table 3-4. Task Management Run-time Services

Service Name	Description
ActivateTask	Activates the task, i.e. moves it from *suspended* to ready state
TerminateTask	Terminates the running task, i.e. moves it from *running* to *suspended* state
ChainTask	Terminates the running task and activates the new one immediately
Schedule	Yields control to a higher-priority ready task (if any)

Table continues on the next page...

Table 3-4. Task Management Run-time Services (continued)

Service Name	Description
GetTaskID	Gets the identifier of the running task
GetTaskState	Gets the status of a specified task

Examples of using the run-time services are provided in Task Management Services.

3.5.5 Constants

The following constants are used within the AUTOSAR Operating System to indicate the task states:

- *RUNNING*

  The constant of data type *TaskStateType* for task state *running*

- *WAITING*

  The constant of data type *TaskStateType* for task state *waiting*

- *READY*

  The constant of data type *TaskStateType* for task state *ready*

- *SUSPENDED*

  The constant of data type *TaskStateType* for task state *suspended*

These constants can be used for the variables of *TaskStateType*.

The following constant is used within the AUTOSAR OS to indicate the task:

- *INVALID_TASK*

  The constant of data type *Task Type* for an undefined task

3.5.6 Conventions

Within the AUTOSAR OS application a task should be defined according to the following pattern:

```
TASK ( TaskName )
{
...
}
```

The name of the task function is generated from *TaskName* by macro *TASK*.

Chapter 4
Interrupt Processing

4.1 Interrupt Processing

This chapter highlights AUTOSAR OS approach to the interrupt handling.

This chapter consists of the following sections:

- General
- ISR Categories
- Interrupt Level Manipulation
- ISR Stack
- Programming Issues
- Interrupt Dispatcher

4.2 General

Interrupt processing is an important part of any real-time operating system. An *Interrupt Service Routine (ISR)* is a routine which is invoked from an interrupt source, such as a timer or an external hardware event. ISRs have higher priority than all tasks and the scheduler.

In the supported scalability classes which lack memory protection with ECC1 class all ISRs of category 2 use the 'ISR stack' which is used only by ISRs during their execution. The size of the ISR stack(s) is defined by the user. The OS protection handlers has their own stacks, not configurable by the user.

According to the *OSEK/VDX Operating System, v.2.2.2, 5 July 2004* specification there are no services for manipulation of CPU and/or OS interrupt levels directly. Therefore nested interrupts with the same hardware level can not occur. For CPU with one hardware level nested interrupts are forbidden. For multilevel modes nested interrupts with different priority are allowed. Application is not allowed to manipulate interrupt enabling bits in the CPU state registers.

NOTE

It is strictly forbidden for application to manipulate directly the registers that control CPU state and interrupts.

ISRs can communicate with Tasks in the AUTOSAR OS by the following means:

- ISR can activate a task;
- ISR can send/receive messages;
- ISR can increment a counter;
- ISR can get Task ID;
- ISR can get state of the task;
- ISR can set event for a task;
- ISR can get event mask of the task;
- ISR can manipulate alarms;

Interrupts cannot use any OS services except those which are specially allowed to be used within ISRs. When using other services, in the Extended Status of the Operating System the error will be reported. See Table 4-1 and System Services for details.

4.3 ISR Categories

In the AUTOSAR Operating System two types of Interrupt Service Routines are considered.

4.3.1 ISR Category 1

ISRs of this type are executed on the stack of the current runnable. In this case, if the ISR uses the stack space for its execution, the user is responsible for the appropriate stack size. Only 6 Interrupt Management services (enabling/disabling interrupts) are allowed in ISRs of category 1 (see Table 4-1). After the ISR is finished, processing continues exactly at the instruction where the interrupt occurred, i.e. the interrupt has no influence on task management.

The following statements are used to define ISR category 1.

```
ISR( ISR_handler )
{
...
/* the code without any OS service calls */
/* except Suspend/ResumeAllInterrupts */
...
}
```

NOTE

ISR category 1 shall have the priority higher than any ISR of category 2 because if ISR category 1 was interrupted by ISR category 2, rescheduling might take place at the end of ISR category 2 execution and ISR category 1 execution would be suspended therefore.

4.3.2 ISR Category 2

In ISR category 2 the Operating System provides an automatic switch to the ISR stack (except in BCC1, SC1) and enters OS ISR execution context. After that, any user's code can be executed, including allowed OS calls (to activate a task, increment a counter). See Service Protection for the list of services allowed for ISR. At the end of the ISR, the System automatically switches back to the stack of interrupted Task/ISR and restores context.

The following statements are used to define ISR category 2.

```
ISR( ISR_handler )
{
/* the code with allowed OS calls */
}
```

Inside the ISR, no rescheduling will take place. Rescheduling may only take place after termination of the ISR of category 2 if a preemptable task has been interrupted.

4.3.3 Nonmaskable Interrupt (NMI)

Interrupts which can not be disabled, such as NMI or any synchronous exception, shall not be assigned to OS ISRs. If the OS can not disable ISR, then the OS will not be able to protect it's critical code sections and will crash.

In SC1 the User can point to NMI handler in vector table and execute it out of context of the OS.

4.4 Interrupt Level Manipulation

Direct manipulation with the CPU interrupt flags or levels is strictly forbidden. The user can not define values of the interrupt masks directly. Interrupts are enabled during task execution if there are any ISR or any Timer is configured, otherwise OS dispatcher does not control interrupt levels of MCU. Interrupts can be disabled via disable/enable interrupt API functions or by using resource mechanism.

DisableAllInterrupts service can be used to temporary disable all interrupts. To return to previous interrupt status *EnableAllInterrupts* service must be called after it in the body of the same Task or ISR where *DisableAllInterrupts* is called.

SuspendAllInterrupts and *ResumeAllInterrupts* pair has the same effect as *DisableAllInterrupts* - *EnableAllInterrupts* pair but allows nesting of pairs.

SuspendOSInterrupts service can be used to temporary disable all interrupts of category 2, IOC callback (it has maximal priority of ISR category 2) and cross-core interrupts. To return to previous interrupt status *ResumeOSInterrupts* service must be called after it in the body of the same Task or ISR where *SuspendOSInterrupts* is called.

Resources can be used to temporary disable interrupts. If a Task or ISR occupies resource which is referenced by ISR of priority 'P' then all ISRs with priority equal or lower than 'P' are disabled and task scheduling is disabled. Interrupts and task scheduling are reenabled after releasing the resource.

4.5 ISR Stack

The purpose of the ISR stack(s) is to save memory. Switching to the ISR stack is performed by the OS at the beginning of ISR category 2. This stack(s) are used only by ISRs of category 2. In ECC1 within SC1 if nested interrupt occurs after the stack has been switched, they will continue to use the ISR stack. After completion of ISRs category 2 OS switches back from ISR stack.

In BCC1 class within SC1 ISRs use common single stack and stack switch is not performed.

The *interrupt stack frame* usually consists of the CPU registers, and optionally some compiler-dependent 'virtual' registers. The CPU registers are pushed onto the stack under hardware or software control. In the latter case the compiler generates a stack frame by means of adding special sequences of machine instructions before the first statement in the function.

Most compilers use function modifiers (like 'interrupt') to generate stack frames. In turn, the *ISR* keyword, specified in OSEK (see Conventions), is a macro for this modifier.

4.6 Interrupt Dispatcher

NXP AUTOSAR OS/S32K provides interrupt dispatcher for the external interrupt distinction. It is necessary for each external interrupt to define an ISR object in the OIL file and assign the *EXTERNAL* value to the *IrqChannel* attribute, set appropriated value for the *IrqNumber* attribute (For the correspondence between *IrqNumber* value and external interrupt sources see the Hardware Technical References) and assign the *PRIORITY* attribute value. The ISRs functions are called via OS Interrupt Dispatcher as conventional C functions. The ISRs shall not access the Interrupt Controller (NVIC) HW in any way.

4.7 Programming Issues

4.7.1 Run-time Services

AUTOSAR OS provides the set of services for interrupt management. These services are shown in the Table 4-1.

Table 4-1. Interrupt Management Services in AUTOSAR OS

Service Name	Description
DisableAllInterrupts	Disable all interrupts, does not allow nesting
EnableAllInterrupts	Restore state of interrupts saved by DisableAllInterrupts service
SuspendAllInterrupts	Disable all interrupts, allows nesting
ResumeAllInterrupts	Restore state of interrupts saved by SuspendAllInterrupts service
SuspendOSInterrupts	Disable interrupts of category 2, allows nesting
ResumeOSInterrupts	Restore state of interrupts saved by SuspendOSInterrupts service
DisableInterruptSource	Disables the interrupt source for this ISR
EnableInterruptSource	Enables the interrupt source for this ISR

Not all OS services may be used inside ISRs. Please refer to the Table 2-4 for the list of allowed services.

4.7.2 Constants

For each ISR defined in oil file the constant with name <IsrName>PRIORITY equal to this ISR *PRIORITY* is defined. This is an NXP OS extension of the AUTOSAR OS

4.7.3 Conventions

Within the application an Interrupt Service Routine should be defined according to the following pattern:

```
ISR( <ISRName> )[1]
{
    ...
}
```

The keyword ISR is the macro for compiler specific interrupt function modifier, which is used to generate valid code to enter and exit ISR.

It is possible to use one function as handler for several ISRs via ISR attribute *IsrFunction*. The value of the attribute shall exactly corresponds to the function name. This function shall be defined as *void <FuncName>(void)*, OS macro '*ISR*' shall not be used in this case.

4.7.4 ISR Definition

To define common ISR parameters like ISR stack size (used in SC1, ECC1) the corresponding OS properties should be specified in the configuration file.

Definition of Interrupt related properties looks like:

```
OS <name> {
    .....
    IsrStackSize = 800;
};
```

Definition of specific ISR object looks like:

```
ISR Handler {
    PRIORITY = 3;
    STACKSIZE = 256;
    CATEGORY = 2;
    RESOURCE = ISRresource;
    TIMING_PROTECTION = FALSE;
    Irq Channel = EXTERNAL {
        IrqNumber = 96; /* ENET0 interrupt request */
    };
};
```

1. OSEK/VDX does not specify using of keyword ISR for ISRs of category 1, it is NXP AUTOSAR OS specific for this category.

NOTE

PRIORITY of ISRs of category 2 shall be less than PRIORITY of any ISR of category 1.

See ISR Definition for details.

Chapter 5
Scheduler

5.1 Scheduler

This chapter provides a description of scheduling policies defined for OSEK OS.

This chapter consists of the following sections:

- General
- Scheduling Policy
- Programming Issues

5.2 General

The algorithm deciding which task has to be started and triggering all necessary OSEK Operating System internal activities is called *scheduler*. It performs all actions to switch the CPU from one instruction thread to another. It is either switching from task to task or from ISR back to a task. The task execution sequence is controlled on the base of task priorities (see section Task Priorities) and the scheduling policy used.

The scheduler is activated whenever a task switch is possible according to the scheduling policy. The principle of multitasking allows the operating system to execute various tasks concurrently. The sequence of their execution depends on the scheduling policy, therefore it has to be clearly defined.

Scheduler also provides the endless idle loop if there is no task ready to run. It may occur, when all tasks are in the suspended or waiting state until the awakening signal from an Interrupt Service Routine occurs. In this case there is no currently running task in the system, and the scheduler occupies the processor performing an endless loop until the ISR awakes a task to be executed.

The scheduler can be treated as a specific resource that can be occupied by any task. See Scheduler as a Resource for details.

The scheduling policy and some scheduler-related parameters are defined by the user, see Global System Attributes.

5.3 Scheduling Policy

The scheduling policy being used determines whether execution of a task may be interrupted by other tasks or not. In this context, a distinction is made between full-, non- and mixed-preemptive scheduling policies. The scheduling policy affects the system performance and memory resources. In the OSEK Operating System, all listed scheduling policies are supported. Each task in an application may be preemptable or not. It is defined via the appropriate task property (preemptable/non-preemptable).

Note that the interruptability of the system depends neither on the Conformance Class, nor on the scheduling policy.

The desired scheduling policy is defined by the user via the tasks configuration option *SCHEDULE*. The valid values are - *NON* and *FULL*. If all tasks use NON scheduling, scheduler works as non-preemptive. If all tasks use FULL scheduling, scheduler works as full-preemptive. If some tasks use NON and other tasks use FULL scheduling, scheduler works as mixed-preemptive.

5.3.1 Non-preemptive Scheduling

The scheduling policy is considered as non-preemptive, if a task switch is only performed via one of a selection of explicitly defined system services (explicit point of rescheduling).

Non-preemptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically, the lower priority nonpreemptable section of a running task delays the start of a task with higher priority, up to the next point of rescheduling. The time diagram of the task execution sequence for this policy looks like the following:

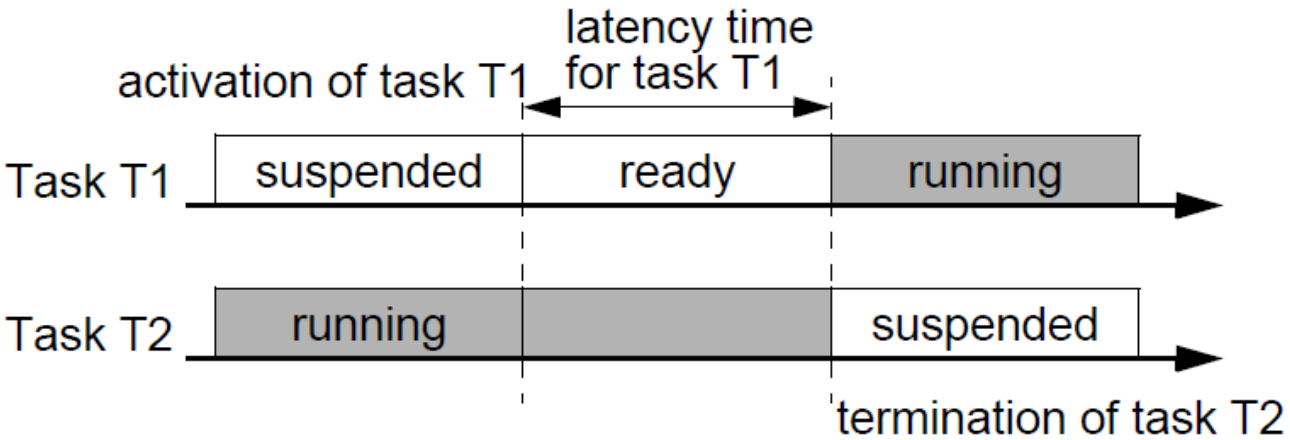

Figure 5-1. Non-preemptive Scheduling

Task T2 has lower priority than task T1. Therefore, it delays task T1 up to the point of rescheduling (in this case termination of task T2).

Only four following *points of rescheduling* exist in the OSEK OS for nonpreemptive scheduling:

- Successful termination of a task (via the *TerminateTask* system service);
- Successful termination of a task with explicit activation of a successor task (via the *ChainTask* system service);
- Explicit call of the scheduler (via the *Schedule* system service);
- Explicit wait call, if a transition into the waiting state takes place (via the *WaitEvent* system service, Extended Tasks only).

In the non-preemptive system, all tasks are non-preemptable and the task switching will take place exactly in the listed cases.

5.3.2 Full-preemptive Scheduling

Full-preemptive scheduling means that a task which is presently *running* may be rescheduled at any instruction by the occurrence of trigger conditions preset by the operating system. Full-preemptive scheduling will put the *running* task into the *ready* state as soon as a higher-priority task has got *ready*. The task context is saved so that the preempted task can be continued at the location where it was interrupted.

With full-preemptive scheduling, the latency time is independent of the run time of lower priority tasks. Certain restrictions are related to the enhanced complexity of features necessary for synchronization between tasks. As each task can theoretically be rescheduled at any location, access to data that are used jointly with other tasks must be synchronized.

In a full-preemptive system all tasks are preemptable.

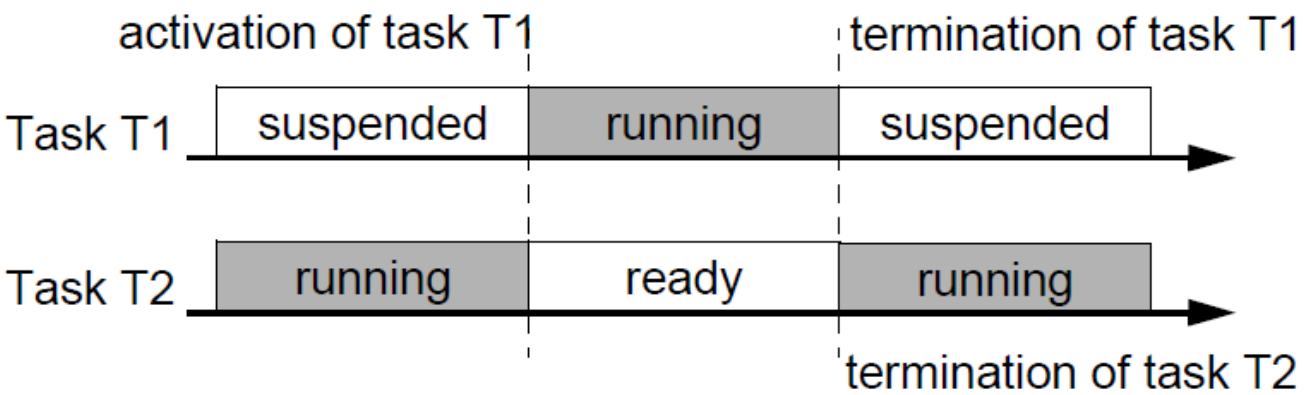

Figure 5-2. Full-preemptive Scheduling

5.3.3 Mixed-preemptive Scheduling

If full-preemptive and non-preemptive scheduling principles are to be used for execution of different tasks on the same system, the resulting policy is called 'mixed-preemptive' scheduling. The distinction is made via the task property (preemptable/non-preemptable) of the running task.

The definition of a non-preemptable task makes sense in a full-preemptive operating system in the following cases:

- if the execution time of the task is in the same magnitude as the time of the task switch,
- if the task must not be preempted.

Many applications comprise only a few parallel tasks with a long execution time, for which a full-preemptive scheduling policy would be convenient, and many short tasks with a defined execution time where non-preemptive scheduling would be more efficient. For this configuration, the mixedpreemptive scheduling policy was developed as a compromise.

5.3.4 Groups of Tasks

The operating system allows tasks to combine aspects of preemptive and non-preemptive scheduling by defining groups of tasks. For the tasks which have the same or lower priority as the highest priority within a group, the tasks within the group behave like non-preemptable tasks: rescheduling will only take place at the points of rescheduling described in Non-preemptive Scheduling. For tasks with a higher priority than the highest priority within the group, tasks within the group behave like preemptable tasks (see Full-preemptive Scheduling).

Chapter Internal Resources describes the mechanism of defining groups by the instrumentality of internal resources.

5.4 Programming Issues

5.4.1 Configuration Options

The following system configuration options are intended to define scheduler properties:

- *CC*

  Specifies conformance class. If AUTO, conformance class is defined according to tasks definitions.

- *USERESSCHEDULER*

  If this option is set to FALSE then *RES_SCHEDULER* is not supported by OS.

5.4.2 Run-time Services

The scheduler is not accessed by the user directly. The user can only pass the CPU control to the scheduler by means of the *Schedule* system service. This leads to task rescheduling if there is a ready task of priority higher than the running one.

The scheduler can be used by the programmer as a resource. To provide this possibility, the services *GetResource* and *ReleaseResource* with the constant *RES_SCHEDULER* as a parameter can be called by a task. It means that the task cannot be preempted by any

other task after the scheduler occupation, before the corresponding call *ReleaseResource* is performed. While the task occupies the scheduler, it has the highest priority and, therefore, cannot be preempted by other tasks (only ISRs can get the CPU control during this period). Such programming practice can be used for important critical sections of code.

See the example:

```
GetResource( RES_SCHEDULER );
...
/* Critical section */
/* this code cannot be interrupted by any other task */
...
ReleaseResource( RES_SCHEDULER );
```

Chapter 6
Resource Management

6.1 Resource Management

This chapter describes resource management and task coordination by means of resources.

This chapter consists of the following sections:

- General
- Access to Resources
- Internal Resources
- Programming Issues

6.2 General

The resource management is used to coordinate concurrent access of several tasks or/and ISRs to shared resources, e.g. management entities (scheduler), program sequences (critical sections), memory or hardware areas. In the NXP AUTOSAR OS the resource management is provided in all Conformance Classes.[1]

The resource management ensures that

- two modules (tasks or ISRs) cannot occupy the same resource at the same time,
- priority inversion cannot arise while resources are used,
- deadlocks do not occur due to the use of these resources,
- access to resources never results in *waiting* state.

The functionality of the resource management is required only in the following cases:

- full-preemptable tasks,

1. This is the NXP OS extension of the AUTOSAR OS, which fully supports resources only in ECC conformance classes.

- non-preemptable tasks, if the user intends to have the application code executed under other scheduling policies too.
- resource sharing between tasks and/or ISRs.

Resources cannot be occupied by more than one task or ISR at a time. The resource which is now occupied by a task or ISR must be released before another task or ISR can get it. The AUTOSAR operating system ensures that tasks are only switched from *ready* to *running* state if all the resources which might be occupied by that task during its execution have been released. The AUTOSAR operating system ensures that an ISR is enabled if all the resources which might be occupied by that ISR during its execution have been released. Consequently, no situation occurs in which a task or an ISR tries to access an occupied resource. A special mechanism is used by the AUTOSAR Operating System to provide such behavior, see Priority Ceiling Protocol for details.

In case of multiple resource occupation, the task or ISR must request and release the resources following the LIFO principle (stack). For example, if the task needs to get the communication hardware and then the scheduler to avoid possible preempts, the following code may be used:

```
GetResource( SCI_res ); /* occupy the SCI resource */
... /* user's code */
GetResource( RES_SCHEDULER ); /* occupy the scheduler resource */
... /* user's code */
ReleaseResource( RES_SCHEDULER ); /* release the scheduler */
ReleaseResource( SCI_res ); /* release the SCI resource */
```

The AUTOSAR OS resource management allows the user to prevent such situations as priority inversion and deadlocks which are the typical problems of common synchronization mechanisms in real-time applications (e.g., semaphores).

It is not allowed to occupy RES_SCHEDULER resource in the ISR.

6.3 Access to Resources

Before they can be used, the resources must be defined by the user at the system configuration stage through the *RESOURCE* definition, see Resource Definition. The resource must be referenced in all TASKs and ISRs which can occupy it. (A special resource RES_SCHEDULER is referenced and can be used by any TASK by default.) After that the task or ISR can occupy and release the resource using the *GetResource* and *ReleaseResource* services. While the resource is occupied, i.e. while the code between these services is executed, this resource cannot be occupied by another task or ISR.

In the AUTOSAR Operating System the resources are ranked by priority. The priority which is statically assigned to each resource is called *Ceiling Priority*. The resource priority is calculated automatically during the system generation. It is possible to have resources of the same priority, but the Ceiling Priority of the resource is higher or equal to the highest task or ISR priority with access to this resource.

6.3.1 Restrictions when using Resources

TerminateTask, *ChainTask*, *Schedule* and *WaitEvent* must not be called while a resource is occupied. The interrupt service routine must not be completed with the resource occupied.

AUTOSAR strictly forbids the nested access to the same resource. In the rare cases when the nested access is required, it is recommended to use a second resource with the same behavior as the first resource. The OIL language especially supports the definition of resources with identical behavior (so-called 'linked resources').

6.3.2 Priority Ceiling Protocol

The Priority Ceiling Protocol is implemented in the AUTOSAR Operating System as a resource management discipline.

The priority ceiling protocol elevates the task or ISR requesting a resource to a resource priority level. This priority can be simply calculated during the system generation. As shown in General the Ceiling Priority is:

- Higher or equal to the highest task or ISR priority with access to this resource (task T1);
- Lower than the priority of those tasks or ISR which priority is higher than the one of task T1.

Note that all ISR priorities are higher than any task and scheduler priorities.

When a task or ISR occupies a resource, the system temporarily changes its priority. It is automatically set to the Ceiling Priority by the resource management. Any other task or ISR which might occupy the same resource does not enter *running* state (ISR stays pending and cannot start) due to its lower or equal priority. If the resource occupied by the task or ISR is released, the task (ISR) returns to its former priority level. Other tasks which might occupy this resource can now enter *running* state (ISR can start).

Hardware interrupt levels are used by resources which can be occupied in the ISR. When such a resource is occupied by a task or ISR, the interrupts of the corresponding priority are disabled and the AUTOSAR OS scheduler is switched off. Therefore, the *running* task can not be switched to *ready* state while such a resource is occupied. Releasing the resource causes enabling interrupts of the corresponding level and switching on the AUTOSAR OS scheduler.

The example shown on Figure 6-1 illustrates the mechanism of the Priority Ceiling Protocol.

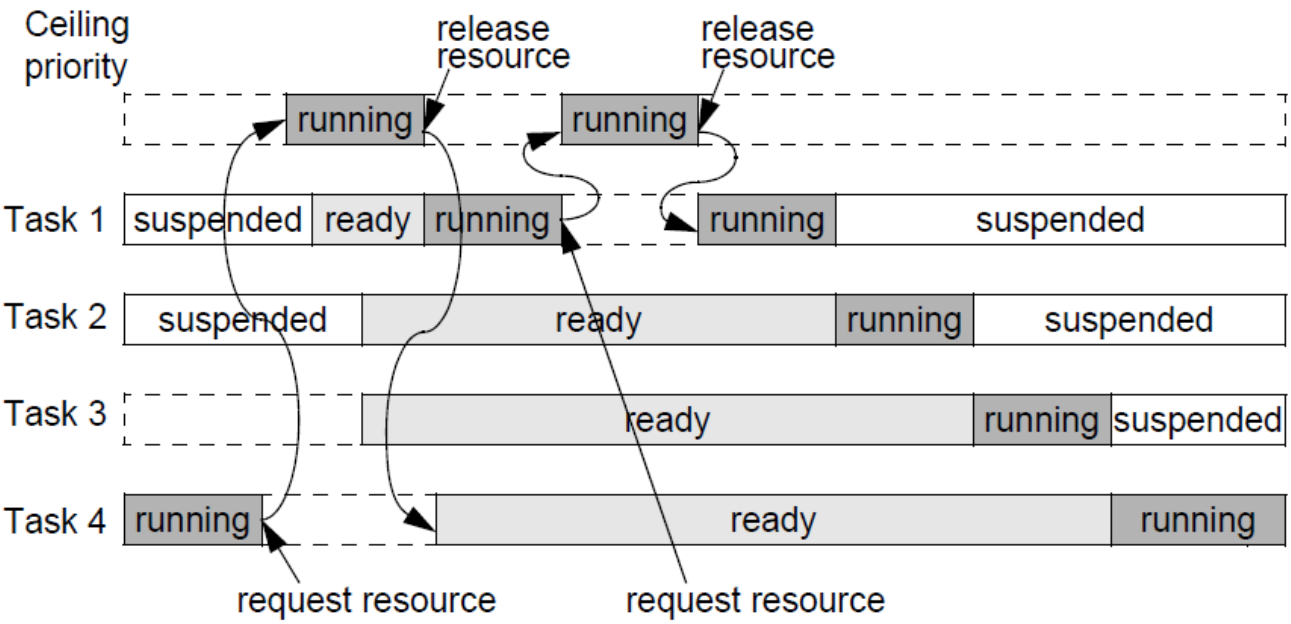

Figure 6-1. Priority Ceiling Protocol

In the figure above Task 1 has the highest priority and Task 4 has the lowest priority. The resource has a priority greater than or equal to the Task 1 priority. When Task 4 occupies the resource, it gets a priority not less than the Task 1 has, therefore it cannot be preempted by *ready* Task 1 until it releases the resource. As soon as the resource is released, Task 4 is returned to its low priority and becomes *ready*, and Task 1 becomes the *running* task. When Task 1, in turn, occupies the resource, its priority is also changed to the Ceiling Priority.

6.3.3 Scheduler as a Resource

The AUTOSAR operating system treats the scheduler as a specific resource which is accessible to all tasks. Therefore, a standard resource with the predefined identifier *RES_SCHEDULER* is generated, and it is supported in all Conformance Classes. If a task

calls the services *GetResource* or *ReleaseResource* with this identifier as a parameter, the task will occupy or release the scheduler in the manner of a simple resource. See the code example in General.

If a task wants to protect itself against preemptions by all other tasks, it can occupy the scheduler exclusively. When it is occupied, interrupts are received and processed normally. However, it prevents the tasks rescheduling.

NOTE

If a task gets the scheduler as a resource, it must release it before the point of rescheduling!

Reference to RES_SCHEDULER from TASK object is optional. The user may define the resource with the name RES_SCHEDULER in the OIL file but this resource will have the maximal (scheduler) priority regardless of which tasks have the reference to it. The RES_SCHEDULER is referenced and can be used from any task by default. The RES_SCHEDULER resource cannot be occupied from the ISR.

6.4 Internal Resources

The internal resources are the resources which are not visible to the user and therefore they can not be addressed by the system functions GetResource and ReleaseResource. Instead, they are managed strictly internally within a clearly defined set of the system functions. Besides, the behavior of the internal resources is exactly the same as the behavior of standard resources (the priority ceiling protocol etc.). At most one internal resource can be assigned to a task during the system generation. If an internal resource is assigned to a task, the internal resource is managed as follows:

- The resource is automatically taken when the task enters running state, except when it has already taken the resource. As a result, the priority of the task is automatically changed to the ceiling priority of the internal resource.
- At the points of rescheduling, defined in chapter Scheduling Policy, the resource is automatically released.

The tasks which have the same internal resource assigned form a group of tasks. The tasks within a certain group behave like the non preemptable tasks - they can not preempt each other; while the tasks with the priority higher than the highest priority within the group preempt the tasks within the group.

The non preemptable tasks may be considered as a special group with an internal resource of the same priority as the RES_SCHEDULER priority (chapter Non-preemptive Scheduling). The internal resources can be used in all cases when it is necessary to avoid unwanted rescheduling within a group of tasks. More than one internal resource can be defined in a system thus defining more than one group of tasks.

The general restriction on some system calls that they must not be called with the resources occupied (see Restrictions when using Resources) is not applied to the internal resources, as the internal resources are handled within those calls.

The tasks which have the same assigned internal resource cover a certain range of priorities. It is possible to have the tasks which do not use this internal resource in the same priority range, but these tasks will not belong to the group. So they are preemptable by the tasks of the group.

6.5 Programming Issues

6.5.1 Configuration Options

The following system configuration option is intended to decrease the amount of RAM and ROM used by the OS:

- *USERESSCHEDULER*

  If this option is set to FALSE, RES_SCHEDULER is not supported by the OS.

6.5.2 Data Types

The AUTOSAR OS determines the following data type for the resource management:

- *ResourceType*

  The abstract data type for referencing a resource.

The only data type must be used for operations with resources.

6.5.3 Run-time Services

The AUTOSAR OS provides a set of services for the user to manage resources. Detailed descriptions of these services are provided in Resource Management Services. Below is only a brief list.

Table 6-1. Resource Management Run-time Services

Service Name	Description
GetResource	This call serves to occupy the resource (to enter critical section of the code, assigned to the resource)
ReleaseResource	Releases the resource assigned to the critical section (to leave the critical section)

6.5.4 Resource Definition

To define a resource, the following definition statement should be specified in the application configuration file:

```
RESOURCE ResourceName {
    RESOURCEPROPERTY = STANDARD;
};
```

For more details see Resource Definition.

Chapter 7
Counters, Alarms, Schedule Tables

This chapter describes usage of these control mechanisms in AUTOSAR OS.

This chapter consists of the following sections:

- General
- Counters
- Alarms
- Alarm Callback
- Schedule Table
- Programming Issues

7.1 General

The AUTOSAR operating system comprises a two level concept to make use of recurring occasions like periodic interrupt of timers, interrupt of the sensors on rotating angles, or any recurring software occasions. To manage such situations, counters and alarms are provided by the NXP AUTOSAR OS. Additionally AUTOSAR OS provides a Schedule Table mechanism for fast tasks activations in accordance with statically defined schedule. The recurring occasions (sources) can be registered by counters. Based on counters, the OS offers an alarm mechanism to the application software. Counters and alarms are provided by the AUTOSAR OS in all Conformance and Scalability Classes.

NXP AUTOSAR OS provides two types of counters: software (SW) counters and hardware (HW) counters. HW counters allow more precise timing while decreases system overhead time because the timer interrupts are occurs only when the alarm(s) attached to the counter expiries.

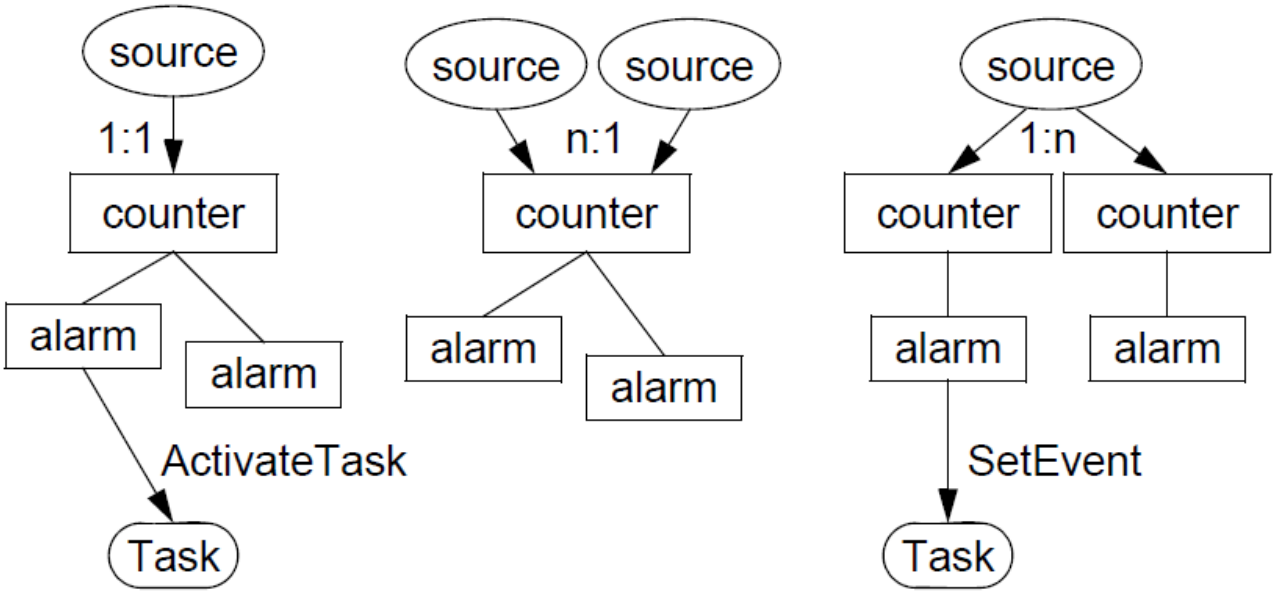

Figure 7-1. Counters and Alarms

7.2 Counters

Any occasion in the system can be linked with a counter, so that when the occasion has occurred, the counter value is changed. A counter is identified in the system via its symbolic name, which is assigned to the counter statically at the configuration stage.

The AUTOSAR OS defines 2 types of counters:

- Hardware Counter - a counter that is advanced by hardware (e.g. timer). The counter value is 'in hardware'.
- Software Counter - a counter which may be incremented by the *IncrementCounter()* API call and keeps it's value in the variable.

The counter is represented by a current counter value and some counter specific parameters: *counter initial value*, *conversion constant* and *maximum allowed counter value*. They are defined by the user. The last two parameters are constants and they are defined at system generation time. The counter initial value is the dynamic parameter. The user can initialize the counter with this value and thereafter on task or on interrupt level advance it using the system service *IncrementCounter*.

The HW counters may use only *System* and *Second* Timers and has a maximum allowed value defined by number of bits in the timer HW. Only FTM timers with output compare registers may be used for HW counters. The source(s) for HW counters is hardware itself. For HW counters one tick of hardware timer is equivalent to a *Period* for SW counter,

thus enabling more precise timing while keeping system overhead on interrupt processing low because timer interrupts are raised only when alarms attached to this counter are expired.

The maximum allowed counter value specifies the number after which the counter rolls over. After a counter reaches its maximum allowed possible value (or rolls over the predefined size - 32 bits), it starts counting again from zero.

The conversion constant *TICKSPERBASE* can be used to convert the counter value into an appropriate user specific unit of measurement, e.g. seconds for timers, angular degrees for rotating axles. The conversion is done by the user's code and the parameter can be treated as a counterspecific reference value, it is not used by the OS for any purposes.

The operating system provides the service *GetCounterInfo* to read these counter specific configuration values. Also the service *GetCounterValue* is provided to read the current counter value.

NXP AUTOSAR OS provides two timers (the internal system clocks): the *System Timer* and the *Second Timer*. The timers attributes are not defined in AUTOSAR OS specifications, this is NXP OS extension of the AUTOSAR OS. User can turn on or turn off the system timer using the *SysTimer* attribute and the second timer using the *SecondTimer* attribute. The timer can be assigned to a standard counter with the following additions:

- special constants are defined to describe counter parameters and to decrease access time;
- the user defines the source of hardware interrupts for the counter attached to the timer.

In the system definition statement for the system (second) timer the user should define one of possible hardware interrupt sources. Parameters to tune the hardware can be also defined by the user in this statement. This possibility allows the user to exactly tune the system (see ARM Architecture Platform-Specific Features for details).

While hardware related parameters are defined, the code to initialize the system (second) timer hardware and the interrupt handler are automatically provided for the user as a part of the OS. The handler is an ISR category 2 but it is not needed to define the ISR in OIL file. In that case the user does not have to care about handling of this interrupt and the provided code can not be changed.

Software Counters may be triggered from user defined ISR(s). Hardware interrupts which are used to trigger counters have to be handled in usual manner. To perform any actions with the counter the application software processing the occasion should call the *IncrementCounter* system service.

The user is free to assign one source exactly to one counter (1:1 relationship), several sources to one counter (n:1 relationship), or one source to several counters (1:n relationship), see Figure 7-1. Meaning that it is possible to advance the same counter in different software routines.

7.3 Alarms

The alarm management is built on top of the counter management. The alarm management allows the user to link task activation or event setting or a call to callback function to a certain counter value. These *alarms* can be defined as either single (one-shoot) or cyclic alarms.

The AUTOSAR OS allows the user to set alarms (relative or absolute), cancel alarms and read information out of alarms by means of system services. Alarm is referenced via its symbolic name which is assigned to the alarm statically at the configuration stage.

Examples of possible alarm usage are:

- 'Activate a certain task, after the counter has been advanced 60 times', or
- 'Set a certain event, after the counter has reached a value of 90'.

The counter addressed in the first example might be derived from a timer which is advanced every second. The task in the example is then activated every minute. The counter addressed in the second example might be derived from a rotating axle. The event is set on a 90 degree angle.

The AUTOSAR OS takes care of the necessary actions of managing alarms when a counter is advanced.

Counters and Alarms are defined statically. The assignment of alarms to counters, as well as the action to be performed when an alarm expires (task and event), are also defined statically. An application can use an alarm after it has been defined and assigned to a counter. Alarms may be either in the stop state or running state. To run an alarm, the special system services are used, which set dynamic alarm parameters to start it.

Dynamic alarm parameters are:

- the counter value when an alarm has to expire.
- the cycle value for cyclic alarms.

An alarm can be started at any moment by means of system services *SetAbsAlarm* or *SetRelAlarm*. An alarm will expire (and predefined actions will take place) when a specified counter value is reached. This counter value can be defined relative to the actual counter value or as an absolute value. The difference between relative and absolute alarms is the following:

- Relative alarm expires when the specified number of counter ticks has elapsed, starting from the current counter value at the moment the alarm was set.
- Absolute alarm expires when the counter reaches the specified number of ticks, starting from zero counter value no matter which value the counter had at the moment the alarm was set. If the specified number of ticks is less than the current counter value, the counter will roll over and count until the specified value. If the specified value is greater than the current value, the alarm will expire just after the counter reaches the desired number. This is illustrated by Figure 7-2. In the latter case, the total time until the alarm expires is the sum of T_1 and T_2.

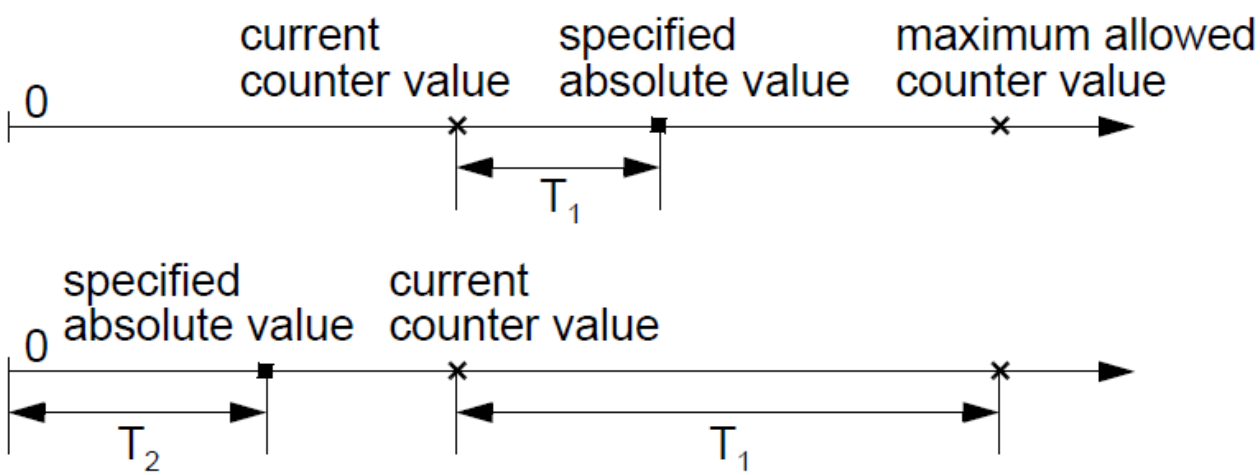

Figure 7-2. Two Cases for the Absolute Alarm

If a cycle value is specified for the alarm, it is logged on again immediately after expiry with this relative value. Specified actions (task activation or event setting) will occur when the counter counts this number of ticks, starting from the current value. This behavior of the cyclic alarm is the same both for relative and absolute alarms. If the cycle value is not specified (it equals zero) the alarm is considered as a single one.

NOTE

It is not recommended to use values of cycle and/or increment parameters close to 0xFFFF (hardware counter *MAXALLOWEDVALUE*) for Alarms configured on a Hardware Timer. The difference between *MAXALLOWEDVALUE* and this values should be greater than interrupt latency of the system (time spent in the longest ISR

Behaviour of alarms started by SetAbsAlarms or with autostart type ABSOLUTE

The new ('strict') behaviour for alarms started by SetAbsAlarms or with autostart type ABSOLUTE is provided optionally. This behaviour is provided only for alarms driven by software counters. In this case alarm is triggered when the underlying counter value next equals to given the absolute value 'start'. So, if the value 'start' passed to the OS service or configured for an autostarted alarm is equal to the underlying counter value, given alarm expires when the absolute value is reached again, i.e. after the next overrun of the underlying counter.

If the compiler option "*–DOS_STRICT_ABS_ALARMS*" is specified, then the 'strict' behaviour is provided by the OS. Otherwise, the OS starts alarm processing immediately.

The same behaviour is supported for schedule tables started by StartScheduleTableAbs or with autostart type ABSOLUTE. See the chapter " Schedule Table".

7.4 Alarm Callback

The user can define an alarm callback function for each alarm. The function is placed in the user application and its name is added to the ALARM object definition as value of *ALARMCALLBACKNAME* attribute.The alarm callback is the usual user's function. It can have neither parameter(s) nor return value.

Only the *SuspendAllInterrupts* and *ResumeAllInterrupts* services may be used within alarm callback.

The callback function shall have next definition:

```
ALARMCALLBACK(CallbackName)
{
    /* user application code */
}
```

7.5 Schedule Table

AUTOSAR OS provides a special feature for fast tasks activations in accordance with statically defined schedule named *Schedule Table*. User can define in a configuration file a sequence of tasks and/or events to be cyclically or one-time activated at predefined time points. Only one *Task* may be activated (or *Event* set) for each *Action* of *Schedule Table* but the *Offset* of several *Actions* may be set to the same value to achieve a simultaneous activation of two or more tasks.

NXP AUTOSAR OS does not limit the number of *Schedule Tables* assigned to the *COUNTER* and running simultaneously.

Schedule Table may be synchronized to external time source via *SyncScheduleTable* OS service in accordance with statically configured *SYNCSTRATEGY - EXPLICIT* or *IMPLICIT*. NXP AUTOSAR OS/S32K supports Global Time Synchronization in all Scalability Classes.

Behaviour of schedule tables started by StartScheduleTableAbs or with autostart type ABSOLUTE

The new ('strict') behaviour for schedule table started by StartScheduleTableAbs or with autostart type ABSOLUTE is provided optionally. This behaviour is provided only for schedule tables driven by software counters. In this case schedule table is started when the underlying counter value next equals to given the absolute value 'start'. So, if the value 'start' passed to the OS service or configured for an autostarted schedule table is equal to the underlying counter value, the OS starts processing given schedule table when the absolute value is reached again, i.e. after the next overrun of the underlying counter.

If the compiler option "*–DOS_STRICT_ABS_ALARMS*" is specified, then the 'strict' behaviour is provided by the OS. Otherwise, the OS starts schedule table processing immediately.

The same behaviour is supported for alarms started by SetAbsAlarms or with autostart type ABSOLUTE. See the chapter " Alarms".

7.6 Programming Issues

7.6.1 Configuration Options

The following system configuration options affect the counter and alarm management:

- *SysTimer*

  If this option is turned on the System Timer is used.

- *SecondTimer*

  If this option is turned on the Second Timer is used.

7.6.2 Data Types

The following data types are established by AUTOSAR OS to operate with counters:

- *CounterType*

  The data type references a counter

- *TickType*

  The data type represents a counter value in system ticks.

- *TickRefType*

  The data type references data corresponding to the data type *TickType*. Reference to *TickType* variable can be used instead of *TickRefType* variable.

- *CtrInfoType*
  This data type represents a structure for storage of counter characteristics. This structure has the following fields:
  - *maxallowedvalue*

    maximum possible allowed count value;

  - *ticksperbase*

    number of ticks required to reach a counter-specific significant unit (it is a user constant, OS does not use it);

  - *mincycle*

    minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status).

  All fields have the data type *TickType*. The following code may illustrate usage of this data type:

```
CtrInfoType CntData;
TickType maxV, minC, cons;
GetCounterInfo( CntID, &CntData );
maxV = CntData.maxallowedvalue;
minC = CntData.ticksperbase;
cons = CntData.mincycle;
```

- CtrInfoRefType

  This data type references data corresponding to the data type *CtrInfoType*. Reference to *CtrInfoType* variable can be used instead of *CtrInfoRefType* variable

The following data types are established by AUTOSAR OS to operate with alarms:

- *AlarmBaseType*

This data type represents a structure for storage of alarm characteristics. It is the same as *CtrInfoType*;

- *AlarmBaseRefType*

  This data type references data corresponding to the data type *AlarmBaseType*;

- *AlarmType*

  The data type represents an alarm element.

7.6.3 Counters and Alarm Generation

To generate a counter in an application, the *COUNTER* definition is used, it looks like the following:

```
COUNTER CounterName {
    MINCYCLE = 5;
    MAXALLOWEDVALUE = 1000;
    TICKSPERBASE = 10;
};
```

To define system and second timer hardware-specific parameters, the following properties may/should be defined in the OS definition statement:

```
OS <name> {
    ...
    SysTimer = HWCOUNTER {
        COUNTER = <CounterName>;
        ISRPRIORITY = <priority>;
        TimerHardware = <TypeOfTimer> {
            Prescaler = OS {
                Value = <PrescalerValue>;
            };
        };
    };
    SecondTimer = SWCOUNTER {
        COUNTER = <CounterName>;
        ISRPRIORITY = <priority>;
        TimerHardware = <TypeOfTimer> {
            Prescaler = OS {
                Value = <PrescalerValue>;
            };
            TimerModuloValue = <TimerModuloValue>;
        };
    };
    ...
};
```

The system timer hardware parameters are described in detail in the section CPU Related Attributes.

To generate an alarm in an application, the ALARM definition is used, it looks like the following:

```
ALARM AlarmName {
    COUNTER = CounterName;
    AUTOSTART = FALSE;
    ACTION = SETEVENT {
        TASK = TaskName;
        EVENT = EventName;
    };
};
```

Detailed counter and alarm generation statements are described in Counter Definition and Alarm Definition

To generate a ScheduleTable in Application the SCHEDULETABLE definition is used, it looks like the following:.

```
SCHEDULETABLE ScheduleTableName {
    COUNTER = CounterName;
    AUTOSTART = FALSE;
    REPEATING = TRUE;
    DURATION = <integer>;
    EXPIRYPOINTS = EXPIRYPOINT {
        OFFSET = 0;
        ACTION = TASKACTIVATION {
            TASK = TASKSND1;
        }
    };
};
```

Detailed ScheduleTable generation statement is described in Schedule Table Definition.

7.6.4 Run-time Services

AUTOSAR OS grants a set of services for the user to manage counters and alarms. Detailed descriptions of these services are provided in System Services. Here only a brief list is given.

Table 7-1. Counter and Alarm Management Run-time Services

Service Name	Description
InitCounter	Sets the initial value of the counter
IncrementCounter	Increments the counter value and process attached alarms
GetCounterValue	Gets the counter current value
GetCounterInfo	Gets counter parameters
SetRelAlarm	Sets the alarm with a relative start value
SetAbsAlarm	Sets the alarm with an absolute start value
CancelAlarm	Cancels the alarm: the alarm is transferred into the STOP state
GetAlarm	Gets the time left before the alarm expires
GetAlarmBase	Gets alarm base (attached counter) parameters
StartScheduleTableRel	Starts Schedule Table processing at relative time offset
StartScheduleTableAbs	Starts Schedule Table processing at absolute time

Table continues on the next page...

Table 7-1. Counter and Alarm Management Run-time Services (continued)

Service Name	Description
StopScheduleTable	Stops Schedule Table processing
NextScheduleTable	Plans start of another Schedule Table
SyncScheduleTable	Synchronizes Schedule Table to external time
SetScheduleTableAsync	Sets Schedule Table status to asynchronous
GetScheduleTableStatus	Returns Schedule Table status

NOTE

InitCounter, GetCounterValue and GetCounterInfo services are not defined in *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. This is NXP OS extension of the AUTOSAR OS.

7.6.5 Constants

For all counters, the following constants are defined:

- *OSMAXALLOWEDVALUE_cname*

  Maximum possible allowed value of counter <cname> in ticks.

- *OSTICKSPERBASE_cname*

  Number of ticks required to reach a specific unit of counter <cname> (it is a user constant, OS does not use it).

- *OSMINCYCLE_cname*

  Minimum allowed number of ticks for a cyclic alarm of counter <cname>. This constant is not defined in STANDARD status

For the counters attached to *SysTimer* and *SecondTimer* special constants are provided by the operating system:

- *OSMAXALLOWEDVALUE*

  maximum possible allowed value of the system timer in ticks;

- *OSMAXALLOWEDVALUE2*

  maximum possible allowed value of the second timer in ticks;

- *OSTICKSPERBASE*

number of ticks that are required to reach a counter-specific value in the system counter (it is a user constant, OS does not use it);

- *OSTICKSPERBASE2*

  number of ticks that are required to reach a counter-specific value in the second counter (it is a user constant, OS does not use it);

- *OSTICKDURATION*

  duration of a tick of the system counter in nanoseconds;

- *OSTICKDURATION2*

  duration of a tick of the second counter in nanoseconds;

- *OSMINCYCLE*

  minimum allowed number of ticks for a cyclic alarm attached to the system counter (only for system with Extended Status);

- *OSMINCYCLE2*

  minimum allowed number of ticks for a cyclic alarm attached to the second counter (only for system with Extended Status);

- *OSTPTICKDURATION*

  duration of the TP counter tick in nanoseconds.

NOTE

OSMAXALLOWEDVALUE2, *OSTICKSPERBASE2*, *OSTICKDURATION2*, *OSTPTICKDURATION* and *OSMINCYCLE2* constants are not defined in the OSEK/VDX Operating System specification. These constants are NXP OS extension of the AUTOSAR OS.

Chapter 8
Events

8.1 Events

This chapter is devoted to event management and task coordination by means of events.

This chapter consists of the following sections:

- General
- Events And Scheduling
- Programming Issues

8.2 General

Within the AUTOSAR operating system, tasks and ISRs can be synchronized via occupation of a resource (see Resource Management). Another means of synchronization is the event mechanism, which is provided for Extended Tasks only. Events are the only mechanism allowing a task to enter the *waiting* state.

An *event* is a synchronization object managed by the AUTOSAR Operating System. The interpretation of the event is up to the user. Examples are: the signalling of a timer's expiry, the availability of data, the receipt of a message, etc.

Within the operating system, events are not independent objects, but allocated to Extended Tasks. Each event is represented by a bit in event masks which belong to Extended Tasks. Maximum number of events for Extended Task is 32. Each Extended Task has the mask of a 'set' events and the mask of events the task is waiting for ('wait' mask). When the Extended Task is activated all its events are cleared.

An Extended Task can wait for several events simultaneously and setting at least one of them causes the task to be transferred into the *ready* state. When a task wants to wait for one event or several ones, the corresponding bits in its 'wait' event mask are set by the system service *WaitEvent* which is designed to force a task to wait for an event. When

another task sets an event, it sets the specified bits of the 'set' event mask and if some bits in both 'wait' and 'set' masks are the same the task is transferred into the *ready* state. The task can clear its own events in the 'set' event mask using *ClearEvent* service.

All tasks can set any events of any Extended Task. Only the appropriate Extended Task (the owner of the particular event mask) is able to clear events and to wait for the setting (receipt) of events. Basic Tasks must not use the operating system services for clearing events or waiting for them.

An alarm can also be set for an Extended Task, which in turn sets an event at a certain time. Thus, the Extended Task can delay itself.

It is not possible for an interrupt service routine or a Basic Task to wait for an event, since the receiver of an event is an Extended Task in any case. On the other hand, any task or ISR can set an event for an Extended Task, and thus inform the appropriate Extended Task about any status change via this event.

8.3 Events And Scheduling

An event is an exclusive signal which is assigned to an Extended Task. For the scheduler, events are the criteria for the transition of Extended Tasks from the *waiting* state into the *ready* state. The operating system provides services for setting, clearing and interrogation of events, and for waiting for events to occur.

Extended Task is in the *waiting* state if an event for which the task is waiting is not set. If an Extended Task tries to wait for an event and this event has already been set, the task remains in the *running* state.

Figure 8-1 illustrates the procedures which are effected by setting an event: Extended Task 1 (with higher priority) waits for an event. Extended Task 2 sets this event for Extended Task 1. The scheduler is activated. Subsequently, Task 1 is transferred from the *waiting* state into the *ready* state. Due to the higher priority of Tasks 1 this results in a task switch, Task 2 being preempted by Task 1. Task 1 resets the event. Thereafter Task 1 waits for this event again and the scheduler continues execution of Task 2.

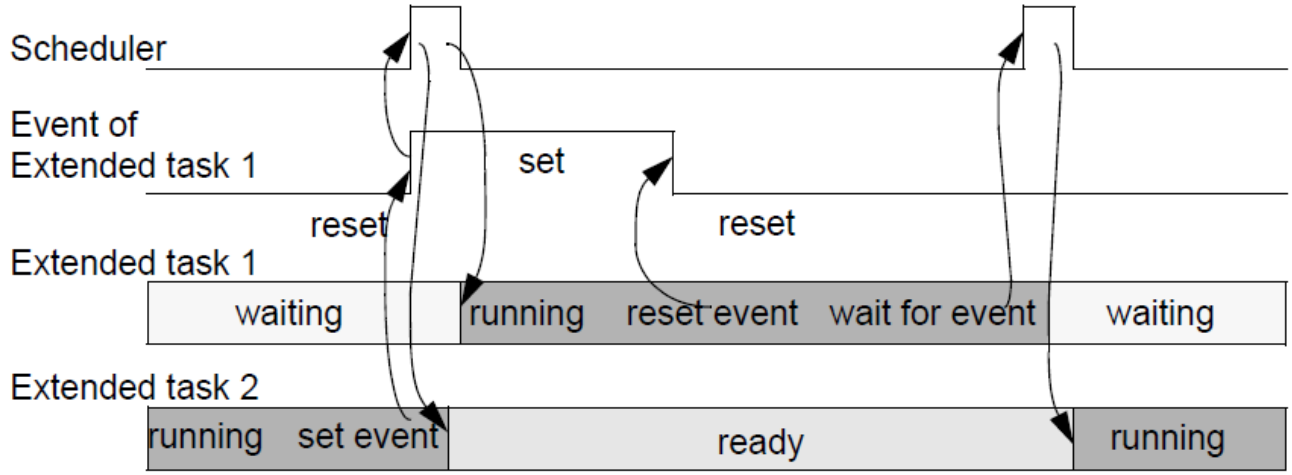

Figure 8-1. Synchronization by Events for Full-preemptive Scheduling

If non-preemptive scheduling is supposed, rescheduling does not take place immediately after the event has been set, as shown on Figure 8-2.

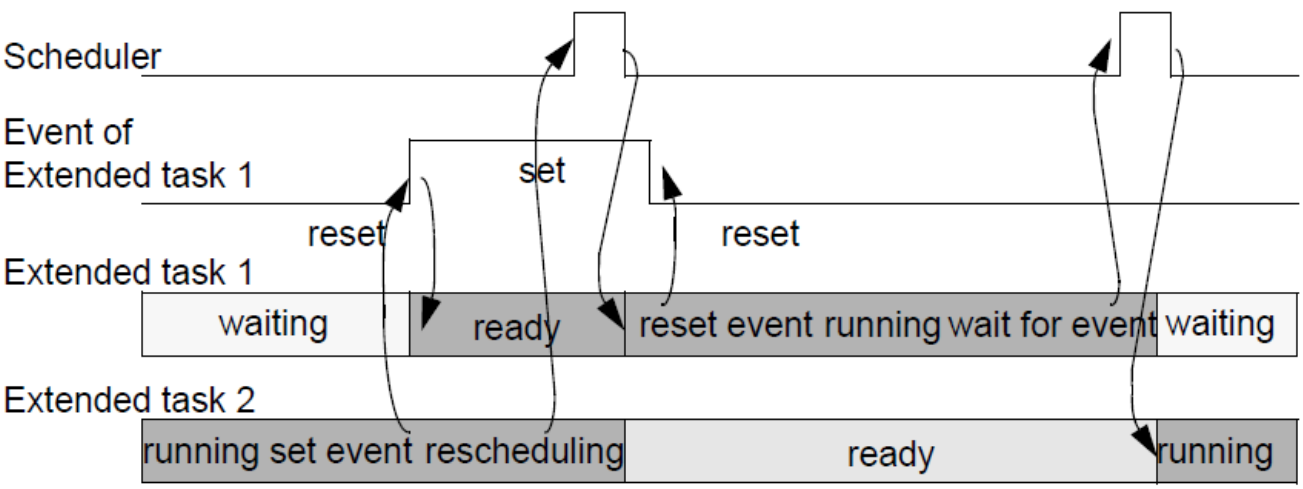

Figure 8-2. Synchronization by Events for Non-preemptive Scheduling

8.4 Programming Issues

8.4.1 Configuration Options

There are no any system configuration options controlling event management in the system.

8.4.2 Data Types

The AUTOSAR Operating System establishes the following data types for the event management:

- *EventMaskType*

  The data type of the event mask;

- *EventMaskRefType*

  The data type to refer to an event mask. Reference to *EventMaskType* variable can be used instead of *EventMaskRefType* variable.

The only data types must be used for operations with events.

8.4.3 Events Definition

To generate an event in an application the *EVENT* definition is used, it looks like the following:

```
EVENT EventName {
    MASK = 0x01;
};
```

Some task events which used by the task as internal flags can be undefined in the OIL file. But it is strictly recommended to define all events and reference them in TASK object. Missing of an event in OIL file can lead to wrong mask assignment. If task has no references to events the task is considered to be *Basic* one.

8.4.4 Run-time Services

AUTOSAR OS grants a set of services for the user to manage events. A detailed description of these services is provided in Event Management Services. Here only a brief list is given.

Table 8-1. Event Management Run-time Services

Service Name	Description
SetEvent	Sets events of the given task according to the event mask
ClearEvent	Clears events of the calling task according to the event mask
GetEvent	Gets the current event setting of the given task
WaitEvent	Transfers the calling task into the waiting state until specified events are set

Examples of the run-time services usage are provided in section Event Management Services.

Chapter 9
Communication

9.1 IOC Concept

Concept of communication between OS-Applications and IOC management system services are implemented according to *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3*.

The IOC defines sender and receiver sides. Several senders can be configured for one receiver (N:1 communication).

The IOC may be defined as Queued or Unqueued using attribute BUFFER_LENGTH (if it is set to 0 or undefined, then the IOC is Unqueued).

An Unqueued communication represents the current value of a system variable, e.g. engine temperature, wheel speed, etc. Unqueued communication are not buffered but overwritten with their actual values. The receive operation reads the Unqueued communication value. Thereby the message data is not consumed.

By contrast, Queued communication are stored in a FIFO buffer and they are read by the application in the order they arrived. A particular Queued communication can therefore be read only once, as the read operation removes it from the queue. A Queued communication would normally be used to track changes of state within a system, where it is important that receiver maintains synchronization of state information with the sender.

The Operating System ensures data consistency of message data during task operation, uniform in all types of scheduling. The received unqueued message data remains unchanged until a further send operation is performed, unless the task or function using the data overwrites the data with a IocWrite service.

As an option, task activation, event signalling or callback function can be defined statically to be performed at message arrival to notify a task. Task activation or event signalling can be used to inform tasks that shall act immediately on new message information. There is no special operating system service to wait for messages, but the

normal event mechanism is used. Name of callback function (*RECEIVER_PULL_CB*) is configured statically on receiver side in case of notification. Only one notification method can be assigned for certain receiving message.

9.2 Queued and Unqueued Communication

The IOC provides:

- unqueued (*Last-is-Best, data semantics*)
- queued (*First-In-First-Out, event semantics*)

communication operations.

Unqueued (*Last-is-Best, data semantics*) messages represent the current value of a state variable. A message can be accessed only via OS services. Message data is accessed indirectly through the application message copy. Access through the message copy guarantees consistency of information in the message between adjacent operations of IocWrite*(send)/ IocRead*(receive). The send operation overwrites the current value of a message. The receive operation reads the current value of an Unqueued Message whereby the message data is not consumed. The user can place copies of messages into the global or into the local memory of the functions.

The IocWrite* and IocRead* services ensure the consistent of writing and reading of message data within the send and receive operation (also in preemptive systems).

The User can set initial value by specifing it in IOC configuration via attribute *INIT_VALUE_SYMBOL*.

Queued (*First-In-First-Out, event semantics*) messages are stored in a FIFO buffer and they are read by the application in the order they arrive. The send operation adds a message data to the end of the message queue. The receive operation reads the first value of a Queued Message in the queue and then removes it from the queue. The user can place copies of messages into the global memory or into the local memory of the functions.

The IocSend* and IocReceive* services ensure the consistent of writing and reading of message data within the send and receive operation (also in preemptive systems).

When a Queued Message is received the information from the first message in the message queue is copied to the message copy and further work with the message information is performed using this copy.

The User can set the length of queue by specifing it in IOC configuration via attribute *BUFFER_LENGTH*.

9.3 Programming Issues

9.3.1 IOC Definition

Each IOC object in an application is generated by means of using statements like the following:

```
IOC MsgA {
        DATA_PROPERTIES = DATA_PROPERTY
        {
            DataTypeName = "MSGATYPE";
            DataTypeProperty = REFERENCE;
        };
        BUFFER_LENGTH = 0;

        RECEIVER = RCV {
            ACTION = SETEVENT {
                TASK = TASKRCV1;
                EVENT = MSGAEVENT;
            };
            RCV_OSAPPLICATION = RCV_APP;
        };

        SENDER = SND {
            SND_OSAPPLICATION = SND_APP;
        };
};
```

In detail IOC configuration statements is described in Inter-OS-application Communicator (IOC) Definition.

9.3.2 Run-time Services

NXP AUTOSAR OS grants a set of services for the user to manage IOCs. Detailed descriptions of these services are provided in section Communication Management Services..

Table 9-1. IOC Management Run-time Services

Service Name[1]	Description
IocSend_<IocId>[_<SenderId>]	Performs data transmission for "Queued" (event semantic) communication type.
IocWrite_<IocId>[_<SenderId>]	Performs data transmission for "LastIsBest" (data semantic) communication type.
IocRead_<IocId>	Performs data reception for "LastIsBest" (data semantic) communication type.

Table continues on the next page...

Table 9-1. IOC Management Run-time Services (continued)

Service Name[1]	Description
IocReceive_<IocId>	Performs data reception for "Queued" (event semantic) communication type.
IocEmptyQueue_<IocId>	In case of queued communication identified by the <IocId> in the function name, the content of the IOC internal communication queue shall be deleted.

1. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication; a name of IOC object from OIL configuration file

   <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. It is separated from <IocId> with an underscore. In case of 1:1 communication, it shall be omitted.

9.3.3 Callback Function

The IOC optionally notifies the receiver as soon as the transferred data is available for access on the receiver side, by calling a configured callback function which gets provided by the user of the communication.

This callback function is called using the OS-internal ISR category 2. The ISR has the following parameters:

- The ISR priority is calculated by SysGen so that it is the highest priority of all ISRs of category 2, including System and Second Timers.

The user can define callback function for each IOC. The function is placed in the user application and its name added to IOC object definition as value of *RECEIVER_PULL_CB* attribute on receiver side. The IOC calls this function on the receiving side for each data transmission.

The callback function is usual user's function.

The callback function shall have next definition:

```
void <CallBackName> (void)
{
/* user code */
}
```

9.3.4 Usage of IOC

IOCs are identified via a symbolic name. This identifier is used for references to the IOC when the system service is used. The variables to hold IOC's copies must be defined within the user's code by means of using the regular C-language definitions.

For example, if the user defines the message Msg_A having type int, and Msg_B as some receiving message, then user's code may access message using the following statements:

```
int Msg_A, Msg_B = 3;
IocRead_MsgA ( &Msg_A );
if( Msg_A == 2 ) { Msg_B = 1; }
IocWrite_MsgB(&Msg_B );
```

Chapter 10
Error Handling and Special Routines

10.1 Error Handling and Special Routines

This chapter describes support provided to the user to debug an application and handle errors.

This chapter consists of the following sections:

- General
- Hook Routines
- Error Handling
- Start-up Routine
- Application Modes
- System Shutdown
- Programming Issues

10.2 General

The AUTOSAR Operating System provides the user with tools for error handling and simple debugging at run time. These are special hook routines with names specified by AUTOSAR OS that are to be written by the user. In this section, error handling at the system configuration stage is not considered; it is described in System Objects Definition.

10.3 Hook Routines

The AUTOSAR Operating System supports system specific *hook routines* to allow user-defined actions within the OS internal processing.

These hook routines in AUTOSAR OS are:

- Called by the operating system, in a special context depending on the implementation of the operating system
- Can not be preempted by tasks
- not interrupted by category 2 interrupt routines
- Part of the operating system, but user defined
- Implemented by the user
- Standardized in interface by AUTOSAR OS, but not standardized in functionality (environment and behavior of the hook routine itself), therefore usually hook routines are not portable
- Only allowed to use a subset of API functions
- Optional

In the AUTOSAR OS hook routines are intended for:

- System startup. The corresponding hook routine(s) (*StartupHook*) is called after the operating system startup and before the scheduler is running. Each OS-Application can has its own routine (*StartupHook_AppName*) in addition to System one.
- Tracing or application dependent debugging purposes as well as user defined extentions of the context switch
- Error handling. The corresponding hook routine (*ErrorHook*) is called if a system call returns a value not equal to *E_OK*. Each OS-Application can has its own routine (*errorHook_AppName*) in addition to System one.
- System shutdown. The corresponding hook routine(s) (*ShutdownHook*) is called. Each OS-Application can has its own routine (*ShutdownHook_AppName*) in addition to System one.

The NXP AUTOSAR OS provides the following hook routines: *ErrorHook*, *PreTaskHook*, *PostTaskHook*, *StartupHook*, *ShutdownHook*. The user must create the code of these routines, the OS only provides description of function prototypes.

- *ErrorHook* - this hook is called by the Operating System at the end of a system service which has a return value not equal to E_OK (see Error Interface). It is called before returning from the service. It is also called when an alarm expires and an error is detected during task activation or event setting. If the application specific ErrorHooks are defined they are called after the OS ErrorHook.
- *PreTaskHook* - this hook is called before the operating system enters the Task, but in the Task context. This hook is called from the scheduler when it passes control to the given task. It may be used by an application to trace the sequences and timing of the tasks' execution.
- *PostTaskHook* - This hook is called after the operating system leaves the Task, it is called in the Task context. It is called from the scheduler when it switches from the current task to another. It may be used by an application to trace the sequences and timing of tasks' execution.

- *PreIsrHook* - this hook is called before the operating system enters the ISR of category 2 or System Timer ISR. This hook is called in the context of the ISR, but with all interrupts disabled. It may be used for debugging purposes only. This hook is not defined in AUTOSAR OS v.4.0.3 specification, it is NXP OS extension of the AUTOSAR OS.
- *PostIsrHook* - This hook is called after the operating system leaves the ISR of category 2 or System Timer ISR. This hook is called in the context of the ISR, but with all interrupts disabled. It may be used for debugging purposes only. This hook is not defined in AUTOSAR OS v.4.0.3 specification, it is NXP OS extension of the AUTOSAR OS.
- *StartupHook* - This hook is called after the operating system startup and internal structures initialization and before initializing System Timer and running scheduler. It may be used by an application to perform initialization actions and task activation. If the application specific StartupHooks are defined they are called after the OS StartupHook.
- *ShutdownHook* - This hook is called when the service *ShutdownOS* has been called. It is called before the Operating System shuts down itself. If the application specific ShutdownHooks are defined they are called before the OS ShutdownHook.

These variables are not defined in AUTOSAR OS v.4.0.3 specification, it is NXP OS extension of the AUTOSAR OS.

The NXP AUTOSAR OS places error hooks code into the memory section '.ostext'.

10.4 Error Handling

10.4.1 Error Interface

The hook routine *ErrorHook* is provided to handle possible errors detected by the AUTOSAR Operating System. Its basic framework is predefined and must be completed by the user. This gives the user a choice of efficient centralized or decentralized error handling.

ErrorHook is called before the OS service returns error code not equal E_OK and from the OS kernel itself in case of wrong *Task* or *ISR* termination - for example the *Task* ends without call to *TerminateTask* or *ISR* does not release resource. *ErrorHook* is also called by the OS when the *ALARM* action can not be performed.

The special system routine *ShutdownOS* is intended to shut down the system. *ShutdownOS* may be called by the user on experiencing a fatal error. User hook *ShutdownHook*, if configured, is called by *ShutdownOS*. The Application-specific shutdown hooks, if configured, are called by the OS before the OS *ShutdownHook*.

The AUTOSAR OS *ErrorHook* is called with a parameter that specifies the error. It is up to the user to decide what to do, depending on the error has occurred. If the Application-specific error hooks are configured, they are called after the OS *ErrorHook*. According to AUTOSAR OS specification, if system service is called from *ErrorHook* user's hook and this service does not return E_OK error code, then *ErrorHook* is not called. Therefore nested *ErrorHook* calls are blocked by AUTOSAR OS.

The NXP AUTOSAR OS specifies the following error codes:

Table 10-1. NXP AUTOSAR OS Error Codes

Name	Value	Type
E_OK	0	No error, successful completion
E_OS_ACCESS	1	Access to the service/object denied
E_OS_CALLEVEL	2	Access to the service from the ISR is not permitted
E_OS_ID	3	The object ID is invalid
E_OS_LIMIT	4	The limit of services/objects exceeded
E_OS_NOFUNC	5	The object is not used, the service is rejected
E_OS_RESOURCE	6	The task still occupies the resource
E_OS_STATE	7	The state of the object is not correct for the required service
E_OS_VALUE	8	A value outside of the admissible limit
E_OS_MISSINGEND	12	Tasks terminates without a TerminateTask() or ChainTask() call
E_OS_DISABLEDINT	13	OS service is called inside an interrupt disable/enable pair
E_OS_STACKFAULT	14	Stack fault detected via stack monitoring by the OS
E_OS_SYS_FATAL	21	Fatal error in the OS code
E_OS_SYS_ORDER	23	Incorrect order of function calling
E_OS_PARAM_POINTER	29	A pointer argument to an API is null
IOC Error Codes		
IOC_E_OK	0	No error, successful completion
IOC_E_NOK	1	Error occurred. Used to identify error cases without error specification.
IOC_E_LIMIT	130	Overflow of FIFO associated with queued messages
IOC_E_LOST_DATA	64	The IOC service refused an IocSend request due to internal buffer overflow

Table continues on the next page...

Table 10-1. NXP AUTOSAR OS Error Codes (continued)

Name	Value	Type
IOC_E_NO_DATA	131	No data is available for reception in case of queued messages

Errors committed by the user in direct conjunction with the Operating System can be intercepted to a large extent via the Extended Status of the Operating System, and displayed. This results in an extended plausibility check on calling OS services.

10.4.1.1 Macros for Errorhook

The special macros are provided by OS to access the ID of the service that caused an error and it's first parameter (only if it is an object ID) inside *ErrorHook* routine:

- the macro *OSErrorGetServiceId()* returns the service identifier where the error has been risen. The service identifier is of type *OSServiceIdType*. Possible values are *OSServiceId_xxxx*, where xxxx is the name of the system service, if the error occurred not inside a service function but detected by the OS kernel then the special value *OSServiceId_NoService*[1] is returned.
- the macros of type <*OSError_serviceID_parameterID*()>, where serviceID is the name of the service and parameterID is the name of the first parameter, returns the value of the first parameter. For example, *OSError_ActivateTask_TaskID*() returns the task identifier if ErrorHook was called because of error detected in ActivateTask.

In order use the *OSErrorGetServiceId* macro, the user has to set the *USEGETSERVICEID* attribute to TRUE. To operate with macros for parameter access, the *USEPARAMETERACCESS* attribute shall be set to TRUE. The list of service identifiers and macros for parameter access are provided in *Quick Reference* section of the User Manual.

10.4.2 Extended Status

The AUTOSAR Operating System version with `Extended Status` requires more execution time and memory space than the release version due to the additional plausibility checks it offers. However, many errors can be found in a test phase. After they have all been eliminated, the system can be recompiled with `Standard Status` to the release version.

The following example can illustrate Extended Status usage:

1. it is NXP OS extension of the AUTOSAR OS, not specified in AUTOSAR OS v.4.0.3 specification

- If a task is activated in the release version, only 'OK' is returned. In the Extended Status version, the additional status like 'Task not defined', 'Task already activated' can be returned. These extended messages must no longer occur in the target application at the time of execution, i.e., the corresponding errors are not intercepted in the operating system's release version.

10.4.3 Possible Error Reasons

Errors in the application software are typically caused by:

- Errors on handling the operating system, i.e. incorrect configuration / initialization / dimensioning of the operating system or non-observance of restrictions regarding the OS service.
- Error in software design, i.e. unwise choice of task priorities, generation of deadlocks, unprotected critical sections, incorrect dimensioning of time, inefficient conceptual design of task organization, etc.

10.5 Start-up Routine

The special system routine *StartOS* is implemented in the AUTOSAR Operating System to allocate and initialize all dynamic system and application resources in RAM. This routine is called from the main() function of the application with the application mode as parameter (see Application Modes) and pass the control to the scheduler to schedule the first task to be running. User hook *StartupHook* is called after operating system startup and before the system and second timers initialization and running scheduler. See Sample Application for details.

The figure below shows system startup.

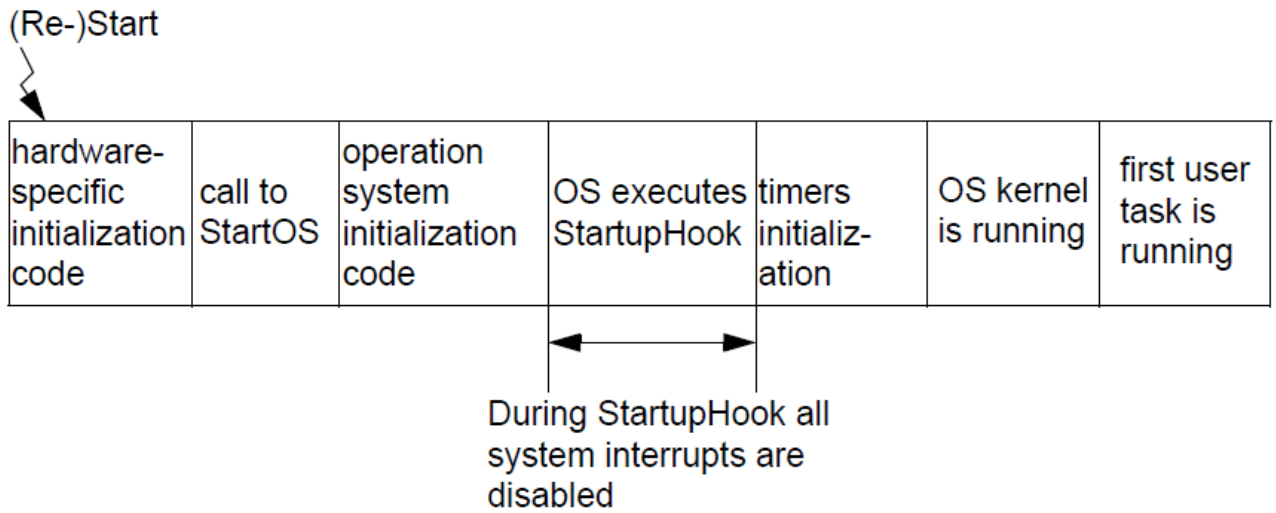

Figure 10-1. System Startup in the AUTOSAR OS

Initializing system and second timers after *StartupHook* avoids loss of timer interrupts during *StartupHook* execution.

If for any of timers the attribute *Prescaler* is set to *USER* then it is the User responsibility to initialize this timer hardware.

10.6 Application Modes

Application modes are intended to allow AUTOSAR OS to come under different modes of operation. The application modes differ in the set of autostarted Tasks and Alarms. Once the operating system has been started, it is not possible to change the application mode.

NXP AUTOSAR OS supports up to 8 application modes.

10.7 System Shutdown

The special *ShutdownOS* service exists in AUTOSAR OS to shut down the operating system. This service could be requested by an application or requested by the operating system due to a fatal error.

When *ShutdownOS* service is called with a defined error code, the operating system will shut down and call the hook routine *ShutdownHook*. The user is free to define any system behavior in *ShutdownHook* e.g. not to return from the routine. If *ShutdownHook* returns, the operating system enters with all interrupts disabled.

It is possible to restart OS by calling setjmp() before calling *StartOS*() and then calling longjmp() in *ShutdownHook*.

If some error occurs inside of OS code, ISR category1, ErrorHook() or in case of machine check exception, the OS calls *ShutdownOS* service with *E_OS_SYS_FATAL* status.

10.8 Programming Issues

10.8.1 Configuration Options

The following configuration options affect error handling and hook routines:

- *SCALABILITYCLASS*

  Specifies the Scalability Class. If AUTO, the Scalability class is defined according to the Application and Task definitions.

- *ERRORHOOK*

  If this option is turned on, the *ErrorHook* is called by the system for error handling

- *USEGETSERVICEID*

  If this option is turned on, it allows getting the service ID inside ErrorHook.

- *USEPARAMETERACCESS*

  If this option is turned on, it allows getting the first service parameter value inside ErrorHook.

- *PRETASKHOOK*

  If this option is turned on, the *PreTaskHook* is called by the system before context switching

- *POSTTASKHOOK*

  If this option is turned on, the *PostTaskHook* is called by the system before context switching

- *STARTUPHOOK*

If this option is turned on, the *StartupHook* is called by the system at the end of the system initialization

- *SHUTDOWNHOOK*

  If this option is turned on, the *ShutdownHook* is called by the system when the OS service *ShutdownOS* has been call

Chapter 11
System Configuration

11.1 System Configuration

This chapter describes possible AUTOSAR OS versions, configuration options and the configuration mechanism.

This chapter consists of the following sections:

- General
- Application Configuration File
- OILConcept

11.2 General

The AUTOSAR Operating System is fully statically configured. All system properties, the number of system objects and their parameters (characteristics of tasks, counters, alarms, IOCs, etc.), run time behavior are defined by the user. Such approach allows the user to create various range of applications with exactly defined characteristics. Different memory and performance requirements can be easily satisfied with such modular approach.

The AUTOSAR specifies usage of XML for configuration, but for the OS the OIL file also may be used. It provides compatibility with OSEK/VDX standard. OIL and XML OS configuration files contains the same information, but in different format.

All application parameters are defined in the special configuration file. This file must conform OIL grammar rules. It is processed by the separate *System Generator utility (SG)*. SG is written in Java and requires the Java run-time environment to be installed on the developer machine, minimum version 1.6. The System Generator analyzes statements in the configuration file and builds output C-language files needed to compile OS source files and to compile and link an application with the specified features. During its

execution SG reports to the user about the errors. The System Generator produces header and source code files that defines all properties and objects in terms of the C language. These files are to be compiled and linked together with the user's and OS source code.

11.3 Application Configuration File

Application configuration file contains the statements which define the system properties and objects. Such file can have any extention and the extention '.oil' is suggested by default. The file of this format is processed by the SG utility. The extention '.oin' is suggested for the included files. NXP AUTOSAR OS provides the special utility for conversion of the XML files, generated by General Configuration Editor, to OIL format.

As the result of application configuration file processing SG produces three types of standard C-language files as it is described in Application Configuration and optional ORTI file as it described in Using OS Extended Status for Debugging. SG produces two header files and one source file. These files provides the code for all system tables, descriptors, arrays etc. both in ROM and RAM according to the user specified application configuration.

11.4 OILConcept

OSEK Implementation Language (OIL) is the specially designed language for development of embedded applications based on OSEK and AUTOSAR concept. OIL is used to describe the application structure (application configuration) as a set of system objects with defined links. OIL allows the user to write an application configuration as a text file. These files have predefined structure and special (standard) grammar rules.

All system objects specified by AUTOSAR and relationships between them can be described using OIL. OIL defines standard types for system objects. Each object is described by a set of attributes and references.

All keywords, attributes, object names, and other identifiers are casesensitive.

11.4.1 OIL File

The OIL file contains two parts - one for the definition of implementation specific features (Implementation Definition) and another one for the definition of the structure of the application located on the particular CPU (Application Definition).

In the very beginning of an OIL file the number of the version of OIL is indicated. The keyword OIL_VERSION is used for this purpose. For example:

```
OIL_VERSION = "3.0";
```

11.4.2 OIL Format

The Standard OIL is intended to configure AUTOSAR OS Operating System (OS). It is strictly defined by the *OSEK/VDX System Generation OIL: OSEK Implementation Language, v.2.5, 01 July 2004* specification with additions defined in *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3*.

11.4.3 Implementation Definition

The Implementation Definition defines implementation specific features for the particular AUTOSAR OS implementation for which this application is developed.

The implementation can limit the given set of values for object attributes (e.g. restrict the possible OS conformance classes).

It is not allowed to exclude any standard attributes from the particular AUTOSAR OS implementation. Additional non-standard attributes can be defined for the objects for the particular AUTOSAR OS implementation.

The include mechanism (see Include Directive) can be used to define the implementation definition as a separate file. Thus corresponding implementation definition files are developed and delivered with particular AUTOSAR OS implementations and then included in user's OIL files. The NXP AUTOSAR OS/S32K implementation is described in the os_s32k file which is delivered in the package. Extention '.oin' is used for OIL files that should be used only as include files.

11.4.3.1 Implementation Definition Grammar

Implementation Definition part starts with keyword *IMPLEMENTATION* and implementation name.

The structure for Implementation Definition part is shown using the following syntax:

```
IMPLEMENTATION <name> {
    <object_descriptions>
};
```

All objects within Implementation Definition part are described using the same syntax.

```
<object_type> {
    <property_definitions>
};
```

Object type is defined by the object keyword. For NXP AUTOSAR OS/S32K implementation the following object types are implemented:

```
OS, APPMODE, APPLICATION, TASK, ISR, RESOURCE, EVENT, COUNTER, ALARM, SCHEDULETABLE, IOC
```

The set of object properties are to be defined within the object description. Both implementation specific attribute and reference shall be defined before it is used.

The attribute definition has the following structure:

```
<attr_type> [ WITH_AUTO ] [ <attr_range> ]
<attr_name> [ = <default_value> ] [
<multiple_specifier> ];
```

The attribute type and attribute value range (if it exists) shall be defined. The range of attribute values can be defined in two ways: either the minimum and maximum allowed attribute values are defined (the [0..12] style) or the list of possible attribute values are presented (like C enumeration type).

The WITH_AUTO specifier can be combined with any type. If WITH_AUTO can be specified it means that this attribute can have the value AUTO and the possibility of automatic assignment.

Data types defined for OIL are listed below. Note that these data types are not necessarily the same as the corresponding C data types.

- UINT32 - any unsigned integer number (possibly restricted to a range of numbers. This data type allows expressing any 32 bit value in the range of $[0..(2^{32}-1)]$.
- INT32 - any signed integer number in the range of $[-2^{31}..(2^{31}-1)]$.
- UINT64 - any unsigned integer number in the range $[0..(2^{64}-1)]$.
- INT64 - any signed integer number in the range $[-2^{63}..(2^{63}-1)]$.
- FLOAT - any floating point number according to IEEE-754 standard (Range: +/-1,176E-38 to +/-3,402E+38).
- ENUM - the list of possible values shall be presented. Any value from the list can be assigned to an attribute of this type. ENUM types can be parameterized, i.e. the particular enumerators can have parameters. The parameter specification is denoted in curly braces after the enumerator. Any kind of attribute type is allowed as parameter of an enumerator.

- BOOLEAN - The attribute of this type can have either TRUE or FALSE value. BOOLEAN types can be parameterized, i.e. the particular boolean values can have parameters. Parameter specification are denoted in curly braces after an explicit enumeration of the boolean values. Any kind of attribute type is allowed as parameter of a boolean value.
- STRING - Any 8-bit character sequence enclosed in double-quotes, but not containing double-quotes can be assigned to this attribute.

A reference is a special type of value intended to define links between system objects. The reference definition has the following structure:

```
<object_type> <reference_name> [<multiple_specifier> ];
```

The reference type is taken from the referenced object (e.g. a reference to a task shall use the TASK_TYPE keyword as reference type). A reference can 'point to' any system object.

Multiple reference is the possibility to refer to several objects of the same type with one OIL statement. For example the task can refer to several events. If the reference shall be defined as a 'multiple' reference then the '[]' brackets shall be present after the reference name.

An attribute can have a subattributes which are described in curly brackets.

11.4.4 Application Definition

In an application definition the AUTOSAR OS application is composed from a set of system objects. In general an application can contain more than one system object of the same type.

Since an application is performed on the CPU the entity called CPU is introduced as the top of the description. This entity encompasses all local objects such as tasks, alarms, etc. Therefore, CPU can be considered as a container for application objects. This concept is introduced to provide future OIL evolution towards to distributed system support. This entity is identified by the keyword CPU.

11.4.4.1 Object Definition

All objects are described using the same syntax.

```
<object_type> <object_name> {
    <property_definitions>
};
```

Objects are labeled by keywords which shall be written in the upper case. Object attributes and references are also labeled by the keywords. The keywords are introduced in System Objects Definition. After an object keyword the object name must follow. Name is combined from any symbols up to 32 symbols long.

A set of attributes and references belonging to an object is enclosed in curly brackets, like in C language.

All assignments are made via the '=' operator. Each statement ends with semicolon - ';' like in the C language. A reference is represented as a reference type keyword assigned with a name of the object referenced. If multiple reference pointed to the set of objects several references shall be used. Here is the example for task referencing to own events:

```
EVENT = MyEvent1;
EVENT = MyEvent2;
```

11.4.4.2 Include Directive

The preprocessing directive to include other OIL files is allowed in any place of the OIL file. The NXP AUTOSAR OS OIL files that is intended to be included into the user files has an extention '.oin'.

Include statement has the same syntax as in ANSI-C:

```
#include "path-spec"
#include <path-spec>
```

Path specification:

- The path-spec is a filename, optionally preceded by a directory specification, and it must point to an existing file. The syntax of the path-spec depends on the operating system on which SysGen runs.

Path resolution: ·

- Using the quoted form (**#include "path-spec"**), SysGen's preprocessor will look for include files in the same directory of the file that contains the include directive, and then in the directories of any files that include that file.

  The preprocessor then searches along the paths specified by the '-i' option, then along the paths specified by the **OSB_INCLUDE_DIR** environment variable.

- Using the angle-bracketed form (**#include <path-spec>**), SysGen's preprocessor is instructed to look for include files along the paths specified by '-i' option, then along the paths specified by the **OSB_INCLUDE_DIR** environment variable.

The preprocessor stops searching as soon as it finds a file with the given name. If a complete, unambiguous, path specification for the include file between the double quotation marks is used, the preprocessor searches only that path specification.

Caution: .

1. A newline is mandatory after each include directive. Failure to add a newline after an include directive may lead to unexpected behavior!
2. Complete path specifications may only be used with quoted include directives.

11.4.4.3 Comments

An OIL file may contain comments. The '/*' and '//' characters define the start of a comment. Any characters after '//' are considered as a comment and the end of line (EOL) terminates the comment. Any characters after '/ *' are considered as comments and the end of the comment is defined by '*/'. Nested comments are not allowed in OIL.

11.4.4.4 File Structure

Any file in the Standard OIL format describes an application for a single CPU and, in general, must have the following structure:

```
OIL_VERSION = <version>;
IMPLEMENTATION <name> {// Implementation definition
    <OBJECT_TYPE> {
        ...list of implementation specific object attributes...
    };
    ...
};

CPU <name> { // Definition of the application on CPU
    <OBJECT_TYPE> <object_name>
    { // System object definition
        <ATTRIBUTE> = <value>;
        <REFERENCE> = <object_name>;
        ... list of object attributes and references ...
    };
    ... list of objects ...
};
```

11.4.5 Configuration File Rules

The application configuration files must conform some simple rules to be successfully processed. The rules are:

- All objects are described using the OIL syntax;
- Each object must have a unique name. Each object may be divided into several parts;

- All object names are accessible from the application;
- An attribute defines some object properties (for example, the task priority). Attributes that take a single value may only be specified once per object. Attributes that take a list of values must be specified as multiple statements;
- An object can have a set of references to other system objects. Per object there may be more than one reference to the same type of object;
- Values must be defined for all standard attributes of all objects, except for multiple attributes which can be empty;
- All statements must be written without errors;
- It is recommended to avoid conflicting statements (e.g., in SC1 the STACKSIZE is defined and no EVENT(s) is defined for a task) since it leads to error or warning messages.

Chapter 12
System Objects Definition

12.1 System Objects Definition

This chapter describes objects that are controlled by the Operating System - tasks, resources, alarms, IOCs, counters, ISRs and even the OS itself - are considered as system objects.

This chapter consists of the following sections:

- General
- OS Definition
- OS-Application Definition
- Task Definition
- ISR Definition
- Resource Definition
- Event Definition
- Counter Definition
- Alarm Definition
- Schedule Table Definition
- Inter-OS-application Communicator (IOC) Definition
- Application Modes Definition

12.2 General

All objects that are controlled by the Operating System - tasks, resources, alarms, IOCs, counters, ISRs and even the OS itself - are considered as system objects. Each of them has its unique characteristics defined by the user. To specify parameters for each system object the special statements are used for each object. All statements are described below in detail.

Each group of attributes has the scheme which describes attributes nesting. Possible attribute values are placed in angle brackets and separated by slash. Default value is marked as bold text. If default value is present, the attribute can be omitted. Scheme includes all attributes, but some of them can be used if parent attribute has the determined value only. The description of these dependencies is included into attribute definitions below.

12.2.1 OIL Attributes Names Mapping to Xml Configuration

The AUTOSAR prescribes usage of XML notation for all BSW modules, including the OS. In each OIL attribute description the correspondent XPath name is provided in square brackets.

Some attributes have different types: UINT64 in OIL and FLOAT in AUTOSAR XML.

SysGen converts values of these attributes from EPC (AUTOSAR XML) to OIL by multiplying * 1000000000 (seconds to nanoseconds) and from OIL to EPC (AUTOSAR XML) by dividing to 1000000000 (nanosecond to seconds).

Table 12-1 shows the list of these attributes.

Table 12-1. Converted attributes

AUTOSAR XML attributes (FLOAT)	OIL attributes (UINT64)
OsTaskAllInterruptLockBudget	MAXALLINTERRUPTLOCKTIME
OsTaskExecutionBudget	EXECUTIONBUDGET
OsTaskOsInterruptLockBudget	MAXOSINTERRUPTLOCKTIME
OsTaskTimeFrame	TIMEFRAME
OsTaskResourceLockBudget	MAXRESOURCELOCKTIME
OsSecondsPerTick	SECONDSPERTICK
OsTimeValue	NS
OsIsrAllInterruptLockBudget	MAXALLINTERRUPTLOCKTIME
OsIsrTimeFrame	TIMEFRAME
OsIsrResourceLockBudget	MAXRESOURCELOCKTIME
OsIsrExecutionBudget	EXECUTIONTIME
OsIsrOsInterruptLockBudget	MAXOSINTERRUPTLOCKTIME

12.3 OS Definition

The *Operating System* is the mandatory object for any application. This object is used to define OS configuration parameters. The keyword OS is used for this object type. Only one OS object can be defined in the application. See Operating System Architecture for more detailed information about OS. The syntax of the OS object definition is as follows:

```
OS <name>{
    <attributes>
};
```

OS object's attributes can be divided into the groups which correspond to appropriate system objects and their interaction. Nested structure of OS object definition is displayed on the syntax schemes of each attribute group.

Different groups of related attributes are described below. Brief explanations are provided. All these attributes should be defined inside the scope of the OS object.

12.3.1 Global System Attributes

This group of attributes represents system features which are common for the whole system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
STATUS = <STANDARD / EXTENDED>;
SCALABILITYCLASS = <SC1 / AUTO>;
CC = <BCC1 / ECC1 / AUTO>;
ISRFLOATINGPOINT = <TRUE / FALSE>;
DEBUG_LEVEL = <0 / 1 / 2 / 3 / 4>;
USERESSCHEDULER = <TRUE / FALSE>;
```

- ***STATUS*** (ENUM)

  [OsOS/OsStatus]

  The attribute specifies the debugging ability of OS, defines whether additional run-time error checks are supported by OS or not. If the system has the *Extended Status* additional checks are performed to detect run-time errors. The *Extended Status* adds approximately 20% of code, and increases timing accordingly. This mode is considered to be a debugging mode for SC1. In the *Standard Status* OS performs only very limited set of checks, the performance is increased and the amount of consumed memory is decreased.

  As a general approach, it is recommended to start application development with the *extended status* to discover configuration and run-time problems. For debugged applications the *status* may be set to *standard* to reduce code size and improve timing.

- ***SCALABILITYCLASS*** (ENUM or AUTO)

  [OsOS/OsScalabilityClass]

  The attribute specifies the Scalability Class which is supported by the OS. Class SC1 is the base class.

  If the value is AUTO then scalability class is defined.

- ***CC*** (ENUM or AUTO)

  [OsOS/OsCC]

  This NXP AUTOSAR OS specific attribute specifies the conformance class which is supported by the OS. However all features of the OS may be selected by means of using other OS additional properties. Therefore, even for given conformance class the functionality may be reduced or increased according to user's needs.

  If the value is AUTO then conformance class is defined according to TASKs definitions.

- ***ISRFLOATINGPOINT*** (BOOLEAN)

  [OsOS/OsIsrUseFloatingPoint]

  This attribute specifies whether the Floating-point unit in the interrupt handler (IRQ only) should be supported or not.

- ***DEBUG_LEVEL*** (ENUM)

  [OsOS/OsDebugLevel]

  This NXP AUTOSAR OS specific attribute specifies the ORTI support in OS. If the system has the DEBUG_LEVEL = 0, ORTI is not supported. If the attribute is set to 1 or higher, the static ORTI mode is turned on. If the attribute is set to 2 or 4, static and run time ORTI (running Task, ISR and system calls identification) modes will be used. When DEBUG_LEVEL = 3 the static ORTI and running Task and ISR identification are supported (system calls tracing is not supported). The value 4 is retained for compatibility with previous versions of OSEKturbo. See Debugging Application for details.

- ***USERESSCHEDULER*** (BOOLEAN)

  [OsOS/OsUseResScheduler]

  This attribute specifies whether *RES_SCHEDULER* should be supported or not.

12.3.2 Multi-Core specific Attributes

The attribute should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
NumberOfCores = <1 / 2>;
```

- ***NumberOfCores*** (ENUM)

  [OsOS/OsNumberOfCores]

  This attribute specifies maximum number of cores that are controlled by the OS. Possible values are 1 or 2.

12.3.3 Memory Related Attributes

This group of attributes is NXP OS extension of the AUTOSAR OS and describes the physical memory available on ECU and the memory used by the OS itself. See also OS-Application Definition for Application memory definition.

NOTE

If at least one of memory attributes is set to *TRUE* then the linker file shall be defined as a SysGen input

These attributes may be omitted in SC1.

The attributes are defined in the scope of the *TargetMCU* object (see CPU Related Attributes) in accordance with the following syntax:

```
TargetMCU = <S32K> {
    InternalROM = <TRUE / FALSE> {
        Address = <integer>;


        Size = <integer>;

    };
    InternalRAM = <TRUE / FALSE> {

        Address = <integer / <start address of Internal RAM> >;


        Size = <integer / <size of Internal RAM> >;

    };
    InternalRAM2 = <TRUE / FALSE> {
        Address = <integer / <start address of Internal RAM2 if its available> >;
        Size = <integer / <size of Internal RAM2 if its available> >;
    };
    ExternalROM = <TRUE / FALSE> {
        Address = <integer>;
        Size = <integer>;
```

```
    };
    ExternalRAM = <TRUE / FALSE> {
        Address = <integer>;
        Size = <integer>;
    };
};
```

- ***InternalROM*** (BOOLEAN)

  [OsOS/OsTargetMCU/OsInternalROM]

  This NXP AUTOSAR OS specific attribute specifies is internal MCU ROM is used by the OS or user application.

- ***InternalRAM*** (BOOLEAN)

  [OsOS/OsTargetMCU/OsInternalRAM]

  This NXP AUTOSAR OS specific attribute specifies is internal MCU RAM is used by the OS or user application.

- ***InternalRAM2*** (BOOLEAN)

  [OsOS/OsTargetMCU/OsInternalRAM2]

  This NXP AUTOSAR OS specific attribute specifies is internal MCU RAM is used by the OS or user application.

- ***ExternalROM*** (BOOLEAN)

  [OsOS/OsTargetMCU/OsExternalROM]

  This NXP AUTOSAR OS specific attribute specifies is external ROM is used by the OS or user application.

- ***ExternalRAM*** (BOOLEAN)

  [OsOS/OsTargetMCU/OsExternalRAM]

  This NXP AUTOSAR OS specific attribute specifies is external RAM is used by the OS or user application.

- ***Address*** (UINT32)

  [OsOS/OsTargetMCU/Os{Int|Ext}ernal{ROM|RAM}/OsAddress]

  This NXP AUTOSAR OS specific attribute specifies the address of memory area.

- ***Size*** (UINT32)

  [OsOS/OsTargetMCU/Os{Int|Ext}ernal{ROM|RAM}/Size]

This NXP AUTOSAR OS specific attribute specifies the size of memory area in bytes.

12.3.4 CPU Related Attributes

This group of attributes provides with possibility to tune the selected hardware. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
TargetMCU = <S32K> {
    ClockFrequency = <integer / 12000>;
    ClockDivider = <integer / 1>;
    ClockMultiplier = <integer / 1>;
    SysTimer = <HWCOUNTER / SWCOUNTER / NONE> {
        COUNTER = <name of COUNTER>;
        ISRPRIORITY = <integer>;
        Period = <integer / AUTO>;
        TimerHardware = <name of hardware timer> {
            GlobalFTMPrescaler = <USER / OS> {
                Value = <integer / AUTO>;
            };
            Channel = <integer>
            TimerModuloValue = <integer / AUTO>;
            Freeze = TRUE / FALSE ;
            PeripheralClockDivider = <integer / 1>;
        };
    };
    SecondTimer = <HWCOUNTER / SWCOUNTER / NONE> {
        COUNTER = <name of COUNTER>;
        ISRPRIORITY = <integer>;
        Period = <integer / AUTO>;
        TimerHardware = <name of hardware timer> {
            GlobalFTMPrescaler = <USER / OS / AUTO> {
                Value = <integer / AUTO>;
            };
            Channel = <integer>;
            TimerModuloValue = <integer / AUTO>;
            PeripheralClockDivider = <integer / 1>;
            Freeze = TRUE / FALSE / AUTO;
        };
    };
};
```

- ***TargetMCU*** (ENUM)

  [OsOS/OsTargetMCU]

  This NXP AUTOSAR OS specific attribute defines the target for which the OS will be configured.

- ***ClockFrequency*** (UINT32)

  [OsOS/OsTargetMCU//OsClockFrequency]

This NXP AUTOSAR OS specific attribute specifies oscillator frequency in kHz for calculating prescaler value and timer modulo value. This attribute shall be defined if any of Period, Prescaler/Value or/and any TimerModuloValue is AUTO. Otherwise this attribute is used to calculate OSTICKDURATION.

- ***ClockDivider*** (UINT32)

  [OsOS/OsTargetMCU/OsClockDivider]

  This NXP AUTOSAR OS specific attribute specifies PLL divider for calculating input timer frequency. It is the user responsibility to initialize hardware to appropriate values. Value of the attribute shall be equal to actual divider, not the value written into the HW register.

- ***ClockMultiplier*** (UINT32)

  [OsOS/OsTargetMCU/OsClockMultiplier]

  This NXP AUTOSAR OS specific attribute specifies PLL multiplier for calculating input timer frequency. It is the user responsibility to initialize PLL hardware with appropriate values. Value of the attribute shall be equal to actual multiplier, not the value written into the HW register. In general, the value of (ClockFrequency * ClockMultiplier / ClockDivider) shall be equal to actual frequency of system clock.

- ***SysTimer***(ENUM)

  [OsOS/OsTargetMCU/OsSysTimer]

  This NXP AUTOSAR OS specific attribute specifies whether the internal OS system timer is used or not.

  If *SysTimer* is set to SWCOUNTER or HWCOUNTER, interrupt services are turned on automatically.

  If this attribute is SWCOUNTER or HWCOUNTER, then specific subattributes can be defined for the system timer.

- ***SecondTimer*** (ENUM)

  [OsOS/OsTargetMCU/OsSecondTimer]

  This NXP AUTOSAR OS specific attribute specifies whether the internal OS second timer is used or not.

If this attribute is SWCOUNTER or HWCOUNTER, then specific subattributes can be defined for the second timer. The *SecondTimer* attribute and its subattributes can be defined only if *SysTimer* value is equal to SWCOUNTER or HWCOUNTER. The *SecondTimer* attribute can not be set to HWCOUNTER if *SysTimer* value is not HWCOUNTER.

- ***COUNTER*** (reference)

  [OsCounter]

  The reference specifies the *COUNTER* which shall be attached to the system or second timer. This attribute shall be defined for system timer and for second timer if respectively *SysTimer* value and *SecondTimer* value is SWCOUNTER or HWCOUNTER. The same counter can not be attached to both System and Second timers.

- ***ISRPRIORITY*** (UINT32)

  [.../{HWCOUNTER | SWCOUNTER}/OsIsrPriority]

  The attribute specifies priority of the system timer (second timer) interrupt . This attribute shall be defined inside the scope of the *SysTimer* and *SecondTimer* attributes if these attributes are set to SWCOUNTER or HWCOUNTER. The rules defined for *PRIORITY* attribute of ISR object are also applicable for this parameter. The value of the attribute is to be set in the range [1..14].

- Period (UINT32)

  [.../{HWCOUNTER | SWCOUNTER}/OsPeriod]

  This NXP AUTOSAR OS specific attribute specifies period of a tick of the system (second) counter in nanoseconds. This attribute can be defined inside the scope of the SysTimer and SecondTimer attributes if these attributes are set SWCOUNTER or HWCOUNTER.

  The Period attribute shall be defined if the corresponded GlobalFTMPrescaler/ Value or/and TimerModuloValue is AUTO. This attribute is ignored if corresponded GlobalFTMPrescaler/Value and TimerModuloValue are not AUTO.

NOTE

OSTICKDURATION and *OSTICKDURATION2* constants are calculated from the *SysTimer/Period* value or *SecondTimer /Period* value respectively if any of corresponding *GlobalFTMPrescaler/Value* and *TimerModuloValue* is AUTO. Otherwise *OSTICKDURATION* and *OSTICKDURATION2* are

calculated from the values of the corresponding *GlobalFTMPrescaler/Value* and *TimerModuloValue* attributes.

- ***TimerHardware*** (ENUM)

  [.../{HWCOUNTER | SWCOUNTER}/OsTimerHardware]

  This NXP AUTOSAR OS specific attribute is intended to select the hardware interrupt source for the system counter or the second counters (among the accessible MCU devices). This attribute inside the *SysTimer* and *SecondTimer* attributes can be defined if the values of these attributes are HWCOUNTER or SWCOUNTER only. The possible values for SWCOUNTER are SYSTICK and FTM(0/1/2/3/4/5/6/7); the possible values for HWCOUNTER: FTM(0/1/2/3/4/5/6/7).

  The *TimerHardware* attributes in *SysTimer* and *SecondTimer* blocks can have the same value if *Channel* value is different. See Timer Hardware for details about possible meanings of these parameters.

- ***Channel*** (UINT32)

  [.../{FTM}/OsChannel]

  The attribute specifies timer channel number. The attribute shall be defined inside the scope of the *SysTimer* and *SecondTimer* attributes. If the *TimerHardware* attribute is set to its allowed values then the *Channel* attribute allowed values are:
  - For FTM(0/1/2/3/4/5/6/7) valid values are [0..7]

  If the *TimerHardware* attributes of the *SysTimer* and *SecondTimer* are the same then the *Channel* values shall be different.

- ***GlobalFTMPrescaler*** (ENUM)

  [.../{FTM}/OsGlobalFTMPrescaler]

  This ***NXP AUTOSAR OS*** specific attribute specifies whether global prescaler value shall be initialized during OS startup or prescaler will be set by the user application.

  The *GlobalFTMPrescaler* attribute inside the *SysTimer* attribute scope and the *GlobalFTMPrescaler* attribute inside the *SecondTimer* attribute shall have the same value (USER or OS).

- ***Value*** (UINT32 or AUTO) – inside GlobalFTMPrescaler attribute.

  [.../OsGlobalFTMPrescaler/{OS | USER}/OsValue]

This NXP AUTOSAR OS specific attribute specifies initial global prescaler value for the selected system or second timer hardware. The value of this attribute is fully hardware-dependent. For more details see ARM Architecture Platform-Specific Features. Note that the *GlobalFTMPrescaler/Value* attribute value is not equal to divide factor of timer hardware.

This attribute can be *AUTO* only if *GlobalFTMPrescaler* value is *OS*. In case of *AUTO*, global prescaler value is calculated during system generation. This attribute shall be defined if *GlobalFTMPrescaler* value is USER. If the *TimerModuloValue* attribute is not *AUTO*, the *Value* shall be defined by numeric value. In case of *AUTO* global prescaler value is calculated from the values of *ClockFrequency*, *ClockDivider*, *ClockMultiplier* and corresponding *Period* attributes during system generation.

If the *GlobalFTMPrescaler* is set to USER then it's value is used by OS only for time calculations, assuming that the appropriate HW register are set by User before call to StartOS.

- ***TimerModuloValue***(UINT32)

  [.../{FTM | SYSTICK}/OsTimerModuloValue]

  This NXP AUTOSAR OS specific attribute specifies timer modulo value. The value of this attribute is fully hardware-dependent. For more details see ARM Architecture Platform-Specific Features.

  This attribute can be defined if *SysTimer* or *SecondTimer* value is *SWCOUNTER* only. If the attribute is set *AUTO*, timer modulo value is calculated from the values of the *ClockFrequency*, and corresponded *Period* attributes during system generation.

- ***Freeze*** (BOOLEAN)

  [.../{FTM}/OsFreeze]

  This attribute specifies timer freeze bit value. If this attribute has value TRUE, the timer will be frozen while the MCU has the debugger connected.

- ***PeripheralClockDivider*** (UINT32)

  [.../{FTM}/ OsPeripheralClockDivider]

  This NXP AUTOSAR OS specific attribute specifies the divider value that is set by the User in application code before calling StartOS(). Valid values are 1..16. Values of the attribute shall be equal for the System and Second Timers if the same TimerHardware is used. Default value is 1. It is used for Timers period calculation.

12.3.5 Stack Related Attributes

These attributes define stack support in the system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
IsrStackSize = <integer / AUTO>;
STACKMONITORING = <TRUE / FALSE> {
    Pattern = <integer / 0x55555555>;
    PatternSize = <1 / 2 / 4>;
};
```

- ***IsrStackSize*** (UINT32)

  [OsOS/OsIsrStackSize]

  This NXP AUTOSAR OS specific attribute specifies the ISR stack size in bytes. This stack is used for all ISRs category 2 and OS Timers ISRs in ECC1 within SC1. In this case the current status of the processor is saved onto the current stack, and stack is switched to the interrupt stack. This stack should be big enough to hold all nested interrupt stack frames in the system. In BCC1 within SC1 the Single Stack is used for Basic tasks and ISRs and this attribute is ignored.

- ***STACKMONITORING*** (BOOLEAN)

  [OsOS/OsStackMonitoring]

  This attribute turns on stack overflow runtime checking and stack usage services in all classes. OS performs check of the main, task and ISR stacks on context switch. If OS is configured to perform stack check then the *ProtectionHook* (or *ShutdownOS*, depending on Scalability Class) is called with *E_OS_STACKFAULT* error status code when OS detects stack overflow..

- ***Pattern*** (UINT32)

  [OsOS/OsStackMonitoring/TRUE/OsPattern]

  This NXP AUTOSAR OS specific attribute defines the pattern used to fill stack memory. Possible values are 0..0xFFFFFFFF

- ***PatternSize*** (ENUM)

  [OsOS/OsStackMonitoring/TRUE/OsPatternSize]

  This NXP AUTOSAR OS specific attribute defines the number of patterns (each pattern is 32 bit width) to be checked.

12.3.6 Hook Routines Related Attributes

These attributes define hook routines support in the system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
STARTUPHOOK = <TRUE / FALSE>;
SHUTDOWNHOOK = <TRUE / FALSE>;
PRETASKHOOK = <TRUE / FALSE>;
POSTTASKHOOK = <TRUE / FALSE>;
IsrHooks = <TRUE / FALSE>;
ERRORHOOK = <TRUE / FALSE>;
USEGETSERVICEID = <TRUE / FALSE>;
USEPARAMETERACCESS = <TRUE / FALSE>;
PROTECTIONHOOK = <TRUE / FALSE>;
```

- ***STARTUPHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsStartupHook]

  The attribute defines whether the user's-provided hook is called by the system after startup but before starting dispatcher and initializing system timer or not (the *StartupHook* hook routine). This hook may be used by an application to perform hardware initialization actions, etc.

  The alternative way is to make such initialization steps in the task, which starts automatically. The hook is called with disabled interrupts.

- ***SHUTDOWNHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsShutdownHook]

  The attribute defines whether the user's-provided hook is called by the system during system shutdown or not (the *ShutdownHook* hook routine). The main purpose of this hook is to shutdown initialized hardware devices like timers, network controllers, etc. Besides, the reason for shutdown may be obtained through the error code.

  This hook is called after system timer shutdown (if system timer is configured in the system). Interrupts are disabled in this hook.

- ***PRETASKHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsPreTaskHook]

  The attribute defines whether the user's-provided hook is called from the scheduler code before the operating system enters context of the task or not (the *PreTaskHook* hook routine). In general, this hook is designed for debugging applications by means of tracing current task.

It is not recommended to use this hook in normal working applications, because it adds significant timing overhead. If the attribute is defined, this hook is called for each task, i.e. it is not allowed to use this hook for particular task(s) only.

- ***POSTTASKHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsPostTaskHook]

  The attribute defines whether the user's-provided hook is called from the scheduler code after the operating system leaves context of the task or not (the *PostTaskHook* hook routine). In general, this hook is designed for debugging applications by means of tracing current task.

  It is not recommended to use this hook in normal working applications, because it adds significant timing overhead. If the attribute is defined, this hook is called for each task, i.e. it is not allowed to use this hook for particular task(s) only.

- ***IsrHooks*** (BOOLEAN)

  [OsOS/OsHooks/OsIsrHooks]

  This NXP AUTOSAR OS specific attribute specifies whether the user-defined *PreIsrHook* and *PostIsrHook* functions are called before and after each ISR of category 2 and OS Timers ISRs.

- ***ERRORHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsErrorHook]

  The attribute defines whether the user's-provided hook is called by the system at the end of each system service which returns status not equal to E_OK. (the *ErrorHook* hook routine). This hook is designed for debugging applications by means of tracing error code, returned by the system service instead of checking error code after each call of system service. This hook increases the OS code size by approximately 10% and increases the timing in case of error during the service call.

  There is no need to check the error status of the each OS service call if this hook is used. This hook is useful as a temporary feature of a working (debugged) applications when some troubles occur. If the attribute is defined, this hook is called from the system service in which error occurs, i.e. it is not allowed to use this hook for particular service(s) only.

- ***USEGETSERVICEID*** (BOOLEAN)

  [OsOS/OsUseGetServiceId]

This attribute specifies the possibility of usage the access macros to the service ID in the error hook.

- ***USEPARAMETERACCESS*** (BOOLEAN)

  [OsOS/OsUseParameterAccess]

  This attribute specifies the possibility of usage the access macros to the context related information in the error hook.

- ***PROTECTIONHOOK*** (BOOLEAN)

  [OsOS/OsHooks/OsProtectionHook]

12.4 OS-Application Definition

APPLICATION object is used in the OIL file to define OS-Application properties and data. It defines other system objects that are owned by OS-Application and data memory placement. Links with other system objects are defined via references.

Only trusted OS-Applications are allowed in SC1 and SC2 classes.

```
APPLICATION <name of APPLICATION>{
    CORE = <0>;
    TRUSTED = <FALSE / TRUE>{
        TRUSTED_FUNCTION = <TRUE / FALSE>{
            NAME = <name of FUNCTION>;
        };
    };
    STARTUPHOOK = <TRUE / FALSE>;
    SHUTDOWNHOOK = <TRUE / FALSE>;
    ERRORHOOK = <TRUE / FALSE>;
    HAS_RESTARTTASK = <TRUE / FALSE>{
        RESTARTTASK = <name of TASK>;
    };
    Peripheral = <TRUE / FALSE>{
        Address = <integer>;
        Size  = <integer>;
    };
    TASK = <name of TASK>;
    ISR = <name of ISR>;
    ALARM = <name of ALARM>;
    SCHEDULETABLE = <name of SCHEDULETABLE>;
    COUNTER = <name of COUNTER>;
};
```

12.4.1 Attributes

The APPLICATION object has the following attributes:

- ***CORE*** (ENUM)

[OsApplication/OsApplicationCoreAssignment]

The attribute defines ID of the core onto which the OS-Application is bound.

- ***TRUSTED*** (BOOLEAN)

  [OsApplication/OsTrusted]

  The attribute defines whether the OS-Application is trusted or not. It shall be set to *FALSE* in SC1..SC2.

- ***TRUSTED_FUNCTION*** (string)

  [OsApplication/OsTrusted/TRUE/OsTrusted_function]

  The attribute defines whether the *TRUSTED APPLICATION* has a 'public' trusted functions that may be called from other OSApplications.

- ***NAME*** ()

  [OsApplication/OsTrusted/TRUE/OsTrusted_function/TRUE/OsName]

  The attribute defines the name of the trusted function.

- ***STARTUPHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppStartupHook]

  The attribute defines whether the user-provided application-specific hook (StartupHook_<App>) is called by the system after system *StartupHook* (if configured).

  The hook is called with all interrupts disabled.

- ***SHUTDOWNHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppShutdownHook]

  The attribute defines whether the user-provided application-specific hook (ShutdownHook_<App>) is called by the system during system shutdown before the OS-wide *ShutdownHook* (if configured).

  The hook is called with all interrupts disabled.

- ***ERRORHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppErrorHook]

  The attribute defines whether the user-provided application-specific error hook (ErrorHook_<App>) is called by the OS after the system ErrorHook (if configured).

The hook is called with OS (category 2) interrupts disabled.

- ***HAS_RESTARTTASK*** (BOOLEAN)

  [OsApplication/OsHas_restarttask]

  The attribute defines whether the OS-Application has a 'restart' *TASK* wich is called at the OS-Application restart.

- ***RESTARTTASK*** (reference)

  [OsApplication/OsRestartTask]

  The attribute defines the name of TASK to be restarted.

- ***TASK*** (reference)

  [OsApplication/OsTask]

  This reference is used to define a task owned by the OS-Application. There can be several *TASK* references. This parameter can be omitted.

- ***ISR*** (reference)

  [OsApplication/OsIsr]

  This reference is used to define an ISR owned by the OS-Application. There can be several *ISR* references. This parameter can be omitted.

- ***ALARM*** (reference)

  [OsApplication/OsAlarm]

  This reference is used to define an alarm owned by the OS-Application. There can be several *ALARM* references. This parameter can be omitted.

- ***SCHEDULETABLE*** (reference)

  [OsApplication/OsScheduletable]

  This reference is used to define a scheduletable owned by the OSApplication. There can be several *SCHEDULETABLE* references. This parameter can be omitted.

- ***COUNTER*** (reference)

  [OsApplication/OsCounter]

  This reference is used to define a counter owned by the OSApplication. There can be several *COUNTER* references. This parameter can be omitted.

- ***Peripheral*** (BOOLEAN)

[OsApplication/OsPeripheral]

This NXP AUTOSAR OS specific attribute defines whether the peripheral address spaces (regions) is configured for non-trusted OS-Application.

- ***Address*** (UINT32)

  [OsApplication/OsPeripheral/OsAddress]

  This NXP AUTOSAR OS specific attribute specifies the address of peripheral memory area. The address shall be multiple of 32.

- ***Size*** (UINT32)

  [OsApplication/OsPeripheral/Size]

  This NXP AUTOSAR OS specific attribute specifies the size of peripheral memory area in bytes. The size shall be more than 32 bytes and divisible by 32.

12.4.1.1 Attributes

The APPLICATION object has the following attributes:

- ***CORE*** (ENUM)

  [OsApplication/OsApplicationCoreAssignment]

  The attribute defines ID of the core onto which the OS-Application is bound.

- ***TRUSTED*** (BOOLEAN)

  [OsApplication/OsTrusted]

  The attribute defines whether the OS-Application is trusted or not. It shall be set to *FALSE* in SC1..SC2.

- ***TRUSTED_FUNCTION*** (string)

  [OsApplication/OsTrusted/TRUE/OsTrusted_function]

  The attribute defines whether the *TRUSTED APPLICATION* has a 'public' trusted functions that may be called from other OSApplications.

- ***NAME*** ()

  [OsApplication/OsTrusted/TRUE/OsTrusted_function/TRUE/OsName]

  The attribute defines the name of the trusted function.

- ***STARTUPHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppStartupHook]

  The attribute defines whether the user-provided application-specific hook (StartupHook_<App>) is called by the system after system *StartupHook* (if configured).

  The hook is called with all interrupts disabled.

- ***SHUTDOWNHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppShutdownHook]

  The attribute defines whether the user-provided application-specific hook (ShutdownHook_<App>) is called by the system during system shutdown before the OS-wide *ShutdownHook* (if configured).

  The hook is called with all interrupts disabled.

- ***ERRORHOOK*** (BOOLEAN)

  [OsApplication/OsAppHooks/OsAppErrorHook]

  The attribute defines whether the user-provided application-specific error hook (ErrorHook_<App>) is called by the OS after the system ErrorHook (if configured).

  The hook is called with OS (category 2) interrupts disabled.

- ***HAS_RESTARTTASK*** (BOOLEAN)

  [OsApplication/OsHas_restarttask]

  The attribute defines whether the OS-Application has a 'restart' *TASK* wich is called at the OS-Application restart.

- ***RESTARTTASK*** (reference)

  [OsApplication/OsRestartTask]

  The attribute defines the name of TASK to be restarted.

- ***TASK*** (reference)

  [OsApplication/OsTask]

  This reference is used to define a task owned by the OS-Application. There can be several *TASK* references. This parameter can be omitted.

- ***ISR*** (reference)

  [OsApplication/OsIsr]

This reference is used to define an ISR owned by the OS-Application. There can be several *ISR* references. This parameter can be omitted.

- ***ALARM*** (reference)

  [OsApplication/OsAlarm]

  This reference is used to define an alarm owned by the OS-Application. There can be several *ALARM* references. This parameter can be omitted.

- ***SCHEDULETABLE*** (reference)

  [OsApplication/OsScheduletable]

  This reference is used to define a scheduletable owned by the OSApplication. There can be several *SCHEDULETABLE* references. This parameter can be omitted.

- ***COUNTER*** (reference)

  [OsApplication/OsCounter]

  This reference is used to define a counter owned by the OSApplication. There can be several *COUNTER* references. This parameter can be omitted.

- ***Peripheral*** (BOOLEAN)

  [OsApplication/OsPeripheral]

  This NXP AUTOSAR OS specific attribute defines whether the peripheral address spaces (regions) is configured for non-trusted OS-Application.

- ***Address*** (UINT32)

  [OsApplication/OsPeripheral/OsAddress]

  This NXP AUTOSAR OS specific attribute specifies the address of peripheral memory area. The address shall be multiple of 32.

- ***Size*** (UINT32)

  [OsApplication/OsPeripheral/Size]

  This NXP AUTOSAR OS specific attribute specifies the size of peripheral memory area in bytes. The size shall be more than 32 bytes and divisible by 32.

12.4.2 Task Definition

Task object is used in the OIL file to define task data. Attributes of this object define task properties. Links with other system objects are defined via references. The keyword TASK is used for this object type. See Task Management for more detailed information about the Tasks. The syntax of the task object is as follows:

```
TASK <name of TASK> {
    PRIORITY = <integer>;
    SCHEDULE = <FULL / NON>;
    AUTOSTART = <TRUE / FALSE>{
        APPMODE = <name of APPMODE>;
    };
    ACTIVATION = <1>;
    STACKSIZE = <integer> / AUTO;
    RESOURCE = <name of RESOURCE>;
    EVENT = <name of EVENT>;
    TIMING_PROTECTION = < TRUE / FALSE >{
        EXECUTIONBUDGET = <integer>;
        TIMEFRAME = <integer>;
        MAXOSINTERRUPTLOCKTIME = <integer>;
        MAXALLINTERRUPTLOCKTIME = 0;
        LOCKINGTIME = RESOURCELOCK{
            RESOURCE = <name of RESOURCE>;
            MAXRESOURCELOCKTIME = <integer>;
        };

    FLOATINGPOINT = <TRUE / FALSE>;

    };
    ACCESSING_APPLICATION = <name of Application>;
};
```

12.4.2.1 Attributes

The TASK object has the following attributes:

- ***PRIORITY*** (UINT32)

  [OsTask/OsTaskPriority]

  The attribute specifies priority of the task. The value of this attribute is to be understood as a relative value; this means that values of the PRIORITY attribute show only the relative ordering of the tasks. AUTOSAR defines the lowest priority as zero (0), the bigger value of the PRIORITY attribute corresponds to the higher priority. The value range is from 0 to 0xFFFFFFFF.

- ***SCHEDULE*** (ENUM)

  [OsTask/OsTaskSchedule]

The attribute specifies the run-time behavior of the task. If the task may be preempted by another one at any point of execution - this task attribute shall have the *FULL* value (preemptable task). If the task can be preempted only at specific points of execution (explicit rescheduling points) the attribute shall have the *NON* value (non-preemptable task).

- ***AUTOSTART*** (BOOLEAN)

  [OsTask/OsTaskAutostart]

  The attribute determines whether the task is activated during the system start-up procedure or not.

- ***APPMODE*** (reference)

  [OsTask/OsTaskAutostart/TRUE/OsTaskAppModeRef]

  The attribute defines the application mode in which the task is autostarted. The attribute can be defined if the *AUTOSTART* attribute is set to *TRUE*. It may be omitted if only one *APPMODE* is defined in the *OS*.

- ***ACTIVATION*** (UINT32)

  [OsTask/OsTaskActivation]

  The attribute defines the maximum number of queued activation requests for the task. NXP AUTOSAR OS does not support multiple activation, so this value is restricted to 1.

- ***STACKSIZE*** (UINT32)

  [OsTask/OsStackSize]

  This NXP AUTOSAR OS specific attribute defines the task stack size in bytes. It is applicable for extended tasks only. The *STACKSIZE* can not be set less than 64 bytes. The valid range is .

- ***RESOURCE*** (reference)

  [OsTask/OsTaskResourceRef]

  This reference is used to define a resource accessed by the task. If the task accesses a resource at run-time this resource shall be pointed. The resource Ceiling priority is calculated as the highest priority of tasks or ISRs accessing this resource. There can be several *RESOURCE* references. This parameter can be omitted.

- ***EVENT*** (reference)

  [OsTask/OsTaskEventRef]

This reference is used to define an event the extended task may react on. The task is considered as extended, if any event is reverenced. Otherwise the task considered as basic.

There can be several *EVENT* references. These events can be cleared and waited for by the task. All task events shall be pointed to define the event mask in case of auto-assignment (see section Event Definition).

- ***TIMING_PROTECTION***(BOOLEAN)

  [OsTask/OsTaskTimingProtection]

- ***EXECUTIONBUDGET***(UINT64)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskExecutionBudget]

- ***TIMEFRAME***(UINT64)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskTimeFrame]

- ***LOCKINGTIME***(ENUM)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskResourceLock]

- ***RESOURCE*** (reference)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskResourceLock/ OsTaskResourceLockResourceRef]

- ***MAXRESOURCELOCKTIME***(UINT64)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskResourceLock/ OsTaskResourceLockBudget]

- ***MAXOSINTERRUPTLOCKTIME***(UINT64)

  [OsTask/OsTaskTimingProtection/TRUE/OsIsrOsInterruptLockBudget]

- ***MAXALLINTERRUPTLOCKTIME***(UINT64)

  [OsTask/OsTaskTimingProtection/TRUE/OsTaskAllInterruptLockBudget]

- ***FLOATINGPOINT*** (BOOLEAN)

  [OsTask/OsTaskUseFloatingPoint]

  This attribute specifies whether the Floating-point unit in task should be supported or not.

- ***ACCESSING_APPLICATION*** (ENUM)

[OsTask/OsTaskAccessingApplication]

12.4.3 ISR Definition

This object presents an Interrupt Service Routine. The keyword ISR is used for this object type. The syntax of the ISR object is as follows:

```
ISR <name of ISR> {
    CATEGORY = <1 / 2>;
    PRIORITY = <integer>;
    IsrFunction = <string / AUTO>;
    STACKSIZE = <integer> / AUTO;
    RESOURCE = <name of RESOURCE>;
    TIMING_PROTECTION = < TRUE / FALSE >{
        EXECUTIONTIME = <integer>;
        TIMEFRAME = <integer>;
        MAXOSINTERRUPTLOCKTIME = <integer>;
        MAXALLINTERRUPTLOCKTIME = 0;
        LOCKINGTIME = RESOURCELOCK {
            RESOURCE = <name of RESOURCE>{
                MAXRESOURCELOCKTIME = <integer>;
            };
        };
    };
};
```

The same ISR name shall be used for corresponding ISR object declaration and definition (see Conventions).

12.4.3.1 Attributes

This object has the following attributes:

- ***CATEGORY*** (UINT32)

  [OsIsr/OsIsrCategory]

  The attribute specifies category of the Interrupt Service Routine. (see ISR Categories for Interrupt Service Routine Categories details).

- ***PRIORITY*** (UINT32)

  [OsIsr/OsPriority]

  The attribute specifies the priority of the interrupt request for this ISR [1..14]. NXP AUTOSAR OS initializes interrupt priority level registers for all configured external ISRs with defined *PRIORITY* values.

  PRIORITY of ISRs of category 2 shall be less than PRIORITY of any ISR of category 1.

- ***IsrFunction***(STRING)

  [OsIsr/OsIsrFunction]

  This NXP AUTOSAR OS specific attribute specifies the name of ISR handler function. By default it is the name of ISR object itself. It may be used to assign one handler to several ISR objects. If the User defines this attribute its value shall exactly corresponds to the function name. This function shall be defined as *void <FuncName>(void)*.

- ***IrqChannel*** (ENUM)

  [OsIsr/OsIrqChannel]

  The attribute specifies the interrupt channel. The only available value of this attribute is EXTERNAL.

- ***IrqNumber*** (UINT32)

  [OsIsr/OsIrqChannel/EXTERNAL/OsIrqNumber]

  The attribute specifies the interrupt source for the EXTERNAL *IrqChannel*. The available values of this attribute are 4..162.

  (For the correspondence between IrqNumber value and external interrupt sources please see the appropriate Hardware Technical References)

- ***RESOURCE*** (reference)

  [OsIsr/OsIsrResourceRef]

  The reference specifies resource accessed by the ISR. There can be several *RESOURCE* references. This parameter can be omitted. The reference can not be defined if *CATEGORY* value is equal to 1.

12.4.4 Schedule Table Definition

This group of attributes controls task features. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
SCHEDULETABLE <name of Schedule Table> {
    COUNTER = <name of Counter>;
    AUTOSTART = < TRUE / FALSE > {
        TYPE = <ABSOLUTE / RELATIVE / SYNCHRON>{
        };
        STARTVALUE = <integer / AUTO>;
        APPMODE = <name of APPMODE>;
    };
```

```
    SYNC = <TRUE / FALSE>{
        SYNCSTRATEGY = <EXPLICIT / IMPLICIT>{
            EXPLICITPRECISION = <integer>;
        };
    };
    REPEATING = <TRUE / FALSE>;
    DURATION = <integer>;
    EXPIRYPOINTS = EXPIRYPOINT {
        OFFSET = <integer>;
        ACTION = <TASKACTIVATION / EVENTSETTING>{
            TASK = <name of TASK>;
            EVENT = <name of EVENT>;
        };
        ADJUSTABLEEXPPOINT = <TRUE / FALSE>{
            MAXADVANCE = <integer>;
            MAXRETARD = <integer>;
        };
    };
    ACCESSING_APPLICATION = <name of Appl>;
};
```

12.4.4.1 Attributes

This object has the following attributes:

- ***COUNTER*** (reference)

  [OsScheduleTable/OsScheduleTableCounterRef]

  The reference specifies the assigned counter. A Schedule Table shall be assigned to a particular counter to have an ability to operate.

- ***AUTOSTART*** (BOOLEAN)

  [OsScheduleTable/OsScheduleTableAutoStart]

  The attribute defines whether the Schedule Table is started automatically at system start-up depending on the application mode.

- ***TYPE*** (ENUM)

  [OsScheduleTable/OsScheduleTableAutoStart/OsScheduleTableAutostartType]

  The attribute defines whether the Schedule Table is automatically started at system start-up synchronously or with given offset or absolute value of the assigned counter

- ***STARTVALUE*** (UINT32) (inside *AUTOSTART*)

  [OsScheduleTable/OsScheduleTableAutoStart/OsScheduleTableStartValue]

  This attribute defines the value of assigned counter at which the Schedule Table will start.

- ***APPMODE*** (reference)

[OsScheduleTable/OsAutoStart/OsAppMode]

The attribute defines the application mode for which the ScheduleTable shall be started automatically at system start-up. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*. It may be omitted if only one *APPMODE* is defined in the OS.

- ***EXPLICITPRECISION*** (UINT32)

  [OsScheduleTable/OsScheduleTblTimeSync/TRUE/ OsScheduleTblExplicitPrecision]

  The value of this attribute defines the threshold between synchronous and asynchronous state of ScheduleTable in ticks.

- ***OFFSET*** (UINT64) (inside)

  [OsScheduleTable/]

  This attribute defines the offset from Schedule Table start for given *EXPIRYPOINT*.

- ***ACTION*** (UINT64)

  [OsScheduleTable/OsScheduleTableTaskActivation or ../ OsScheduleTableEventSetting]

  The attribute defines type of task notification used when the expiry point reached. If the *ACTION* attribute is defined as *TASKACTIVATION*, the *TASK* reference defines the task to be activated when the point expires. If the *ACTION* attribute is defined as *EVENTSETTING*, then the TASK reference defines the task to be awaken, and *EVENT* reference defines the event to be set when the point expires.

- ***TASK*** (reference) (inside *ACTION*)

  [OsScheduleTable/OsScheduleTableTaskActivation/ OsScheduleTableActivateTaskRef]

  The reference to a task which is to be notified via activation or event setting at the expiration point.

- ***EVENT*** (reference) (inside *ACTION*)

  [OsScheduleTable/OsScheduleTableAction/OsScheduleTableSetEvent/ OsScheduleTableSetEventRef]

  The reference specifies the event mask to be set at the expiration point. The event is considered as an inseparable pair of the task and the event belonging to this task, so the reference to the task which owns the events shall be also defined for this *ACTION*.

The reference shall be defined if the *ACTION* value is *EVENTSETTING*.

- ***ADJUSTABLEEXPPOINT*** (BOOLEAN)

  [OsScheduleTable/OsScheduleTableExpiryPoint/ OsScheduleTblAdjustableExpPoint/OsScheduleTableMaxAdvance]

  This attribute defines is the offset value of the EXPIRYPOINT is adjustable while synchronizing.

- ***MAXADVANCE*** (UINT32)

  [OsScheduleTable/OsScheduleTableExpiryPoint/ OsScheduleTblAdjustableExpPoint/OsScheduleTableMaxLengthen]

  The attribute defines the maximum allowed value of offset increase while the schedule table is synchronized in the *EXPIRYPOINT*.

- ***MAXRETARD*** (UINT32)

  [OsScheduleTable/OsScheduleTableExpiryPoint/ OsScheduleTblAdjustableExpPoint/OsScheduleTableMaxShorten]

  The attribute defines the maximum allowed value of offset decrease while the schedule table is synchronized in the *EXPIRYPOINT*.

- ***ACCESSING_APPLICATION*** (ENUM)

  [OsScheduleTable/OsScheduleTableAccessingApp]

  This attribute defines which application(s) may access this object via OS services.

12.4.5 Resource Definition

This object is intended for the resource management. The resource *Ceiling priority* is calculated automatically on the basis of information about priorities of tasks using the resource. The keyword RESOURCE is used for this object type. Section Resource Management describes resource concept in AUTOSAR. The syntax of the resource object is as follows:

```
RESOURCE <name of resource> {
    RESOURCEPROPERTY = <STANDARD / LINKED / INTERNAL> {
        LINKEDRESOURCE = <name of RESOURCE>
    };
    ACCESSING_APPLICATION = <name of Application>;
};
```

- ***RESOURCEPROPERTY*** (ENUM)

[OsResource/OsResourceProperty]

The attribute specifies a property of the resource. The *STANDARD* value corresponds to a normal resource which is not linked to another resource and is not an internal resource. The *LINKED* value corresponds to a resource linked to another resource with the property *STANDARD* or *LINKED*. The *INTERNAL* value is appropriate to an internal resource which cannot be accessed by an application.

Performance decreases if the RESOURCE object with the INTERNAL value of the RESOURCEPROPERTY attribute is defined.

- ***LINKEDRESOURCE*** (reference)

  [OsResource/OsResourceProperty/LINKED/OsResourceLinkedResource]

  The attribute specifies the resource to which the linking shall be performed. The OS System Generator resolves chains of linked resources. This reference should be defined only if the value of the *RESOURCEPROPERTY* attribute is *LINKED*.

- ***ACCESSING_APPLICATION*** (ENUM)

  [OsResource/OsResourceAccessingApplication]

  This attribute defines which application(s) may access this object via OS services. At least one ACCESSING_APPLICATION shall be defined.

12.4.6 Event Definition

This object is intended for the event management. The event object has no references. The keyword EVENT is used for this object type. Section Events describes events in AUTOSAR. The syntax of the event object is as follows:

```
EVENT <name of EVENT> {
    MASK = <integer / AUTO>;
};
```

12.4.6.1 Attributes

The object has one standard attribute:

- ***MASK*** (UINT64)

  [OsEvent/OsEventMask]

The event is represented by its mask. The event mask is the number which range is from 1 to 0xFFFFFFFF, preferably with only one bit set. The other way to assign event mask is to declare it as *AUTO*. In this case event masks will be assigned automatically according to their distribution among the tasks.

12.4.7 Counter Definition

This object presents AUTOSAR Operating system counters. Attributes of this object type define counter properties. A counter has no references, it is referenced to by another object. The keyword COUNTER is used for this object type. AUTOSAR OS counters are described in section Counters, Alarms, Schedule Tables. The syntax of the counter object is:

```
COUNTER <name of COUNTER> {
    MINCYCLE = <integer>;
    MAXALLOWEDVALUE = <integer>;
    TICKSPERBASE = <integer>;
    SECONDSPERTICK = <integer / AUTO>;
    TYPE = <SOFTWARE / HARDWARE>{
        DRIVER = OSINTERNAL/GPT {
            NS_PER_HW_TICK = <integer>;
            GPTCHANNELNAME = "name";
        };
        TIMECONSTANTS = TIMECONSTANT{
            NS = <integer>;
            CONSTNAME = 'name';
        };
    };
    ACCESSING_APPLICATION = <name of Appl>;
};
```

12.4.7.1 Attributes

The object has the following standard attributes:

- ***MINCYCLE*** (UINT32)

  [OsCounter/OsCounterMinCycle]

  The attribute specifies the minimum allowed number of counter ticks for a cyclic alarm linked to the counter. This parameter is always defined in ticks. The parameter has a sense only for systems with extended OS status since it is checked in this case only. The value of the attribute is to be set in the range [1..0xFFFFFFFF].

- ***MAXALLOWEDVALUE*** (UINT32)

  [OsCounter/OsCounterMaxAllowedValue]

The attribute defines the maximum allowed counter value. After the counter reaches this value it rolls over and starts count again from zero. This parameter is always defined in ticks. The value of the attribute is to be set in the range [1..0xFFFFFFFF].

- ***TICKSPERBASE*** (UINT32)

  [OsCounter/OsCounterTicksPerBase]

  This is the user constant for representation of the number of ticks that are required to reach some counter-specific value. This value cannot be derived automatically from other counter related attributes. The interpretation is up to the user; it is not used by OS itself, OS just keeps this value. The value of the attribute is to be set in the range [1..0xFFFFFFFF].

- ***SECONDSPERTICK*** (UINT32)

  [OsCounter/OsCounterSecondsPerTick]

  This is the user constant for representation of the length of COUNTER tick. If it is defined for COUNTERs referred from Sys/ SecondTimer, then it has the same meaning as the timers Period. If both Period and SECONDSPERTICK are defined they shall have the same value.

- ***TYPE*** (ENUM)

  [OsCounter/OsCounterType]

  specifies the counter type, for counters assigned to System/Second Timers the type shall match the type of the Timer.

- ***NS_PER_HW_TICK*** (UINT64)

  [OsCounter/OsCounterType/HARDWARE/OsCounterDriver/GPT/ NS_PER_HW_TICK]

  Defines the value in ticks; GPT is not supported.

- ***GPTCHANNELNAME*** (STRING)

  [OsCounter/OsCounterType/HARDWARE/OsCounterDriver/GPT/ GPTCHANNELNAME]

  Defines the name of GPT Channel; GPT is not supported

- ***DRIVER*** (ENUM)

  [OsCounter/OsCounterType/HARDWARE/OsCounterDriver]

  - shall be OSINTERNAL, GPT is not supported.

- ***TIMECONSTANTS***

  [OsCounter/OsCounterType/HARDWARE/OsCounterTimeConstants]

  - allows definition of the User constants, there may be several definitions or no one. The constants are defined in timer ticks

- ***NS*** (UINT64)

  [OsCounter/OsCounterType/HARDWARE/OsCounterTimeConstants/ OsCounterTimeConstant/NS]

  - defines the time value for constant in nanoseconds

- ***CONSTNAME*** (STRING)

  [OsCounter/OsCounterType/HARDWARE/OsCounterTimeConstants/ OsCounterTimeConstant/CONSTNAME]

  - specifies the name of generated constant. The value of generated constant in ticks corresponds to NS value.

- ***ACCESSING_APPLICATION*** (ENUM)

  [OsCounter/OsCounterAccessingApplication]

  This attribute defines which application(s) may access this object via OS services.

12.4.8 Alarm Definition

This object presents alarms. Links with other system objects are defined via references. The referenced counter and task must be already defined. The keyword ALARM is used for this object type. See section Alarms for information about alarms.

The syntax of an alarm object is as follows:

```
ALARM <name of ALARM> {
    COUNTER = <name of COUNTER>;
    ACTION = <SETEVENT / ACTIVATETASK / ALARMCALLBACK / INCREMENTCOUNTER> {
        TASK = <name of TASK>;
        EVENT = <name of EVENT>;
        ALARMCALLBACKNAME = <string>;
        COUNTER = <name of COUNTER>;
    };
    AUTOSTART = <TRUE / FALSE> {
        ALARMTIME = <integer>;
        CYCLETIME = <integer>;
        APPMODE = <name of APPMODE>;
    };
    ACCESSING_APPLICATION = <name of Appl>;
};
```

12.4.8.1 Attributes

The object has the following attributes:

- ***COUNTER*** (reference)

  [OsAlarm/OsAlarmCounterRef]

  The reference specifies the assigned counter. An alarm shall be assigned to a particular counter to have an ability to operate. Only one counter has to be assigned to the alarm.

- ***ACTION*** (ENUM)

  [OsAlarm/OsAlarmAction]

  The attribute defines which type of task notification is used when the alarm expires. For one alarm only one action is allowed. If the *ACTION* attribute is defined as *ACTIVATETASK*, the *TASK* reference defines the task to be activated when the alarm expires. If the *ACTION* attribute is defined as *SETEVENT*, then the *TASK* reference defines the task to be activated, and *EVENT* reference defines the event to be set when the alarm expires. If the *ACTION* attribute is defined as *ALARMCALLBACK*, then the *ALARMCALLBACKNAME* subattribute specifies the name of the callback routine called when the alarm expires. If the *ACTION* attribute is defined as *INCREMENTCOUNTER*, the *COUNTER* reference defines the counter to be incremented when the alarm expires.

- ***ALARMCALLBACKNAME*** (STRING)

  [OsAlarm/OsAlarmAction/OsAlarmCallback/OsAlarmCallbackName]

  The attribute specifies the name of the callback routine called when the alarm expires. The parameter should be specified if the *ACTION* attribute is set as *ALARMCALLBACK*.

- ***COUNTER*** (reference) (inside *ACTION*)

  [OsAlarm/OsAlarmAction/OsAlarmIncrementCounter/ OsAlarmIncrementCounterRef]

  The attribute specifies the counter to be incremented as ALARM action. It shall be different from the *COUNTER* to which this *ALARM* is assigned.

- ***TASK*** (reference)

  [OsAlarm/OsAlarmAction/OsAlarmActivateTask/OsAlarmActivateTaskRef]

The reference to a task which is to be notified via activation or event setting when the alarm expires.

- ***EVENT*** (reference)

  [OsAlarm/OsAlarmAction/OsAlarmSetEvent/OsAlarmSetEventRef]

  The reference specifies the event mask to be set when the alarm expires. The event is considered as an inseparable pair of the task and the event belonging to this task, so the reference to the task which owns the events shall be also defined for this alarm. The reference shall be defined if the *ACTION* value is *SETEVENT*.

- ***AUTOSTART*** (BOOLEAN)

  [OsAlarm/OsAlarmAutostart]

  The attribute defines whether an alarm is started automatically at system start-up depending on the application mode. If the alarm should be started at the system start-up, the value is set to TRUE otherwise the value is set to FALSE. When the *AUTOSTART* attribute set to *TRUE*, the *ALARMTIME*, *CYCLETIME*, and *APPMODE* parameters should be defined.

- ***ALARMTIME*** (UINT32)

  [OsAlarm/OsAlarmAutostart/TRUE/OsAlarmAlarmTime]

  The attribute defines the time (in ticks) when the alarm shall expire first. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*.

- ***CYCLETIME*** (UINT32)

  [OsAlarm/OsAlarmAutostart/TRUE/OsAlarmCycleTime]

  The attribute defines the cycle time of a cyclic alarm in ticks. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*.

- ***APPMODE*** (reference)

  [OsAlarm/OsAlarmAutostart/TRUE/OsAlarmAppModeRef]

  The attribute defines the application mode for which the alarm shall be started automatically at system start-up. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*. It may be omitted if only one *APPMODE* is defined in the *OS*.

- ***ACCESSING_APPLICATION*** (ENUM)

  [OsAlarm/OsAlarmAccessingApplication]

This attribute defines which application(s) may access this object via OS services.

12.4.9 Inter-OS-application Communicator (IOC) Definition

This object is intended to present either a "Queued" (event semantic) or "LastIsBest" (data semantic) communication type. Attributes of this object type define communication properties. Links with other system objects are defined via references. The keyword IOC is used for this object type. IOC concept is described in section Communication. The syntax of a IOC object definition is as follows:

```
IOC <name of IOC> {
    DATA_PROPERTIES = DATA_PROPERTY{
        DATA_PROPERTY_INDEX = <integer / AUTO>
        DataTypeName = <string>;
        INIT_VALUE = <string>;
        DataTypeProperty = <REFERENCE / DATA>;
    };
    BUFFER_LENGTH = <integer>;
    RECEIVER = RCV {
        FUNCTION_IMPLEMENTATION_KIND=<DO_NOT_CARE / FUNCTION /MACRO>;
        RCV_OSAPPLICATION = <name of APPLICATION>;
        RECEIVER_PULL_CB = <string> / AUTO;
        ACTION = <NONE / ACTIVATETASK / SETEVENT> {
            TASK = <name of TASK>;
            EVENT = <name of EVENT>;
        };
    };
    SENDER = SND {
        FUNCTION_IMPLEMENTATION_KIND=<DO_NOT_CARE / FUNCTION /MACRO>;
        SENDER_ID = <integer> / AUTO;
        SND_OSAPPLICATION = <name of APPLICATION>;
    };
};
```

12.4.9.1 Attributes

The object has the following standard attributes:

- ***DATA_PROPERTIES = DATA_PROPERTY***

  [OsIoc/OsIocCommunication/OsIocDataProperties]

  Container(s) for data properties.

- ***DataTypeName*** (STRING)

  [OsIoc/OsIocCommunication/OsIocDataProperties/OsDataTypeName]

  This NXP-specific attribute specifies the type of the data to be transferred on the IOC communication channel. This attribute is necessary to generate the parameter type of the Ioc functions. It shall be any C data type. Any ANSI-C type specifier is allowed.

It is the standard C type identifier - *char, int, float, double* with any type modifiers (*signed, unsigned, short, long*) and also structure or union specifier (starting *struct* or *union*), enum specifier (starting *enum*), *typedef* name (any valid C-language identifier) enclosed in the double quotas. To use an array of standard C-language type the user must define the new type via *typedef* operator. In case of user's defined data types or enumerations such definitions must be in the user's code before using files produced by SG.

NOTE

[OsIoc/OsIocCommunication/OsIocDataTypeRef] - this standard attribute is not used by the OS because it is not defined how OS SysGen shall find referenced value. The attribute is ignored.

- ***DataTypeProperty*** <REFERENCE / DATA>

  [OsIoc/OsIocCommunication/OsIocDataProperties/OsDataTypeProperty]

  This NXP-specific attribute defines how the data is passed to sending functions (IOCSend, IOCWrite) - by reference or by value. Note that in existing implementation usage of REFERENCE is preferred - it has better performance

- ***DATA_PROPERTY_INDEX*** <UINT32>

  [OsIoc/OsIocCommunication/OsIocDataProperties/OsIocDataPropertyIndex]

  This parameter is used to define in which order the data is send, e.g. whether IocSendGroup(A,B) or IocSendGroup(B,A) shall be used. The valid range is [0.. 0xFF].

- ***INIT_VALUE***(ENUM)

  [OsIoc/OsIocCommunication/OsIocDataProperties/OsIocInitValue]

  The attribute defines Initial Value for the data to be transferred on the IOC communication channel.

- ***BUFFER_LENGTH***(UINT32)

  [OsIoc/OsIocCommunication/OsIocBufferLength]

  This attribute defines the size of the IOC internal queue to be allocated for a queued communication. If it is set to 0 or undefined, the IOC is not QUEUED.

- ***SENDER***(ENUM)

  [OsIoc/OsIocCommunication/OsIocSenderProperties]

This attribute defines sender properties for communication. For each communication one (1:1) or many senders (N:1) have to be defined. This parameter shall be set only to *SND* value.

- ***FUNCTION_IMPLEMENTATION_KIND***(ENUM)

  [OsIoc/OsIocCommunication/OsIocSenderProperties/ OsIocFunctionImplementationKind]

  This attribute defines whether this communication is implemented as a macro or as a function. IOC is implemented as '*MACRO*'. The rest implementation kind values are ignored.

- ***SENDER_ID***(UINT32)

  [OsIoc/OsIocCommunication/OsIocSenderProperties/OsIocSenderId]

  This attribute defines a sender in a N:1 communication to distinguish between senders. This parameter can be omited in 1:1 communication.

- ***SND_OSAPPLICATION***(reference)

  [OsIoc/OsIocCommunication/OsIocSenderProperties/ OsIocSendingOsApplicationRef]

  The reference to a sending OS-Application.

- ***RECEIVER***(ENUM)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties]

  This attribute defines receiver properties for communication. For each communication (1:1 or N:1) one receiver has to be defined. This parameter shall be set only to *RCV* value.

- ***FUNCTION_IMPLEMENTATION_KIND***(ENUM)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/ OsIocFunctionImplementationKind]

  This attribute defines whether this communication is implemented as a macro or as a function. IOC is implemented as '*MACRO*'. The rest implementation kind values are ignored.

- ***RECEIVER_PULL_CB***(STRING)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/OsIocReceiverPullCB]

This attribute defines the name of a callback function that the IOC shall call on the receiving core for each data reception. In case of non existence of this attribute no ReceiverPullCB notification shall be applied by the IOC. The attribute can not be set if *ACTION* attribute is set to *ACTIVATETASK* or *SETEVENT*.

- ***RCV_OSAPPLICATION***(reference)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/ OsIocReceivingOsApplicationRef]

  This attribute is a reference to the receiving OsApplication.

- ***ACTION***(ENUM)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/OsIocAction]

  The attribute defines which type of task notification is used when the message arrives. For one IOC only one action is allowed. If the ACTION attribute is defined as ACTIVATETASK, the TASK reference defines the task to be activated when the message arrives. If the ACTION attribute is defined as SETEVENT, then the TASK reference defines the task to be activated, and EVENT reference defines the event to be set when the message arrives. The attribute can not be set if *RECEIVER_PULL_CB* attribute is defined.

- ***TASK*** (reference)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/OsIocAction/ [OsIocActionActivateTask | OsIocActionSetEvent]/OsIocActionTask]

  The reference to a task which is to be notified via activation or event setting when the message arrives. The reference shall be defined if the *ACTION* value is *SETEVENT* or ACTIVATETASK.

- ***EVENT*** (reference)

  [OsIoc/OsIocCommunication/OsIocReceiverProperties/OsIocAction/ OsIocActionSetEvent/OsIocActionEvent]

  The reference specifies the event mask to be set when the message arrives. The event is considered as an inseparable pair of the task and the event belonging to this task, so the reference to the task which owns the events shall be also defined for this alarm. The reference shall be defined if the *ACTION* value is *SETEVENT*.

12.4.10 Application Modes Definition

It is possible to introduce different application modes inside one CPU container by means of objects named *APPMODE*. Each *APPMODE* object defines a set of autostarted Tasks and Alarms for the AUTOSAR OS application mode.

No standard attributes are defined for the APPMODE object. At least one APPMODE object has to be defined in a CPU.

The syntax of an application mode object definition is as follows:

```
APPMODE <name of mode>;
```

NXP AUTOSAR OS supports up to 8 application modes.

Chapter 13
Building of Application

13.1 Building of Application

This chapter contains information on how to build a user's application using NXP AUTOSAR OS without the AUTOSAR framework. It may be useful for the first experiments with the OS. The Sample Application may be used as an example.

This chapter consists of the following sections:

- Application Structure
- Action Sequence to Build Application

13.2 Application Structure

The application developed using the AUTOSAR Operating System basis has a defined structure. An application consists of the Operating System kernel and several user's tasks and ISRs, which interact with the kernel by means of system services and internal mechanisms. ISRs receive control from hardware interrupt sources via the vector table. Tasks are controlled by the scheduler. They may use all means for intertask communications granted by AUTOSAR OS to pass data and synchronize each other.

Tasks and ISRs are considered as system objects. Resources, IOCs, counters, and alarms are also considered as system objects, because they are controlled by the Operating System. An application typically also has configuration tables for different system objects, task stacks and other entities. To create an application, the user should develop the desired application structure with all necessary objects and define interactions between them.

All global Operating System properties, system objects and their parameters are defined by the user statically and cannot be redefined at run time. Special application configuration file is designed to perform such definition and the special tool that

processes this file. See System Configuration. After processing, files with system object descriptors are created automatically. These files provide the code for all required ROM and RAM structures, arrays, tables, variables, etc. for all system objects defined in the configuration file. Memory allocation is performed during start-up procedure.

13.3 Action Sequence to Build Application

To build an application using the AUTOSAR Operating System the user should perform a set of actions. These actions are relatively simple since the most important requirement is a clear understanding of the application algorithm. The actions include creating the application configuration file, processing this file by the System Generator, writing the user's source code, compiling all files and linking the application files together with the OS code. This process is shown on Figure 13-1.

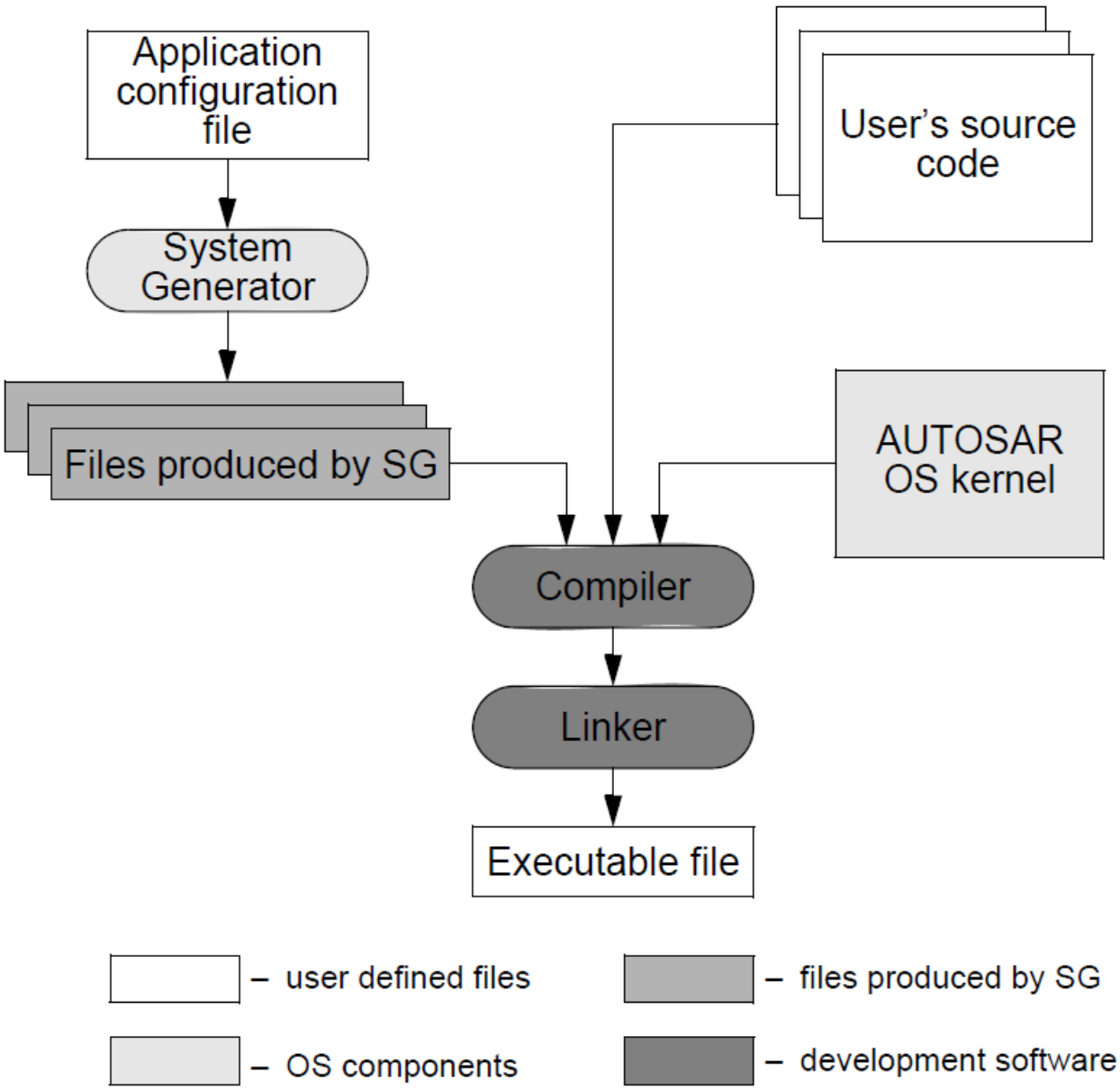

Figure 13-1. Application Building Process

13.3.1 Application Configuration

Applications built using AUTOSAR OS are configured statically via the special configuration file written in OIL. System Configuration describes the structure of such file and System Objects Definition describes all possible statements in detail. This configuration file defines system specific parameters as well as system objects. Such a file can have any extention and the extention '.oil' is suggested by default.

The configuration file has to be processed by the special utility named System Generator (SG/SysGen). This utility is delivered as one of the parts of the NXP AUTOSAR OS. This tool runs as a 32-bit console application for Windows and produces header and source files.

The following command is used to run SG:

```
sg [-options] OS_configuration_file
```

The following command line options are intended to control SG:

Table 13-1. System Generator Command Line Options

Option	Description	Default value
-c<name>	Defines <name> as the output c-data file name	'Os_cfg.c' in the source OIL file directory
-h<name>	Defines <name> as the output header file name	'Os_cfg.h' in the source OIL file directory
-i<path>	Defines <path> as the path for include files	N/A
-p<name>	Defines <name> as the OS property file name	'Os_prop.h' in the source OIL file directory
-o<name>	Defines <name> as the ORTI information file name	Input OIL file name with '.ort' extention
-t	Only verification for OIL input file. No output files shall be generated	No
-v	When this option is defined, SysGen shall output versions of all its components	No
-m<name>	Specifies the directory for generated Os_memmap.h file	N/A
-L<name>	Specifies name of input link configuration file	N/A
-M<name>	Specifies name of output link configuration file	N/A
-T	When this option is defined, SysGen shall output information about system timers	No
-C	When this option is defined, SysGen shall output the name of the license file and lists the features to be considered by Sysgen	No
-g<OIL file name>[1]	Specifies name of output OIL configuration file generated from input EPC configuration file	N/A
-e<EPC file name>[2]	Specifies name of output EPC configuration file generated from input OIL configuration file	N/A
-j <dir>	This option can be used to select a different java command than the system default; if the directory contains spaces it	The java command found in the default system path

Table 13-1. System Generator Command Line Options

Option	Description	Default value
	will be set using quotes and the slash character e.g. -j "c:/Program Files/Java/jre6/bin"	

1. Usage of <EPC file>-<OIL file> conversion see in chapter Troubleshooting
2. Usage of <OIL file>-<EPC file> conversion see in chapter Troubleshooting

The Java executable must be present in the PATH system variable.

Resource plugin is not mandatory for OS plugin generation.

In order to view all the error messages SG generates, sg.exe must be run from command line to see all errors.

In addition to command line option the OSB_INCLUDE_DIR environment variable can be used to specify the set of directories to search for include files.

The SG utility produces three types of standard C-language files which are to be compiled and linked together with OS kernel code and user's source code:

1. The header file which describes the current properties of the operating system. This file contains the preprocessor directives `#define` and `#undef`. This file is used at compile time to build the OS kernel with the specified properties. The default filename is 'Os_prop.h' but the user can assign another name (see Source Files).
2. The application configuration header file which contains definitions of all system objects, data types, constants and external declarations of variables which describe system objects. This file is used to compile application files. The default filename is 'Os_cfg.h' but the user can assign another name (see Table 13-1).
3. The source file which contains initialized data and memory allocation for system objects. This file is compiled with 'Os_prop.h' and another header files and then linked together with another application and OS files. The default filename is 'Os_cfg.c' but the user can assign another name (see Table 13-1).

To insure consistency between Os_memmap.h and linker command file the SG inserts data sections into the "template" linker command file provided by User.

If IOC objects are configured, System Generator generates an additional header file "Ioc.h" which shall be included in the User application.

NOTE

As a rule, the user is not allowed to edit files produced by the System Generator. It may lead to data inconsistency, compilation errors or unpredictable application behavior.

13.3.2 Source Files

NXP AUTOSAR OS is delivered to the user as a set of source files. Header and source files of the Operating System are located in the predefined directories. Paths to these directories have to be provided by the user.

The OS source code is compiled and linked together with other application's files. The header file 'Os_prop.h' describing system properties defines which functionality will have the OS kernel in run time. Generally, changes in OIL file result in 'Os_prop.h' modification and require recompilation of OS files. However some of object attributes do not affect 'Os_prop.h' contents, see OS Object Files Dependency for details.

This file is included in all user's and OS source files from the OS file 'Os.h'. Since the user can specify another name for property file the special macro *OSPROPH* is designed to substitute the name. The main header file 'Os.h shall be included in all C files to make OS services visible to the user application. Then OS configuration file (Os_cfg.h), which contains declarations of all OS objects is also included from 'Os.h'. Macro *OSCFGH* may be used to substitute a name of configuration header file. The following code could be used in all user's files:

```
#include 'Os.h'
```

The compiler command line (see Compiling and Linking) in this case should have the options like this:

```
-dOSPROPH="<property_name>".
-dOSCFGH="<config_name>".
```

Where <*property_name*> is the name of the file with system properties definitions and <*config_name*> is the name of the file with system configuration definitions.

Among other files SG generates configuration C-file containing definitions and initialization of OS configuration data declared in corresponding configuration header file.

Configuration C-file is a separate module, however, in some particular OS configurations, it could contain references to user defined data and structures (e.g. user's message structure types). This requires a method to provide SG generated configuration C-file with such user defined types and data declarations. Thus SG generates the following code in configuration C-file:

```
#if defined(APPTYPESH)
#include APPTYPESH /* user's header file */
#endif /* defined(APPTYPESH) */
```

The compiler command line (see Compiling and Linking) in this case should have the options like this:

```
-dAPPTYPESH="<filename>".
```

<*filename*> is the name of the file with user defined structures and data declarations.

In the example below one data type and variable are defined by the user which are referenced in files generated by SG. Variable are defined in the user's file 'user.c' and referenced in the produced file 'Os_cfg.c'. The data type is defined in the user's file 'user.h' and referenced in the produced file 'Os_cfg.c'. The user's code should be the following:

USER.H file:

```
typedef struct tagMSG MSGTYPE;
struct tagMSG
{
    TickType timeStamp;
    int x;
};
extern MSGTYPE MsgA;
```

USER.C file:

```
#include "user.h" /* include user defined data type */
...
MSGTYPE MsgA; /* user defined variables */
...
```

Os_cfg.c file (generated by SG):

```
...
#if defined(APPTYPESH)
#include APPTYPESH /* user's header file */
#endif /* defined(APPTYPESH) */
...
/* SG generated code referring to user's type and data */
...
```

The compiler command line has the following option:

```
-dAPPTYPESH="USER.H".
```

Other variants are also possible.

The code of user's tasks and functions should be developed according to common rules of the C language. But some exceptions exist:

- The keyword *TASK* and *ISR* should be used to define a task and ISR correspondingly;
- For objects controlled by the Operating System the data types defined by the system must be used. The data types are described at the end of previous sections and in System Services.

13.3.3 Compiling and Linking

When all needed header and source files are created or produced by the System Generator an application can be compiled and linked (for details see Platform-Specific Features).

Linking process is controlled by the typical linker directive file.

13.3.4 OS Object Files Dependency

The OS object files are recompiled when content of the OS property file is changed. The OS configuration depends on many parameters defined in OIL file but there exists a set of parameters which can be changed without necessity to recompile OS files (or rebuild OS library). Note that number of objects affects constants defined in OS property file, so adding or deleting an object will cause OS recompiling.

The configuration attributes which do not require recompiling of OS when their values are changed listed below.

TASK attributes:

- ***PRIORITY***
- ***AUTOSTART***
- ***RESOURCE***
- ***STACKSIZE***

ISR attributes:

- ***RESOURCE***
- ***STACKSIZE***

COUNTER attributes:

- ***MINCYCLE***
- ***MAXALLOWEDVALUE***
- ***TICKSPERBASE***

ALARM attributes:

- ***COUNTER***
- The ACTION subattributes ***TASK***, ***EVENT***, ***ALARMCALLBACKNAME***

IOC attributes:

- ***DataTypeProperty***
- ***RECEIVER_PULL_CB***
- The following ACTION subattributes:

- ***TASK***
- ***EVENT***

In Sample Application the code of an AUTOSAR OS based application is provided. This code is a simple demonstration of Operating System mechanisms. It also demonstrates how to write the configuration file and source code.

Chapter 14
ARM Architecture Platform-Specific Features

14.1 ARM Architecture Platform-Specific Features

This chapter discusses special NXP AUTOSAR OS features for different MCU types and issues connected with porting applications to these MCUs.

This chapter consists of the following sections:

- Compiler-specific Features
- S32K Specific Features

14.2 Compiler-specific Features

The versions of tools that should be used with NXP AUTOSAR OS are listed in the OS readme.txt file.

14.2.1 Used Library Functions

NXP AUTOSAR OS itself does not use any functions from compiler run-time libraries except *memcpy* and *memset*.

14.2.2 Compiler Issues

Installation procedure defines environment variable values in sample makefiles and in common.mak. If they were not set during installation the user should do it manually to compile sample. These variables are the following:

- in sample\standard\common.mak

  CYGWINDIR = [path] - path to the Cygwin 'bin' directory

GHSDIR = [path] - path to the GreenHills compiler

TRESOS_BASE = [path] - path to the Tresos Studio

- in each sample\standard\sc<n>\makefile

  SSC_ROOT = [path] - path to the AUTOSAR directory

  os_dir = [name] name of the OS directory

  See the makefiles in the SAMPLE\STANDARD directory for additional information.

  User should use the same compiler options as was used for developing and testing for compiling NXP AUTOSAR OS modules.

- **NOTE**

  It is recommended to use compiler/linker options list as it is set in the sample makefiles (except the debugging options). Using of the other compiler options may lead to possible compiling, linking or run-time problems. Therefore, in order to escape such problems and be sure that proper code is to be generated please refer to makefiles for exact option list.

  NOTE

  The compiler options used for the entire application can differ from the ones described above and should be linker compatible. The compiler options used for the OS code should not be changed because of the issue with inlining functions described below:

  If we have a file named test.c with the content

```
#include <stdio.h>

#pragma arm section code=".mysection"

static __inline int my_inline_func(int a)

{

int b;

b = a*3;

printf("test %i\n", b);

return 0;
```

```
}

#pragma arm section code


#pragma arm section code=".mysection"

int main()

{

my_inline_func(3);

my_inline_func(3);

my_inline_func(3);

return 0;

}

#pragma arm section code
```

In test.c we have a function named *my_inline_func* called 3 times from main. *my_inline_func* is declared in test.h and it should be located in a section called mysection, as specified by the pragma. Instead, the compiler places my_inline_func in the text section (please see map file), because it decides not to inline it, even if main is placed in mysection. The command lines used for compiling are:

```
armcc.exe --strict --strict-warnings --c90 --dollar -g --cpu=Cortex-M4 -Ospace --
library_type=microlib -IC:/ARM_Compiler_5/lib -I. -c test.c -o test.o

armlink --cpu=Cortex-M4 --library_type=microlib --info unused --symbols --entry main --map --
list=test.map test.o -o test
```

We decided to use -O3 -Otime instead of -Ospace.

14.2.3 Options for GreenHills MULTI

Compiler options:

```
/* select the Arm Cortex core */
-cpu=cortexm4f

/* enables several warnings which are off by default */
-Wall

/* all assembly files are preprocessed */
-preprocess_assembly_files

/* Disables support for exception handling. C++ */
```

```
--no_exceptions

/* Enables the generation of DWARF debugging information in the object file */
-dual_debug

/* Generates  errors when  functions referenced  or called have no prototype */
--prototype_errors

/* Generates warnings for undefined symbols in preprocessor expressions */
-Wundef

/* Specifies  ANSI  C  with  extensions. */
-ansi

/* Optimize for size */
-Osize

/* Generates  source  level  debugging  information and  allows   procedure  call   from
debugger's command line */
-G

/* C++ like  comments will  generate a  compilation error */
-noslashcomment

/* Issues a warning if the return type of a function is not declared before it is called */
-Wimplicit-int

/* Issues a warning if the declaration of a local variable shadows the declaration of a
variable of the same name declared at the global scope, or at an outer scope */
-Wshadow

/* Issues a warning for any use of trigraphs */
-Wtrigraphs

/* Do not use the GHS Start files */
-nostartfile

/* Allocates  uninitialized global  variables to  a section and initializes them to zero at
program startup. */
--no_commons

/* Valid #pragma directives  with wrong syntax  are treated as warnings */
--incorrect_pragma_warnings

/* Prevents the deletion of temporary files after they are used. If an assembly language
file is created by the compiler, this option will place it in the current directory instead
of the temporary directory. Produces an object file (called input-file.o) for each source
file. */
-keeptempfiles

/* Creates a listing by using the name of the object file with the .lst extension. Assembler
option */
-list

/* Produces an object file (called input-file.o) for each source file. */
-c

/* Store enumerations in the smallest possible type. */
--short_enum

/* Use hard floating point unit */
-fhard

/* Floating point format */
-fpu=vfpv4_d16
```

14.2.3.1 Assembler Options

```
/* select the Arm Cortex Core */
-cpu=cortexm4f
/* Produces an object file (called input-file.o) for each source file. */
-c

/* all assembly files are preprocessed */
-preprocess_assembly_files

/* Creates a listing by using the name of the object file with the .lst extension */
-asm=-list

/* Use hard floating point unit */
-fhard

/* Floating point format */
-mfpu=fpv4_d16
```

14.2.3.2 Linker Options

```
/* map file numeric ordering */
-Mn

/* removal from the executable of functions that are unused and unreferenced */
-delete

/* display removed unused functions */
-v

/* Ignores relocations from DWARF debug sections when using -delete. */
-ignore_debug_references

/* Creates a detailed map file */
-map

/* keep the map file in the event of a link error */
-keepmap

/* run-time environment startup routines */
-lstartup

/* run-time environment system routines */
-lsys

/* target specific runtime support */
-larch

/* standard C library */
-lansi
```

14.2.4 Options for Linaro ARM EABI

Compiler options:

```
/* Compile or assemble the source files, but do not link */
-c

/* Optimize */
-O1

/* Produce debugging information for use by GDB. */
-ggdb3

/* select the Arm Cortex core */
-cpu=cortexm4

/* Thumb instructions set. */
-mthumb

/* Generate little-endian code. */
-mlittle-endian

/* Do not keep the frame pointer in a register for functions that do not need one. */
-fomit-frame-pointer

/* Use hardware instructions for floating-point operations. */
-mhard-float

/* Specifes that the compiler should place uninitialized. */
-fno-common

/* Enable all the warnings about constructions that some users consider questionable. */
-Wall

/* Enables some extra warning flags that are not enabled by "-Wall" */
-Wextra

/* Get the other warning of "-Wextra" */
-Wno-sign-compare

/* Warn if a function is declared or defined without specifying the argument types. */
-Wstrict-prototypes

/* Hard error when a function is used before being declared. */
-Werror=implicit-function-declaration

/* Use hard floating point unit */
-mfloat-abi=hard

/* Floating point format */
-mfpu=fpv4-sp-d16
```

14.2.4.1 Assembler Options

Assembler aka compiler frontend used to do assembly preprocessing:

```
/* select the Arm Cortex Core */
-mcpu=cortex-m4
/* Compile or assemble the source files, but do not link */
-c

/* Thumb instructions set. */
-mthumb

/* Ignore this option */
-x assembler-with-cpp
```

```
/* Use hard floating point unit */
-mfloat-abi=hard

/* Floating point format */
-mfpu=fpv4-sp-d16
```

14.2.4.2 Linker Options

```
-T <linker_file>

-Map <executable>.map

-L <clib>
```

14.2.5 Options for IAR ARM EABI

Compiler options:

```
/* select the Arm Cortex core */

--cpu=Cortex-M4F

/* Selects generating code that executes in Thumb state. */
--cpu_mode=thumb

/* Specifies little-endian as the default byte order */
--endian=little

/* Sets the optimization level to High, favoring size. */
-Ohs

/* Produces an object file (called input-file.o) for each source file. */
-c

/* Makes the compiler include information in the object modules.*/
--debug

/* Disable line wrapping of diagnostic messages */
--no_wrap_diagnostics

/* Force the compiler to verify that all functions have proper prototypes */
--require_prototypes

/* Disable the automatic search for system include files */
--no_system_include

/* Floating point format */
--fpu VFPv4_sp
```

14.2.5.1 Assembler Options

Assembler aka compiler frontend used to do assembly preprocessing:

```
/* select the Arm Cortex core */

--cpu Cortex-M4F

/* Selects generating code that executes in Thumb state. */
--cpu_mode thumb

/* Disables the automatic search for system include files */
--g

/* Floating point format */
--fpu VFPv4_sp
```

14.2.5.2 Linker Options

```
/* select the Arm Cortex core */

--cpu=Cortex-M4F

/* Produces a map file. */
--map filename

/* Disable line wrapping of diagnostic messages */
--no_wrap_diagnostics

/* Suppresses dynamic initialization during system startup */
--skip_dynamic_initialization

/* Specify the configuration file to be used by the linker */
--config script.icf

/* clib */
-L
```

14.2.6 Linker Command File

Strict discipline of data placement is required to provide memory protection in AUTOSAR OS. It implies usage of System Generator to insert code and data sections, as they defined by User in the OIL file, into the linker command file. In order to support this insertion the linker file shall have special comments in predefined places. The input link file shall be specified to SysGen with switch '-L' and output link file with switch '-M'. Please note that SysGen generates the data for linker from OIL, thus insures the consistency of memory definition. The User may also use his own linker command file without SysGen pre-processing, providing that OS
sections .ostext, .osrodata, .oshook, .osbss and .vects have appropriate placement.

The user application code (Tasks and ISRs bodies) is placed into the section .appcode.

Please refer to the sample linker command files for example.

The following linker variables used by OS shall be defined in linker command file:

- The highest address of the main application stack
  - OS_MAIN_STACK_MAX
- The lowest address of the main application stack
  - OS_MAIN_STACK_MIN

14.2.7 Limitations

14.2.7.1 General Limitations

This chapter discusses NXP AUTOSAR OS specific limitations of this release.

Because one interrupt is shared between all PIT channels, System Timer and Second Timer cannot function both on PIT.

Because one interrupt is shared between all channels from a FTM module, System Timer and Second Timer cannot function both on the same FTM module.

If any of the PIT channels is allocated for System Timer or Second Timer then all the other channels cannot be used by the user, because the interrupt is used by the OS timer handler.

If any of the channels from a FTM module is allocated for System Timer or Second Timer then all the other channels cannot be used by the user, because the corresponding FTM module interrupt is used by the OS timer handler.

Freeze bit in PIT and FTM counters does not work properly. If it is set, then the counters are frozen whether the core is running or not.

14.2.7.2 S32K Specific Limitations

The following table describes timers implement in S32K.

Table 14-1. System Timer - Hardware Counter

Timer	System timer HW	Second timer HW	System timer SW	Second timer SW
SYSTICK	No	No	Yes	Yes
FTM	Yes	Yes	Yes	Yes

14.2.8 S32K Specific Features

14.2.8.1 Programming Model

NXP AUTOSAR OS/S32K is developed to work with Thumb2 with Little Endian byte ordering. Only Scalability Class 1 (SC1) is available in the current release.

14.2.8.2 Target Hardware Initialization

NXP AUTOSAR OS initializes used system timers. All other target hardware (memory used, CPU clock frequency, etc.) shall be initialized before OS startup. The NXP AUTOSAR OS/S32K package includes the files containing some initialization code located in the HWSPEC directory.

These files are the example of the HW initialization for work with MCUs supported by the OS.

14.2.8.3 Interrupt Handling and Vector Table

The interrupt vector table is defined in the file '`vector_ghs.asm`'. The file is delivered with the NXP AUTOSAR OS and located in the `ssc\hwspec\ghsarm` directory. This file contains entries for external interrupts dispatcher and for exception handlers.

The user should copy `vector_ghs.asm` file into the project directory. CAT1 and CAT2 ISRs are routed through OSInterruptDispatcher. PendSV is used internally by the OS. The placement of the vector table into the code memory region instead of SRAM memory region is done by setting the bit 29 with value 0 from VTOR register. If bit 29 is 1, the provided linker script with one section for data and one for code cannot be used. For more details, please consult Vector Table Offset Register section from the book called "Cortex™-M4 Devices; Generic User Guide, Year 2010".

14.2.8.4 Using Floating Point

The NXP AUTOSAR OS/S32K supports hard *Floating-point Unit* (FPU) based on Cortex-M4(F) Lazy Stacking and Context Switching. User can use FPU transparently, in the interrupt handler (IRQ only) or in any user task. The FPU context is saved and restored by OS on task switching and in interrupt (IRQ only). To use the FPU, user must follow steps below:

- **Step 1:** Enable the FPU in the OS's configuration file.

  Example:

  Enable the FPU to use in a task.

```
TASK <name>{
      ...
      FLOATINGPOINT = TRUE;
      ...
      };
```

  Enable the FPU to use in the interrupt handler.

```
OS <name>{
      ...
      ISRFLOATINGPOINT = TRUE;
      ...
      };
```

- **Step 2:** Enable FPU peripheral: The FPU is disabled from RESET, so user must enable the FPU peripheral manually.

  Example: Enable the FPU from Cortex-M4 devices generic user guide.

```
; CPACR is located at address 0xE000ED88
LDR.W R0, =0xE000ED88
; Read CPACR
LDR R1, [R0]
; Set bits 20-23 to enable CP10 and CP11 coprocessors
ORR R1, R1, #(0xF << 20)
; Write back the modified value to the CPACR
STR R1, [R0]; wait for store to complete
DSB
; reset pipeline now the FPU is enabled
ISB
```

- **Step 3:** Enable Lazy Stacking and Context Switching: The FPU is based on Cortex-M4(F) Lazy Stacking and Context Switching so the user needs to enable Lazy Stacking in code: set bit[30] `LSPEN = 1` and bit[31] `ASPEN = 1` (*these are default values*) in Floating-point Context Control Register (`FPCCR`).
- **Step 4:** Instruct the compiler to enable FPU:

  Options for GreenHills MULTI

  Options for Linaro ARM EABI

  Options for IAR ARM EABI

14.2.8.5 Nested Interrupts

According to the specification there are no any services which enables interrupts directly. Therefore nested interrupts with the same hardware levels can not occur. Application must not enable interrupts in ISR using direct manipulation of CPU registers - it may cause an unpredictable application behavior. Interrupts of different levels are processed in usual way and may be nested.

14.2.8.6 Interrupt Dispatcher

NXP AUTOSAR OS/S32K supports multilevel interrupts. OS Interrupt Dispatcher processes interrupts in accordance with their priorities, so interrupts of higher levels are enabled while interrupts of the same and lower levels are disabled when the ISR code is executed.

NOTE

The IOC system uses software settable interrupt 0 (NVIC id 0, CM4 vector 16). Interrupt priority level 15 is used internally by the OS and it's not available to the user. The user should not configure any interrupt with this priority level. Interrupt priority level 0 should not be used as interrupts with this interrupt level are always disabled.

14.2.9 Timer Hardware

The special OS attributes are introduced to define hardware interrupt source and desired parameters for counters assigned to the System and/or Second Timer.

Note that counter assigned to System (Second) Timer uses interrupts from corresponding timer channel. At run time NXP AUTOSAR OS enables CPU interrupts and timers interrupts corresponding to System and Second Timers. User shall not directly manipulate with System/Second timer hardware when the OS is running. However the timers hardware may be initialized prior to calling to StartOS function. In this case the *Prescaler* attribute may be set to *USER* thus disabling timer prescaler reinitialization at OS startup. If *Prescaler* is set to *OS*, its *Value* is written to the prescaler bits of the correspondent timer. Note that the *Prescaler/Value* attribute value is not equal to divide factor of timer hardware.

Timers configuration example

The system definition statements for the NXP AUTOSAR OS/S32K has the following form:

```
OS <name> {
...
    SysTimer = <SWCOUNTER / HWCOUNTER>{
        COUNTER = <CounterName>;
        ISRPRIORITY = <Priority>;
        TimerHardware = <HardwareType> {
            Prescaler = <USER / OS> {
                Value = <integer / AUTO>;
            };

            Channel = <integer>;
            PeripheralClockDivider=<integer / 1>;
            TimerModuloValue = <HardwareModulo>;
            Freeze = TRUE / FALSE;
        };
    };
...
};
```

This OS contains system timers code for supported derivatives. The following hardware sources can be used for System and/or Second Timers:

- SYSTICK (Decrementer Counter)
- FTM0, FTM1, FTM2, FTM3, FTM4, FTM5, FTM6, FTM7 (FlexTimer Module)

Only FTM can be used for hardware System or Second Timers.

The *ISRPRIORITY* attribute specifies the timer interrupt request priority. This attribute shall have the value from the range 1..14.

Chapter 15
Application Troubleshooting

15.1 Application Troubleshooting

In this chapter some advice is given which may be useful for developers working with the NXP AUTOSAR OS.

This chapter consists of the following sections:

- System Generation
- Known Problems

15.2 System Generation

The System Generator is used to generate the code for the NXP AUTOSAR OS kernel and all application objects (tasks, IOCs, etc.). This tool processed the configuration file created by the user and reports about inconsistencies and errors in it. Most of possible mistakes in application configuration process can be eliminated with the help of SG. See System Configuration and Building of Application about system generation process.

15.3 Known Problems

15.3.1 Troubleshooting

Problem A: Development software does not work under MS Windows.

- *Reason 1*: Inappropriate development hardware or software is used.
- *Reason 2*: The OS software is installed into the directory with spaces (like "`C:\Program Files`").

- *Workaround 1*: Install the software into the directory without spaces.
- *Reason 3*: Environment variables in the sample\standard\common.mak are not correct.
- *Workaround 1*:Set correct environment variables.
- *Reason 4*: Long execution time on for sg.exe.
- *Workaround 1*: It is possible that long execution times for sg.exe to occur on computers which have the environment variable LM_LICENSE_FILE set to point to servers which are unavailable, inaccessible or slow to access. This is due to the licensing library which checks for both LM_LICENSE_FILE variable and the application specific environment variable. If this problem occurs, it can be avoided if the LM_LICENSE_FILE environment variable is unset or removed entirely from the host environment. To preserve the functionality of the application that required the settings provided in the LM_LICENSE_FILE environment variable, without interfering with the sg.exe performance, the application specific environment variable shall be used instead of LM_LICENSE_FILE.

Problem B: EPC configuration file (in AUTOSAR XML format) does not work with EB tresos Studio.

- *Reason*: Any OS release is designed to work with a specific version of EB tresos Studio.

  EB tresos Studio version is specified in Release Notes.

  EPC file created with the OS release for an older version of EB tresos Studio would not work with the current version of EB tresos Studio.
- *Workaround* : EPC configuration file <*old EPC file*> shall be converted using the steps below in order to work with the new EB tresos Studio (<*new EPC file*>).

  OIL configuration file is used as an intermediate file during this conversion.

  The following steps shall be executed:

  - Run SysGen utility from the previous OS release:

    ```
    sg.exe -i . -g <OIL intermediate file> <old EPC file>
    ```

  - Adjust <OIL intermediate file> to new version of SysGen utility if necessary
  - Run SysGen utility from the new OS release:

    ```
    sg.exe -i . -e <new EPC file> <OIL intermediate file>
    ```

Chapter 16
System Services

16.1 System Services

This chapter provides a detailed description for all AUTOSAR Operating System run-time services, with appropriate examples.

This chapter consists of the following sections:

- General
- Autosar OS-Application Services
- Task Management Services
- ISR Management Services
- Resource Management Services
- Event Management Services
- Counter Management Services
- Alarm Management Services
- Scheduletable Management Services
- Communication Management Services
- Debugging Services
- Operating System Execution Control

16.2 General

This chapter provides detailed description of all AUTOSAR OS run-time services including hook routines. Also declarations of system objects - the constructional elements - are described here. The services are arranged in logical groups - for the task management, the interrupt management, etc.

Examples of code are also provided for every logical group. These examples have no practical meaning, they only show how it is possible to use OS calls in an application.

The following scheme is used for service description:

Declaration element:

Syntax:	Operating System interface in ANSI-C syntax.
Input:	List of all input parameters.
Description:	Explanation of the constructional element.
Particularities:	Explanation of restrictions relating to the utilization.
Conformance:	Specifies the Conformance Classes where the declaration element is provided.

Service description:

Syntax:	Operating System interface in ANSI-C syntax.
Input:	List of all input parameters.
Output:	List of all output parameters. Transfers via the memory use the memory reference as input parameter and the memory contents as output parameter. To clarify the description, the reference is already specified among the output parameters.
Description:	Explanation of the functionality of the operating system service.
Particularities:	Explanations of restrictions relating to the utilization of the service.
Status:	List of possible return values if service returns status of StatusType type. • Standard: List of return values provided in the operating system's standard version. Special case - service does not return status. • Extended: List of additional return values in the operating system's extended version.
Conformance:	Specifies the Conformance Classes where the service is provided.

16.3 Autosar OS-Application Services

This section describes services that are related to OS-Application handling.

16.3.1 Data Types

The AUTOSAR OS establishes the following data types for the OSApplications and memory management:

- ApplicationType - the abstract data type for OS-Application identification
- ObjectTypeType - the abstract data type for object identification
- RestartType - argument type for TerminateApplication() service

16.3.2 System Macros for Memory Access

The following macros return non-zero value if the corresponded type of access is available:

- OSMEMORY_IS_READABLE(x)
- OSMEMORY_IS_WRITEABLE(x)
- OSMEMORY_IS_EXECUTABLE(x)
- OSMEMORY_IS_STACKSPACE(x)

This macros are applicable only to the values of type AccessType.

16.3.3 Constants

- *INVALID_OSAPPLICATION* - constant of data type *ApplicationType* for undefined OS-Application
- *OBJECT_TASK* - constant of data type *ObjectTypeType* for Task object
- *OBJECT_ISR* - constant of data type *ObjectTypeType* for ISR object
- *OBJECT_ALARM* - constant of data type *ObjectTypeType* for Alarm object
- *OBJECT_RESOURCE* - constant of data type *ObjectTypeType* for Resource object
- *OBJECT_COUNTER* - constant of data type *ObjectTypeType* for Counter object
- *OBJECT_SCHEDULETABLE* - constant of data type *ObjectTypeType* for ScheduleTable object

16.3.4 States of OS-Applications

- *APPLICATION_ACCESSIBLE* (active and accessible) : OS objects may be accessed from other OS-Applications. This is the default state at startup.
- *APPLICATION_RESTART* (currently in restart phase): OS objects can not be accessed from other OS-Applications. State is valid until the OSApplication calls AllowAccess().
- *APPLICATION_TERMINATED* (terminated and not accessible): OS objects can not be accessed from other OS-Applications. State will not change.

16.3.5 GetApplicationID

Syntax:	`ApplicationType GetApplicationID (void);`
Input:	None.

Table continues on the next page...

Output:	None.
Description:	Returns the current OS-Application Id.
Particularities:	GetApplicationID returns INVALID_OSAPPLICATION, if • no OS-Application is running OR • the service is called in the wrong context OR • the service is called when interrupts are disabled by OS services.
Scalability:	SC1, SC2, SC3, SC4

16.4 CheckObjectAccess

Syntax:	`ObjectAccessType CheckObjectAccess (ApplicationType ApplID, ObjectTypeType ObjectType,OSObjectType objectId);`
Input:	<ApplID> - OS-Application Id, <ObjectType> - type of the object, <objectId> - Id of the object to be checked.
Output:	none
Description:	Checks is the object is accessible from the given OS-Application.
Status:	• ACCESS - access allowed or if the object to be examined is the RES_SCHEDULER. • NO_ACCESS - access from this OS-Application is not allowed OR if the input parameters are invalid OR the service is called in a wrong context.
Particularities:	The service always return ACCESS if the object to be examined is the RES_SCHEDULER.
Scalability:	SC1, SC2, SC3, SC4

16.5 CheckObjectOwnership

Syntax:	`ApplicationType CheckObjectOwnership (ObjectTypeType ObjectType, OSObjectType objectId);`
Input:	<ObjectType> - type of the next parameter <objectId> - reference to the object
Output:	none
Description:	If the specified object exists, CheckObjectOwnership returns the identifier of the OS-Application to which the object belongs.
Status:	• The service returns the OS-Application to which the object ObjectType belongs. • INVALID_OSAPPLICATION - if object does not exist OR the object does not belong to any OS-Application OR the ObjectType does not correcponds to <objectId> (last one in EXTENDED status).
Particularities:	If the object to be examined is the RES_SCHEDULER the service returns INVALID_OSAPPLICATION.
Scalability:	SC1, SC2, SC3, SC4

16.6 TerminateApplication

Syntax:	`StatusType TerminateApplication(ApplicationType Application, RestartType RestartOption);`
Input:	<Application> - The identifier of the OSApplication to be terminated. If the caller belongs to <Application> the call results in a self termination <RestartOption> - Either RESTART for doing a restart of the OS-Application or NO_RESTART if OSApplication shall not be restarted.
Output:	none
Description:	Terminates the calling OS-Application - kills all Tasks and ISR, frees all other OS resources associated with the application. If RestartOption equals RESTART, the configured RESTARTTASK of the terminated OS-Application is activated. TerminateApplication disables interrupt sources of all ISRs of terminating OS-Application regardless of terminating with restarting or without restarting.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - the state of <Application> is APPLICATION_TERMINATED or the state of <Application> is APPLICATION_RESTARTING and the caller does not belong to the <Application>
	• Extended:
	  • E_OS_ACCESS - if the caller belongs to a non-trusted OSApplication AND the caller does not belong to <Application>
	  • E_OS_VALUE - restart option is neither RESTART no NO_RESTART.
	  • E_OS_ID - <Application> was not valid.
	  • E_OS_CALLEVEL - called from wrong context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task level, ISR level and in the Application-specific ErrorHook The OS does not clear the interrupt request flags for killed ISRs. Tasks and interrupts that are owned by a trusted application can terminate any OS-Application. Tasks and interrupts that are owned by a non-trusted application can only terminate their owning OS-Application. Does not return if there are no error.
Scalability:	SC1, SC2, SC3, SC4

16.7 AllowAccess

Syntax:	`StatusType AllowAccess();`
Input:	none
Output:	none
Description:	This service sets the own state of an OS-Application from APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE. If the state of the OS-Application of the caller is not APPLICATION_RESTARTING the service will return E_OS_STATE. If the state of the OS-Application of the caller is APPLICATION_RESTARTING, AllowAccess() will set the state to APPLICATION_ACCESSIBLE and allow other OS-Applications to access the configured objects of the callers OS-Application.
Status:	• Standard:

Table continues on the next page...

	• E_OK - no error. • E_OS_STATE - The OS-Application of the caller is in the wrong state. • Extended: • E_OS_DISABLEDINT - call when interrupts are disabled by OS services. • E_OS_CALLEVEL - called from wrong context.
Particularities:	Allowed on Task level, ISR category 2 level
Scalability:	SC1, SC2, SC3, SC4

16.8 GetApplicationState

Syntax:	`StatusType GetApplicationState(ApplicationType Application, ApplicationStateRefType Value);`
Input:	<Application> - The OS-Application from which the state is requested
Output:	<Value> - The current state of the OS-Application
Description:	This service returns the state of OSApplication <Application> in <Value>. If the <Application> is not valid the service returns E_OS_ID.
Status:	• Standard: • E_OK - no error. • Extended: • E_OS_ID - the <Application> is not valid. • E_OS_PARAM_POINTER - <Value> is NULL pointer. • E_OS_ILLEGAL_ADDRESS - illegal <Value>. • E_OS_DISABLEDINT - call when interrupts are disabled by OS services. • E_OS_CALLEVEL - called from wrong context.
Particularities:	None
Scalability:	SC1, SC2, SC3, SC4

16.9 Task Management Services

16.9.1 Data Types

The AUTOSAR OS establishes the following data types for the task management:

- *TaskType* - the abstract data type for task identification
- *TaskRefType* - the data type to refer variables of the *TaskType* data type. Reference or pointer to *TaskType* variable can be used instead of *TaskRefType* variable
- *TaskStateType* - the data type for variables to store the state of a task
- *TaskStateRefType* - the data type to refer variables of the *TaskStateType* data type. Reference or pointer to *TaskStateType* variable can be used instead of *TaskStateRefType* variable

16.9.2 Constants

The following constants are used within the AUTOSAR Operating System to indicate task states:

- *RUNNING* - constant of data type *TaskStateType* for task state *running*
- *WAITING* - constant of data type *TaskStateType* for task state *waiting*
- *READY* - constant of data type *TaskStateType* for task state *ready*
- *SUSPENDED* - constant of data type *TaskStateType* for task state *suspended*

The following constant is used within the AUTOSAR OS to indicate task:

- *INVALID_TASK* - constant of data type *TaskType* for undefined task

16.9.3 Conventions

Within an application of the AUTOSAR OS a task should be defined according to the following pattern:

```
TASK ( <name of task> )
{
...
}
```

The name of the task function will be generated from <name of task> by macro TASK.

16.9.4 Task Declaration

The constructional statement *DeclareTask* may be used for compatibility with OSEK versions. It may be omitted in application code.

Syntax:	`DeclareTask( <name of task> );`
Input:	<name of task> - reference to the task.
Description:	This is a dummy declaration.
Particularities:	There is no need in this declaration because all system objects are defined at system generation phase.

16.9.5 ActivateTask

Syntax:	`StatusType ActivateTask( TaskType <TaskID> );`
Input:	<TaskID> - a reference to the task.
Output:	None.

Table continues on the next page...

Description:	The specified task <TaskID> is transferred from the *suspended* state into the *ready* state.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_LIMIT - too many task activations of the specified task.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service may be called both on the task level (from a task) and the interrupt level (from ISR). In the case of calling from ISR, the operating system will reschedule tasks only after the ISR completion.
Conformance:	BCC1, ECC1

16.9.6 TerminateTask

Syntax:	`StatusType TerminateTask( void );`
Input:	None.
Output:	None.
Description:	This service causes the termination of the calling task. The calling task is transferred from the *running* state into the *suspended* state.

Status:	• Standard:
	  • No return to call level.
	• Extended:
	  • E_OS_RESOURCE - the task still occupies resources
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The resources occupied by the task shall be released before the call to *TerminateTask* service. If the call was successful, *TerminateTask* does not return to the call level and enforces a rescheduling. Ending a task function without calling *TerminateTask* or *ChainTask* service is strictly forbidden. If the system with extended status is used, the service returns in case of error, and provides a status which can be evaluated in the application. The service call is allowed on task level only.
Conformance:	BCC1, ECC1

16.9.7 ChainTask

Syntax:	`StatusType ChainTask( TaskType <TaskID> );`
Input:	<TaskID> - a reference to the sequential succeeding task to be activated.
Output:	None.
Description:	This service causes the termination of the calling task. After termination of the calling task a succeeding task *<TaskID>* is transferred from the *suspended* state into the *ready* state. Using this service ensures that the succeeding task only starts to run after the calling task has been terminated.

Table continues on the next page...

Status:	• Standard:
	  • No return to call level.
	  • E_OS_LIMIT - too many task activations of the specified task.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_RESOURCE - the task still occupies resources.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The resources occupied by the calling task shall be released before the call to *ChainTask* service. If the call was successful, *ChainTask* does not return to the call level and enforces a rescheduling. Ending a task function without calling *TerminateTask* or *ChainTask* service is strictly forbidden. If the succeeding task is identical to the current task, this does not result in multiple requests. The service returns in case of error and provides a status which can be evaluated by the application. The service call is allowed on task level only.
Conformance:	BCC1, ECC1

16.9.8 Schedule

Syntax:	`StatusType Schedule( void );`
Input:	None.
Output:	None.
Description:	If there is a task in *ready* state with priority higher than assigned priority of calling task, the internal resource (if any) of the task is released, the calling task is put into the *ready* state and the higher-priority task is transferred into the *running* state. Otherwise the calling task is continued.

Status:	• Standard:
	  • E_OK - no error.
	• Extended:
	  • E_OS_RESOURCE - the task still occupies resources
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Rescheduling can only take place if an internal resource is assigned to the calling task during system generation (non-preemptable tasks are considered as a tasks with internal resource of highest priority). For these tasks, Schedule enables a processor assignment to other tasks with assigned priority not higher than the ceiling priority of the internal resource and higher than the assigned priority of the calling task. When returning from Schedule, the internal resource is taken again. This service has no influence on tasks with no internal resource assigned (preemptable tasks). The service call is allowed on task level only.
Conformance:	BCC1, ECC1

16.9.9 GetTaskID

Syntax:	`StatusType GetTaskID( TaskRefType <TaskIDRef> );`
Input:	None.

Table continues on the next page...

Output:	<*TaskIDRef*> - a pointer to the variable contained reference to the task which is currently running. The service saves the task reference into the variable, that is addressed by pointer <*TaskIDRef*>. Reference to *TaskType* variable can be used instead of *TaskRefType* variable.
Description:	This service returns reference to the task which is currently running. If there is no task in the running state, the service returns INVALID_TASK into the variable.

Status:	• Standard:
	  • E_OK - no error.
	• Extended:
	  • E_OS_PARAM_POINTER - <TaskIDRef> is NULL pointer
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.
Conformance:	BCC1, ECC1

16.9.10 GetTaskState

Syntax:	`StatusType GetTaskState( TaskType <TaskID>, TaskStateRefType <StateRef> );`
Input:	<*TaskID*> - a reference to the task.
Output:	<*StateRef*> - a pointer to the state of task. The service saves the task state into the variable, that is addressed by pointer <*StateRef*>. Reference to *TaskStateType* variable can be used instead of *TaskStateRefType* variable.
Description:	The service returns the state of the specified task <*TaskID*> *(running, ready, waiting, suspended)* at the time of calling *GetTaskState*.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient confgured access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_PARAM_POINTER - <TaskIDRef> is NULL pointer
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service may be called both on the task level (from a task) and the interrupt level (from ISR). This service may be called from *ErrorHook*, *PreTaskHook*, *PostTaskHook* hook routines. Within a full-preemptive system, calling this operating system service only provides a meaningful result if the task runs in an interrupt disabling state at the time of calling. When a call is made from a task in a full-preemptive system, the result may already be incorrect at the time of evaluation. When the service is called for a task, which is multiply activated, the state is set to *running* if any instance of the task is running.
Conformance:	BCC1, ECC1

16.10 ISR Management Services

16.10.1 Data Types

ISRType - the abstract data type for ISR identification.

16.10.2 Constants

INVALID_ISR - constant of type *ISRType* for undefined ISR.

For each defined ISR the constant with name <IsrName>PRIORITY equal to this ISR *PRIORITY* is defined.

This is an NXP OS extension of the AUTOSAR OS.

16.10.3 Conventions

Within an application an Interrupt Service Routine should be defined according to the following pattern:

```
ISR( <name of ISR>)
{
...
}
```

The keyword *ISR* is the macro for OS and compiler specific interrupt function modifier, which is used to generate valid code to enter and exit ISR.

16.10.4 ISR Declaration

The constructional statement DeclareISR[1] may be used for compatibility with OSEK versions.

Syntax:	`DeclareISR( <name of ISR> );`
Input:	<name of ISR> - reference to the ISR.
Description:	This statement declares ISR function.

1. This declaration is not defined by OSEK/VDX Operating System, v.2.2.2, 5 July 2004 specification. This is NXP OS extension of the AUTOSAR OS.

16.10.5 EnableAllInterrupts

Syntax:	`void EnableAllInterrupts ( void );`
Input:	None.
Output:	None.
Description:	This service restores the interrupts state saved by *DisableAllInterrupts* service. It can be called after *DisableAllInterrupts* only. This service is a counterpart of *DisableAllInterrupts* service, and its aim is the completion of the critical section of code. No API service calls are allowed within this critical section.

Status:	• Standard:
	  • None.
	• Extended:
	  • None.

Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. This service does not support nesting.
Conformance:	BCC1, ECC1

16.10.6 DisableAllInterrupts

Syntax:	`void DisableAllInterrupts ( void );`
Input:	None.
Output:	None.
Description:	This service saves the current interrupts state and disables all hardware interrupts. This service is intended to start a critical section of the code. This section must be finished by calling the *EnableAllInterrupts* service. No API service calls are allowed within this critical section.

Status:	• Standard:
	  • None.
	• Extended:
	  • None.

Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. This service does not support nesting. If EnableAllInterrupts service was not called after this service and before point of rescheduling then NXP AUTOSAR OS dispatcher calls ErrorHook, if it is defined, with parameter E_OS_DISABLEDINT.
Conformance:	BCC1, ECC1

16.10.7 ResumeAllInterrupts

Syntax:	`void ResumeAllInterrupts ( void );`
Input:	None.
Output:	None.
Description:	This service restores the recognition status of all interrupts saved by *SuspendAllInterrupts* service.
Status:	• Standard:

Table continues on the next page...

	• None. • Extended: • None.
Particularities:	The service may be called from an ISR category 1 and category 2, from the alarm-callbacks and from the task level, from *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. This service is the counterpart of the *SuspendAllInterrupts* service, which must have been called before, and its aim is the completion of the critical section of code. No API service calls beside *SuspendAllInterrupts*/ *ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/*ResumeOSInterrupts* pairs are allowed within this critical section. *SuspendAllInterrupts*/*ResumeAllInterrupts* can be nested. In case of nesting pairs of the calls *SuspendAllInterrupts* and *ResumeAllInterrupts* the interrupt recognition status saved by the first call of *SuspendAllInterrupts* is restored by the last call of the *ResumeAllInterrupts* service. If *STATUS* is set to EXTENDED the service call is ignored in case of a wrong sequence of interrupt management functions calls. This check is limited to 32 levels of nesting pairs of Suspend/Resume functions.
Conformance:	BCC1, ECC1

16.10.8 SuspendAllInterrupts

Syntax:	`void SuspendAllInterrupts ( void );`
Input:	None.
Output:	None.
Description:	This service saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling.
Status:	• Standard: • None. • Extended: • None.
Particularities:	The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, from *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. This service is intended to protect a critical section of code from interruptions of any kind. This section must be finished by calling the *ResumeAllInterrupts* service. No API service calls beside *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/ *ResumeOSInterrupts* pairs are allowed within this critical section. If ResumeAllInterrupts service was not called after this service and before point of rescheduling then NXP AUTOSAR OS dispatcher calls ErrorHook, if it is defined, with parameter E_OS_DISABLEDINT.
Conformance:	BCC1, ECC1

16.10.9 ResumeOSInterrupts

Syntax:	`void ResumeOSInterrupts ( void );`
Input:	None.
Output:	None.

Table continues on the next page...

Description:	This service restores the interrupts state saved by *SuspendOSInterrupts* service. It can be called after *SuspendOSInterrupts* only. This service is the counterpart of *SuspendOSInterrupts* service, and its aim is the completion of the critical section of code. No API service calls beside *SuspendAllInterrupts*/ *ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/ *ResumeOSInterrupts* pairs are allowed within this critical section.

Status:	• Standard:
	  • None.
	• Extended:
	  • None.

Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. In case of nesting pairs of the calls *SuspendOSInterrupts* and *ResumeOSInterrupts* the interrupt recognition status saved by the first call of *SuspendOSInterrupts* is restored by the last call of the *ResumeOSInterrupts* service. If *STATUS* is set to EXTENDED the service call is ignored in case of a wrong sequence of interrupt management functions calls. This check is limited to 32 levels of nesting pairs of Suspend/Resume functions. If no ISRs of category 2 are defined, then this service does nothing.
Conformance:	BCC1, ECC1

16.10.10 SuspendOSInterrupts

Syntax:	`void SuspendOSInterrupts ( void );`
Input:	None.
Output:	None.
Description:	This service saves current interrupt state and disables all interrupts of category 2. Interrupts category 1 which priority is not higher than priority of any ISR category 2 are disabled also. This service is intended to start a critical section of the code. This section must be finished by calling the *ResumeOSInterrupts* service. No API service calls beside *SuspendAllInterrupts*/*ResumeAllInterrupts* pairs and *SuspendOSInterrupts*/ *ResumeOSInterrupts* pairs are allowed within this critical section.

Status:	• Standard:
	  • None.
	• Extended:
	  • None.

Particularities:	The service may be called from an ISR and from the task level, but not from hook routines. In case of nesting pairs of the calls *SuspendOSInterrupts* and *ResumeOSInterrupts* the interrupt status saved by the first call of *SuspendOSInterrupts* is restored by the last call of the *ResumeOSInterrupts* service. If ResumeOSInterrupts service was not called after this service and before point of rescheduling then NXP AUTOSAR OS dispatcher calls ErrorHook, if it is defined, with parameter E_OS_DISABLEDINT.. If no ISRs of category 2 are defined, then this service does nothing.
Conformance:	BCC1, ECC1

16.10.11 GetISRID

Syntax:	`ISRType GetISRID (void);`
Input:	None.
Output:	None.
Description:	Returns Id of active ISR.
Particularities:	In the case of calling not from ISR returns INVALID_ISR.
Conformance:	SC2 and SC4, but in the NXP AUTOSAR OS the service is available in all classes.

16.10.12 DisableInterruptSource

Syntax:	`StatusType DisableInterruptSource (ISRType DisableISR);`
Input:	DisableISR - identifier of ISR
Output:	none
Description:	Disables the interrupt source for this ISR.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Scalability:	SC1, SC2, SC3, SC4

16.10.13 EnableInterruptSource

Syntax:	`StatusType EnableInterruptSource (ISRType EnableISR);`
Input:	<EnableISR> - ISR to be enabled by the service
Output:	none
Description:	If the arrival rate of ISR <EnableISR> is not reached, enables the source for this ISR. If the arrival rate of ISR <EnableISR> is reached then the interrupt source will be enabled at the start of the next timeframe.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Scalability:	SC1, SC2, SC3, SC4

16.11 Resource Management Services

16.11.1 Data Types

The AUTOSAR OS establishes the following data type for the resource management:

- ResourceType - the abstract data type for referencing a resource

The only data type must be used for operations with resources.

16.11.2 Constants

RES_SCHEDULER - constant of data type ResourceType corresponded to Scheduler Resource (see Scheduler as a Resource)

16.11.3 Resource Declaration

The declaration statement DeclareResource may be used for compatibility with OSEK versions. It may be omitted in application code.

Syntax:	`DeclareResource( <name of resource> );`
Input:	<name of resource> - a reference to the resource.
Description:	This is a dummy declaration.
Particularities:	There is no need in this declaration because all system objects are defined at system generation phase.

16.11.4 GetResource

Syntax:	`StatusType GetResource( ResourceType <ResID> );`
Input:	<ResID> - a reference to the resource.
Output:	None.
Description:	This service changes current priority of the calling task or ISR according to ceiling priority protocol for resource management. *GetResource* serves to enter critical section in the code and blocks execution of any task or ISR which can get the resource <*ResID*>. A critical section must always be left using *ReleaseResource* within the same task or ISR.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.

Table continues on the next page...

	• E_OS_ACCESS - attempt to get resource which is already occupied by any task or ISR, or the assigned in OIL priority of the calling task or interrupt routine is higher than the calculated ceiling priority. • E_OS_CALLEVEL - call at not allowed context. • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
Particularities:	This function is fully supported in all Conformance Classes. It is NXP OS extension of the AUTOSAR OS because OSEK/VDX specifies full support only beginning from BCC2. Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden. The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no *standard* resources are defined in the configuration file. Regarding Extended Tasks, please note that *WaitEvent* within a critical section is prohibited.
Conformance:	BCC1, ECC1

16.11.5 ReleaseResource

Syntax:	`StatusType ReleaseResource( ResourceType <ResID> );`
Input:	<ResID> - a reference to the resource.
Output:	None.
Description:	This call serves to leave the critical sections in the code that are assigned to the resources referenced by <ResID>. A *ReleaseResource* call is a counterpart of a *GetResource* service call. This service returns task or ISR priority to the level saved by corresponded *GetResource* service.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_NOFUNC - attempt to release a resource which is not occupied by any task or ISR, or another resource has to be released before.
	  • E_OS_ACCESS - attempt to release a resource which has a lower ceiling priority than the assigned in OIL priority of the calling task or interrupt routine.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	This function is fully supported in all Conformance Classes. It is NXP OS extension of the AUTOSAR OS because OSEK/VDX specifies full support only beginning from BCC2. Nested resource occupation is allowed only if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden. The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no *standard* resources are defined in the configuration file.
Conformance:	BCC1, ECC1

16.12 Event Management Services

16.12.1 Data Types

The AUTOSAR Operating System establishes the following data types for the event management:

- *EventMaskType* - the data type of the event mask
- *EventMaskRefType* - the data type of the pointer to an event mask

16.12.2 Event Declaration

DeclareEvent declaration statement may be used for compatibility with OSEK versions. It may be omitted in application code.

Syntax:	`DeclareEvent( <name of event> );`
Input:	<name of event> - event name.
Description:	This is a dummy declaration.
Particularities:	There is no need in this declaration because all system objects are defined at system generation phase.

16.12.3 SetEvent

Syntax:	`StatusType SetEvent( TaskType <TaskID>, EventMaskType <Mask> );`
Input:	<TaskID> - a reference to the task for which one or several events are to be set. <Mask> - an event mask to be set.
Output:	None.
Description:	This service is used to set one or several events of the desired task according to the event mask. If the task was *waiting* for at least one of the specified events, then it is transferred into the *ready* state. The events not specified by the mask remain unchanged. Only an extended task which is not suspended may be referenced to set an event.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_ACCESS - the referenced task is not an Extended Task.
	  • E_OS_STATE - the referenced task is in the suspended state.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	It is possible to set events for the running task (task-caller). The service call is allowed on task level and ISR level, but not in hook routines. This service is not implemented if no events are defined in the configuration file.
Conformance:	ECC1

16.12.4 ClearEvent

Syntax:	`StatusType ClearEvent( TaskType <TaskID>, EventMaskType <Mask> );`
Input:	<Mask> - an event mask to be set.
Output:	None.
Description:	The task which calls this service defines the event which has to be cleared.

Status:	• Standard:
	  • E_OK - no error.
	• Extended:
	  • E_OS_ACCESS - the referenced task is not an Extended Task.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The system service *ClearEvent* can be called from extended tasks which own an event only. This service is not implemented if no events are defined in the configuration file.
Conformance:	ECC1

16.12.5 GetEvent

Syntax:	`StatusType GetEvent( TaskType <TaskID>, EventMaskRefType <Event> );`
Input:	<TaskID> - a reference to the task whose event mask is to be returned.
Output:	<Event> - a pointer to the variable of the return state of events.
Description:	The event mask which is referenced to in the call is filled according to the state of the events of the desired task. Current state of events is returned but not the mask of events that task is waiting for. The function has been updated for multicore configurations to work for cross core calls. It is possible to get event mask of the running task (task-caller).

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient configured access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the task identifier is invalid.
	  • E_OS_PARAM_POINTER - <Event> is NULL pointer.
	  • E_OS_ACCESS - the referenced task is not an Extended Task.
	  • E_OS_STATE - the referenced task is in the suspended state.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_CORE - remote call when remote core is not active.

Particularities:	The referenced task must be an extended task and it can not be in *suspended* state. The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. This service is not implemented if no events are defined in the configuration file.
Conformance:	ECC1

16.12.6 WaitEvent

Syntax:	`StatusType WaitEvent( EventMaskType <Mask> );`
Input:	<Mask> - an event mask to wait for.
Output:	None.
Description:	The calling task is transferred into the *waiting* state until at least one of the events specified by the mask is set. The task is kept the *running* state if any of the specified events is set at the time of the service call.

Status:	• Standard:
	  • E_OK - no error.
	• Extended:
	  • E_OS_ACCESS - the referenced task is not an Extended Task.
	  • E_OS_RESOURCE - the calling task occupies resources..
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	This call enforces the rescheduling, if the wait condition occurs. All resources occupied by the task must be released before WaitEvent service call. The service can be called from extended tasks which own an event only. This service is not implemented if no events are defined in the configuration file.
Conformance:	ECC1

16.13 Counter Management Services

16.13.1 Data Types And Identifiers

The following data types are established by AUTOSAR OS to operate with counters:

- *TickType*- the data type represent count value in ticks
- *TickRefType*- the data type of a pointer to the variable of data type *TickType*
- *CounterType*- the data type references a counter
- *CtrInfoRefType* - the data type of a pointer to the structure of data type *CtrInfoType*
- *CtrInfoType*- the data type represents a structure for storage of counter characteristics. This structure has the following elements:
  - *maxallowedvalue* - maximum possible allowed counter value in ticks

  - *ticksperbase* - number of ticks required to reach a counterspecific significant unit (it is a user constant, OS does not use it)
  - *mincycle* - minimum allowed number of ticks for the cycle parameter of *SetRelAlarm* and *SetAbsAlarm* services (only for system with Extended Status)
- PhysicalTimeType - the data type is used for values returned by the conversion macro OS_TICKS2<Unit>_<Counter>(ticks) (System Macros for Time Conversion). It is defined as OSDWORD (32-bit) for the OS.

All elements of CtrInfoType structure have the data type TickType, and the structure looks like the following:

```
/* for EXTENDED status */
typedef CtrInfoType tagCIT;
struct tagCIT
{
    TickType maxallowedvalue;
    TickType ticksperbase;
    TickType mincycle;
};
/* for STANDARD status */
typedef CtrInfoType tagCIT;
struct tagCIT
{
    TickType maxallowedvalue;
    TickType ticksperbase;
};
```

NOTE

CtrInfoType and CtrInfoRefType data types are not defined in the *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. These are NXP OS extension of the AUTOSAR OS.

16.13.2 System Macros for Time Conversion

The conversation macros to convert counter ticks into real time are defined for Counters attached to System(Second)Timer or for Counters which has the OsSecondsPerTick attribute defined. The format of the macros is *OS_TICKS2<Unit>_<Counter>(ticks)* whereas <Unit> is one of *NS* (nanoseconds), *US* (microseconds), *MS* (milliseconds) or *SEC* (seconds) and <*Counter*> is the name of the counter.

16.13.3 Constants

For all counters, the following constants are defined:

- *OSMAXALLOWEDVALUE_cname*

Maximum possible allowed value of counter <cname> in ticks.

- *OSTICKSPERBASE_cname*

  Number of ticks required to reach a specific unit of counter <cname> (it is a user constant, OS does not use it).

- *OSMINCYCLE_cname*

  Minimum allowed number of ticks for a cyclic alarm of counter <cname>. This constant is not defined in STANDARD status

For system counters, which are always time counters, the special constants are provided by the operating system:

- *OSMAXALLOWEDVALUE / OSMAXALLOWEDVALUE2*

  maximum possible allowed value of the system/second timer in ticks (see also Counter Definition)

- *OSTICKSPERBASE / OSTICKSPERBASE2*

  number of ticks required to reach a counter-specific value in the system/second counter (it is a user constants, not used by OS) (see also Counter Definition)

- *OSTICKDURATION / OSTICKDURATION2*

  duration of a tick of the system/second counter in nanoseconds (defined automatically by System Generator utility (see also CPU Related Attributes)

- *OSMINCYCLE / OSMINCYCLE2*

  minimum allowed number of ticks for a cyclic alarm attached to the system/second counter (only for system with Extended Status, see also Alarm Definition)

NOTE

OSMAXALLOWEDVALUE2, *OSTICKSPERBASE2*, *OSTICKDURATION2*, and *OSMINCYCLE2* constants are not defined in the *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. These are NXP OS extension of the AUTOSAR OS.

16.13.4 Counter Declaration

DeclareCounter declaration statement may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax:	`DeclareCounter( <name of counter> );`
Input:	*<name of counter>* - reference to the counter.
Description:	This is a dummy declaration.
Particularities:	There is no need in this declaration because all system objects are defined at system generation phase.

16.13.5 InitCounter

Syntax:	`StatusType InitCounter( CounterType <CounterID>, TickType <Ticks> );`
Input:	*<CounterID>* - a reference to the counter. *<Ticks>* - a counter initialization value in ticks.
Output:	None.
Description:	Sets the initial value of the counter with the value *<Ticks>*. After this call the counter will advance this initial value by one via the following call of *IncrementCounter*. If there are running attached alarms, then their state stays unchanged, but the expiration time becomes indeterminate.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the counter identifier is invalid.
	  • E_OS_VALUE - the counter initialization value exceeds the maximum admissible value.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service call is allowed on task level only. This service is not implemented if no counters are defined in the configuration file. The *InitCounter* service is not defined in the *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. This is NXP OS extension of the AUTOSAR OS.
Conformance:	BCC1, ECC1

16.13.6 IncrementCounter

Syntax:	`StatusType IncrementCounter( CounterType <CounterID> );`
Input:	<CounterID> - a reference to the counter.
Output:	None.
Description:	The service increments the current value of the counter. If the counter value was equal to *maxallowedvalue* (see Data Types And Identifiers) it is reset to zero. If alarms are linked to the counter, the system checks whether they expired after this tick and performs appropriate actions (task activation and/or event setting).

Table continues on the next page...

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the counter identifier is invalid or belongs to hardware counter.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service call is allowed on task level and ISR level, but not in hook routines. The service shall not be used for counters assigned to System and Second timers. This service is not implemented if no counters are defined in the configuration file.
Conformance:	BCC1, ECC1

16.13.7 GetCounterValue

Syntax:	`StatusType GetCounterValue( CounterType <CounterID>, TickRefType <TicksRef> );`
Input:	<*CounterID*> - a reference to the counter.
Output:	<*TicksRef*> - a pointer to counter value in ticks. Reference to *TickType* variable can be used instead of *TickRefType* variable.
Description:	The service provides the current value of the counter <*CounterID*> in ticks and saves it in the variable referenced by <*TicksRef*>.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the counter identifier is invalid or belongs to hardware counter.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER - <TicksRef> is NULL pointer

Particularities:	The service call is allowed on task and ISR level. This service is not implemented if no counters are defined in the configuration file.
Conformance:	BCC1, ECC1

16.13.8 GetElapsedValue

Syntax:	`StatusType GetElapsedValue ( CounterType <CounterID>, TickRefType <PreviousValue>, TickRefType <Value> );`
Input:	<*CounterID*> - a reference to the counter. <*PreviousValue*> - a reference to the previously read value
Output:	<*PreviousValue*> - updated with current read value. <*Value*> - reference to the difference with previous read value in ticks.
Description:	The service returns in the variable referenced by <Value> the number of elapsed ticks since the given <PreviousValue> value and updates <Value> with new counter value

Table continues on the next page...

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the counter identifier is invalid or belongs to hardware counter.
	  • E_OS_VALUE - given PreviousValue is not valid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER - <TicksRef> is NULL pointer

Particularities:	The service call is allowed on task and ISR level. If the timer already passed the <PreviousTick> value a second time, the returned result is invalid. This service is not implemented if no counters are defined in the configuration file.
Conformance:	BCC1, ECC1

16.13.9 GetCounterInfo

Syntax:	`StatusType GetCounterInfo( CounterType <CounterID>, CtrInfoRefType <InfoRef> );`
Input:	<*CounterID*> - a reference to the counter.
Output:	<*InfoRef*> - a pointer to the structure of *CtrInfoType* data type. Reference to the *CtrInfoType* variable can be used instead of the *CtrInfoRefType* variable.
Description:	The service provides the counter characteristics into the structure referenced by <*InfoRef*>. For a system counter special constants may be used instead of this service.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured)
	• Extended:
	  • E_OS_ID - the counter identifier is invalid or belongs to hardware counter.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER - <TicksRef> is NULL pointer

Particularities:	The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. The structure referenced by <*InfoRef*> consists of two elements in case of the 'Standard Status', and of three elements in case of the 'Extended Status'. This service is not implemented if no counters are defined in the configuration file. The *GetCounterInfo* service is not defined in the *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. This is NXP OS extension of the AUTOSAR OS.
Conformance:	BCC1, ECC1

16.14 Alarm Management Services

16.14.1 Data Types and Identifiers

The following data types are established by AUTOSAR OS to operate with alarms:

- *TickType* - data type represents count values in ticks
- *TickRefType* - the data type of a pointer to the variable of data type *TickType*
- *AlarmBaseType* - the data type represents a structure for storage of counter characteristics. The elements of the structure are:
  - *maxallowedvalue* - maximum possible allowed counter value in ticks
  - *ticksperbase* - number of ticks required to reach a counterspecific significant unit
  - *min*cycle - minimum allowed number of ticks for the cycle parameter of *SetRelAlarm* and *SetAbsAlarm* services (only for system with Extended Status)

  All elements of the structure are of data type *TickType*.
- *AlarmBaseRefType*- the data type references data corresponding to the data type *AlarmBaseType*
- *AlarmType*- the data type represents an alarm object

16.14.2 Constants

OSMINCYCLE - minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status)

16.14.3 Alarm Declaration

The declaration statement *DeclareAlarm* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax:	`DeclareAlarm( <name of alarm> );`
Input:	*<name of alarm>* - reference to the alarm.
Description:	This is a dummy declaration.
Particularities:	There is no need in this declaration because all system objects are defined at system generation phase.

16.14.4 GetAlarmBase

Syntax:	`StatusType GetAlarmBase( AlarmType <AlarmID>, AlarmBaseRefType <InfoRef> );`
Input:	*<AlarmID>* - a reference to the alarm.
Output:	*<InfoRef>* - a pointer to the structure *<InfoRef>* with returned values of the alarm base. Reference to *AlarmBaseType* variable can be used instead of *AlarmBaseRefType* variable.

Table continues on the next page...

Description:	The service returns the alarm base characteristics into the structure pointed by <*InfoRef*>. The return value is a structure in which the information of data type *AlarmBaseType* is stored.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (for SC3, SC4); OR the alarm task owned by the OS-Application which has unappropriated state (not APPLICATION_ACCESSIBLE).
	• Extended:
	  • E_OS_ID - the alarm identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER - <TicksRef> is NULL pointer.

Particularities:	The structure consists of two elements in case of the 'Standard Status', and of three elements in case of the 'Extended Status'. The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. This service is not implemented if no alarms are defined in the configuration file.
Conformance:	BCC1, ECC1

16.14.5 GetAlarm

Syntax:	`StatusType GetAlarm( AlarmType <AlarmID>, TickRefType <TicksRef> );`
Input:	<*AlarmID*> - a reference to the alarm.
Output:	<*TicksRef*> - a pointer to a variable which gets a relative value in ticks before the alarm expires. Reference to *TickType* variable can be used instead of *TickRefType* variable.
Description:	This service calculates the time in ticks before the alarm expires. If the alarm is not in use, then returned value is not defined.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_NOFUNC - the alarm is not in use.
	  • E_OS_ACCESS - insufficient access rights (for SC3, SC4); OR the alarm task owned by the OS-Application which has unappropriated state (not APPLICATION_ACCESSIBLE).
	• Extended:
	  • E_OS_ID - the alarm identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER - <TicksRef> is NULL pointer.

Particularities:	It is up to the application to decide whether for example an alarm may still be useful or not. The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. This service is not implemented if no alarms are defined in the configuration file.
Conformance:	BCC1, ECC1

16.14.6 SetRelAlarm

Syntax:	`StatusType SetRelAlarm( AlarmType <AlarmID>, TickType <Increment>, TickType <Cycle> );`
Input:	<AlarmID> - a reference to the alarm; <Increment> - an alarm initialization value in ticks; <Cycle> - an alarm cycle value in ticks in case of cyclic alarm. In case of single alarms, the value cycle has to be equal zero.
Output:	None.
Description:	The system service occupies the alarm <AlarmID> element. After <Increment> counter ticks have elapsed, the task assigned to the alarm <AlarmID> is activated or the assigned event (only for Extended Tasks) is set. If <Cycle> is unequal to 0, the alarm element is logged on again immediately after expiry with the relative value <Cycle>. Otherwise, the alarm triggers only once. If relative value<Increment> equals 0, the alarm expires immediately and assigned task becomes ready before the system service returns to the calling task or ISR.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - the alarm is already in use.
	  • E_OS_VALUE - Value of <Increment> is equal to 0.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - the alarm identifier is invalid.
	  • E_OS_VALUE - an alarm initialization value is outside of the admissible limits (lower than zero or greater than the maximum allowed value of the counter), or alarm cycle value is unequal to 0 and outside of the admissible counter limits (less than the minimum cycle value of the counter or greater than the maximum allowed value of the counter).
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on task level and ISR level, but not in hook routines. If alarm is already in use, the service call is ignored. To change values of alarms already in use the alarm has to be cancelled first. This service is not implemented if no alarms are defined in the configuration file.
Conformance:	BCC1, ECC1.

16.14.7 SetAbsAlarm

Syntax:	`StatusType SetAbsAlarm( AlarmType <AlarmID>, TickType <Start>, TickType <Cycle> );`
Input:	<*AlarmID*> - a reference to the alarm; <*Start*> - an absolute value in ticks; <*Cycle*> - an alarm cycle value in ticks in case of cyclic alarm. In case of single alarms, cycle has to be equal zero.
Output:	None.
Description:	The system service occupies the alarm <AlarmID> element. When <Start> ticks are reached, the task assigned to the alarm <AlarmID> is activated or the assigned event (only for Extended Tasks) is set.

Table continues on the next page...

	If <Cycle> is unequal to 0, the alarm element is logged on again immediately after expiry with the relative value <Cycle>. Otherwise, the alarm triggers only once. If the absolute value <Start> is very close to the current counter value, the alarm may expire and assigned task may become ready before the system service returns to the calling task or ISR. If the absolute value <Start> have been reached before the service call, the alarm will only expire when <Start> value will be reached again.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - the alarm is already in use.
	  • E_OS_VALUE - Value of <Increment> is equal to 0.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - the alarm identifier is invalid.
	  • E_OS_VALUE - an alarm initialization value is outside of the admissible limits (lower than zero or greater than the maximum allowed value of the counter), or alarm cycle value is unequal to 0 and outside of the admissible counter limits (less than the minimum cycle value of the counter or greater than the maximum allowed value of the counter).
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on task level and ISR level, but not in hook routines. If alarm is already in use, the service call is ignored. To change values of alarms already in use the alarm has to be cancelled first. This service is not implemented if no alarms are defined in the configuration file.
Conformance:	BCC1, ECC1.

16.14.8 CancelAlarm

Syntax:	`StatusType CancelAlarm( AlarmType <AlarmID> );`
Input:	*<AlarmID>* - a reference to the alarm.
Output:	None.
Description:	The service cancels the alarm (transfers it into the stop state).

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_NOFUNC - the alarm is not in use.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - the alarm identifier is invalid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service is allowed on task level and in ISR, but not in hook routines. This service is not implemented if no alarms are defined in the configuration file.
Conformance:	BCC1, ECC1

16.15 Scheduletable Management Services

16.15.1 Data Types

- ScheduleTableType - identifier of ScheduleTable.
- ScheduleTableStatusType - the status of a schedule table;
- ScheduleTableStatusRefType - reference to a variable of the data type ScheduleTableStatusType;
- GlobalTimeTickType - the global time source type.

16.15.2 Constants

The following constants of ScheduleTableStatusType type are defined:

- SCHEDULETABLE_STOPPED
- SCHEDULETABLE_NEXT
- SCHEDULETABLE_WAITING
- SCHEDULETABLE_RUNNING
- SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

For meaning of this constants please refer to GetScheduleTableStatus description.

16.15.3 StartScheduleTableRel

Syntax:	`StatusType StartScheduleTableRel(ScheduleTableType ScheduleTableID, TickType Offset);`
Input:	<ScheduleTableID> - schedule table to be started. <Offset> - relative tick value between now and the first action point.
Output:	None.
Description:	If its input parameters are valid and the state of schedule table <ScheduleTableID> is SCHEDULETABLE_STOPPED, then StartScheduleTableRel() shall start the processing of a schedule table <ScheduleTableID>. The Initial Expiry Point shall be processed after <Offset> + Initial Offset ticks have elapsed on the underlying counter. The state of <ScheduleTableID> is set to SCHEDULETABLE_RUNNING before the service returns to the caller.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - Schedule table is not in state SCHEDULETABLE_STOPPED.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid or implicitly synchronized.
	  • E_OS_VALUE - <Offset> is zero or greater than (MAXALLOWEDVALUE - InitialOffset).
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR category 2 level.

16.15.4 StartScheduleTableAbs

Syntax:	`StatusType StartScheduleTableAbs(ScheduleTableType ScheduleTableID, TickType TickValue);`
Input:	<ScheduleTableID> - schedule table to be started. <TickValue> - absolute tick value of the first action point.
Output:	None.
Description:	If its input parameters are valid, the service starts the processing schedule table <ScheduleTableID> at its first expiry point after underlaying counter reaches <TickValue> and return E_OK.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - Schedule table is not in state SCHEDULETABLE_STOPPED.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid or implicitly synchronized.
	  • E_OS_VALUE - <TickValue> is greater than MAXALLOWEDVALUE.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR level.

16.15.5 StartScheduleTableSynchron

Syntax:	`StatusType StartScheduleTableSynchron(ScheduleTableType ScheduleTableID);`
Input:	<ScheduleTableID> - schedule table to be started.
Output:	None.
Description:	If its input parameters are valid, the service sets the state of <ScheduleTableID> to SCHEDULETABLE_WAITING and start the processing of schedule table <ScheduleTableID> after the synchronization count of the schedule table is set via SyncScheduleTable().

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_STATE - Schedule table is not in state SCHEDULETABLE_STOPPED.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid or implicitly synchronized.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR level. Available in all Scalability Classes - it is NXP OS extension of the AUTOSAR OS specification, which requires this service only in SC2, SC4.

16.15.6 StopScheduleTable

Syntax:	`StatusType StopScheduleTable(ScheduleTableType ScheduleTableID);`
Input:	<ScheduleTableID> - schedule table to be stopped.
Output:	None.
Description:	The service stops a *Schedule Table* processing immediately.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_NOFUNC - Schedule table is not in state SCHEDULETABLE_STOPPED.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR level.

16.15.7 NextScheduleTable

Syntax:	`StatusType NextScheduleTable(ScheduleTableType ScheduleTableIDCurrent, ScheduleTableType ScheduleTableIDNext);`
Input:	<ScheduleTableIDCurrent> - schedule table. <ScheduleTableIDNext> - schedule table that provides its series of expiry points.
Output:	None.
Description:	If the input parameters are valid and the <ScheduleTableIDCurrent> is running, the service plans the start of the schedule table <ScheduleTableIDNext> after <ScheduleTableCurrent> reaches its period and returns E_OK. If its input parameters are valid and the <ScheduleTableIDCurrent> is running and NextScheduleTable() was previously successfully called, the new <ScheduletableIDNext> replace the previous next value. The current ScheduleTable is stopped when it's period is reached.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_NOFUNC - <ScheduleTableIDCurrent> was not started.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	  • E_OS_STATE - <ScheduleTableIDCurrent> is in state SCHEDULETABLE_STOPPED or in state SCHEDULETABLE_NEXT
	• Extended:
	  • E_OS_ID - <ScheduleTableIDCurrent> or <ScheduleTableIDNext> is not valid or belongs to different counters.
	  • E_OS_STATE - <ScheduleTableIDNext> not in state OSCHEDULETABLE_STOPPED.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR level.

16.15.8 SyncScheduleTable

Syntax:	`StatusType SyncScheduleTable(ScheduleTableType ScheduleTableID, TickType <Value>);`

Table continues on the next page...

Input:	<ScheduleTableID> - the schedule table identifier. <Value> - the current value of global time.
Output:	None.
Description:	This service provides the OS with current global time. It is used to synchronize the processing of schedule table to global time.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	  • E_OS_STATE - the ScheduleTable is not started (its state is SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid or SYNCSTRATEGY is not EXPLICIT.
	  • E_OS_STATE - the ScheduleTable is not started (its state is SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT).
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	The service is applicable for ScheduleTable in waiting state (SCHEDULETABLE_WAITING). Allowed on Task and ISR category 2 level. Available in all Scalability Classes - it is NXP OS extension of the AUTOSAR OS specification, which requires this service only in SC2, SC4.
Conformance:	

16.15.9 SetScheduleTableAsync

Syntax:	`StatusType SetScheduleTableAsync(ScheduleTableType ScheduleTableID);`
Input:	<ScheduleTableID> - the schedule table identifier.
Output:	None.
Description:	This service set the synchronization status of the <ScheduleTableID> to asynchronous. If this service is called for a running schedule table the OS continue it's processing.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	  • E_OS_STATE - <ScheduleTableID> is not in one of *_RUNNING* states.
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.

Particularities:	Allowed on Task and ISR level. Available in all Scalability Classes - it is NXP OS extension of the AUTOSAR OS specification, which requires this service only in SC2, SC4.

16.15.10 GetScheduleTableStatus

Syntax:	`StatusType GetScheduleTableStatus (ScheduleTableType ScheduleID, ScheduleTableStatusRefType ScheduleStatus);`
Input:	<ScheduleTableID> - the schedule table identifier.
Output:	<ScheduleStatus> - reference to the variable of ScheduleStatusType.
Description:	If schedule table <ScheduleTableID> is not yet started, this service passes back SCHEDULETABLE_STOPPED via the reference parameter <ScheduleStatus> and return E_OK. If the schedule table <ScheduleTableID> is configured with explicit synchronization strategy and no synchronization count was provided to the OS, the service returns SCHEDULETABLE_WAITING via the reference parameter <ScheduleStatus> and returns E_OK. If the schedule table <ScheduleTableID> was used in a NextScheduleTable () call and waits for the end of the current schedule table, this service SCHEDULETABLE_NEXT via the reference parameter <ScheduleStatus> and return E_OK. If schedule table <ScheduleTableID> started and synchronous, the service passes back SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS via the reference parameter <ScheduleStatus> and returns E_OK. If schedule table <ScheduleTableID> is started, but is not synchronous (deviation is not within the precision interval or the global time is not available), the service passes back SCHEDULETABLE_RUNNUING via the reference parameter <ScheduleStatus> and returns E_OK.

Status:	• Standard:
	  • E_OK - no error.
	  • E_OS_ACCESS - insufficient access rights (if more than one OS-Application configured).
	• Extended:
	  • E_OS_ID - <ScheduleTableID> is not valid.
	  • E_OS_CALLEVEL - call at not allowed context.
	  • E_OS_DISABLEDINT - call when interrupts are disabled by OS services.
	  • E_OS_PARAM_POINTER <ScheduleStatus> is NULL pointer.

Particularities:	none

16.16 Communication Management Services

16.16.1 Data Types

The following data types are established by AUTOSAR OS to operate with IOCs:

- *Std_ReturnType* - the data type represent variable for saving IOC status.

16.16.2 Constants

The following constants of *Std_ReturnType* type are defined:

- IOC_E_OK = 0;
- IOC_E_NOK;
- IOC_E_LIMIT;
- IOC_E_LOST_DATA;
- IOC_E_NO_DATA.

16.16.3 IocSend

Syntax:	`Std_StatusType IocSend_<IocId>[_<SenderId>] ( OSBYTEPTR <Data> );`
Input:	<Data> - reference to the message data to be sent.
Output:	None.
Description:	The service performs an transmission of data elements with "event" semantic for a unidirectional 1:1 or N:1 communication between OSApplications located on the same or on different cores. The service is implemented as a macro generated by Sysgen if the corresponding IOC object is queued (BUFFER_LENGTH > 0). <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. It is the name of the corresponding IOC object. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. In case of 1:1 communication, it is omitted. This function is generated individually for each sender. The service adds a message data given by <Data> to the end of the message queue. It activates a notification mechanism of the receiving side of IOC.

Status:	• Standard:
	  • IOC_E_OK - no error.
	  • IOC_E_LIMIT - IOC internal communication buffer is full.
	  • IOC_E_NOK - IOC internal communication buffer is broken.
	• Extended:
	  • IOC_E_NOK - call while interrupts are disabled by OS services.
	  • IOC_E_NOK - call at not allowed context.
	  • IOC_E_NOK - illegal <Data> (for SC3,4).

Particularities:	If the queue of receiving message is full and a new message data arrives, this message data will be lost and notification will not be performed. The service is allowed in Tasks, ISRs Category 2 functions.
Conformance:	BCC1, ECC1

16.16.4 IocWrite

Syntax:	`Std_StatusType IocWrite_<IocId>[_<SenderId>] ( OSBYTEPTR <Data> );`
Input:	<*Data*> - reference to the message data to be sent.
Output:	None.
Description:	The service performs an transmission of data elements with "data" semantic for a unidirectional 1:1 or N:1 communication between OSApplications located on the same or on different cores. The service is implemented as a macro generated by Sysgen if the corresponding IOC object is unqueued (*last_is_best*), BUFFER_LENGTH is set to 0 or undefined. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. It is the name of the corresponding IOC object. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. In case of 1:1 communication, it is omitted. This function is generated individually for each sender. The service overwrites the current value of a message with data given by <Data>. It activates a notification mechanism of the receiving side of IOC.

Table continues on the next page...

Status:	• Standard:
	  • IOC_E_OK - no error.
	  • IOC_E_NOK - insufficient access rights.
	• Extended:
	  • IOC_E_NOK - call while interrupts are disabled by OS services.
	  • IOC_E_NOK - call at not allowed context.
	  • IOC_E_NOK - illegal <Data> (for SC3,4).

Particularities:	The service is allowed in Tasks, ISRs Category 2 functions.
Conformance:	BCC1, ECC1

16.16.5 IocReceive

Syntax:	`Std_StatusType IocReceive_<IocId> ( OSBYTEPTR <Data> );`
Input:	*<Data>* - reference to the message data to be received.
Output:	None.
Description:	The service performs an transmission of data elements with "event" semantic for a unidirectional 1:1 or N:1 communication between OSApplications located on the same or on different cores. The service is implemented as a macro generated by Sysgen if the corresponding IOC object is queued (BUFFER_LENGTH > 0). <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. It is the name of the corresponding IOC object. This function is generated individually for each receiver. This service reads the first value from the queue to the copy referenced by <Data> and then removes it from the queue.

Status:	• Standard:
	  • IOC_E_OK - no error.
	  • IOC_E_NOK - insufficient access rights.
	  • IOC_E_NO_DATA - no data is available for reception.
	  • IOC_E_LOST_DATA - internal queue overflow happened during last IocSend and data passed with it was lost. There is no error in the data returned by IocReceive however.
	• Extended:
	  • IOC_E_NOK - call while interrupts are disabled by OS services.
	  • IOC_E_NOK - call at not allowed context.
	  • IOC_E_NOK - illegal <Data> (for SC3,4).

Particularities:	The service is allowed in Tasks, ISRs Category 2 functions.
Conformance:	BCC1, ECC1

16.16.6 IocRead

Syntax:	`Std_StatusType IocRead_<IocId>(OSBYTEPTR <Data>);`
Input:	*<Data>* - reference to the message data to be received.
Output:	None.
Description:	The service performs an transmission of data elements with "event" semantic for a unidirectional 1:1 or N:1 communication between OSApplications located on the same or on different cores.

Table continues on the next page...

	The service is implemented as a macro generated by Sysgen if the corresponding IOC object is unqueued (*last_is_bes*t), BUFFER_LENGTH is set to 0 or undefined. \<IocId\> is a unique identifier that references a unidirectional 1:1 or N:1 communication. It is the name of the corresponding IOC object. This function is generated individually for each receiver. This service reads the current value of a message to the copy referenced by \<Data\>.

Status:	• Standard:
	  • IOC_E_OK - no error.
	  • IOC_E_NOK - insufficient access rights.
	• Extended:
	  • IOC_E_NOK - call while interrupts are disabled by OS services.
	  • IOC_E_NOK - call at not allowed context.
	  • IOC_E_NOK - illegal \<Data\> (for SC3,4).

Particularities:	The service is allowed in Tasks, ISRs Category 2 functions.
Conformance:	BCC1, ECC1

16.16.7 IocEmptyQueue

Syntax:	`Std_StatusType IocEmptyQueue_<IocId>(void);`
Input:	None.
Output:	None.
Description:	The service deletes the content of the IOC internal communication queue. The service is implemented as a macro generated by Sysgen if the corresponding IOC object is queued (BUFFER_LENGTH > 0). \<IocId\> is a unique identifier of queued communication. It is the name of the corresponding IOC object. This function is generated individually for each IOC.

Status:	• Standard:
	  • IOC_E_OK - no error.
	  • IOC_E_NOK - insufficient access rights.
	• Extended:
	  • IOC_E_NOK - call while interrupts are disabled by OS services.
	  • IOC_E_NOK - call at not allowed context.

Particularities:	The service is allowed in Tasks, ISRs Category 2 functions.
Conformance:	BCC1, ECC1

16.17 Debugging Services

These services are not defined by *AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3* specification. This is NXP OS extension of the AUTOSAR OS.

16.17.1 GetRunningStackUsage

Syntax:	`unsigned short GetRunningStackUsage ( void );`
Input:	None.
Output:	• amount of stack used by running task in bytes. • 0xFFFF if there is not any running task or the task uses 'single stack'.
Description:	The service returns amount of stack used by running task in bytes. The service returns 0xFFFF for basic task in SC1, when single stack is used.
Particularities:	The service is implemented if the value of the *STACKMONITORING* attribute is *TRUE* or the value of the *DEBUG_LEVEL* attribute is greater than 0. The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. In SC3 and SC4 the service call is allowed only from the Task.
Conformance:	BCC1, ECC1

16.17.2 GetStackUsage

Syntax:	`unsigned short GetStackUsage ( TaskType <TaskID> );`
Input:	<*TaskID*> - a reference to the task.
Output:	• amount of stack used by task <TaskID> in bytes. • 0xFFFF in SC1 if the task is basic (uses 'single stack').
Description:	The service returns stack usage by task <*TaskID*> in bytes.
Particularities:	The service is implemented if the value of the *STACKMONITORING* attribute is set *TRUE* or the value of the *DEBUG_LEVEL* attribute is greater than 0 but only for SC1 and SC2. The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.
Conformance:	

16.18 Operating System Execution Control

16.18.1 Data Types

The AUTOSAR OS establishes the following data type for operation mode representation:

- *StatusType* - the data type represent variable for saving system status;
- *AppModeType* - the data type represents the operating mode.

16.18.2 Constants

The following constant is used within the AUTOSAR OS to indicate default application mode:

- OSDEFAULTAPPMODE

  The constant of data type *AppModeType*. The constant is assigned to one of the application modes defined in the OIL file. This constant is always a valid parameter for *StartOS* service.

The following constants are used within the AUTOSAR Operating System to indicate system status. All of them have type StatusType. Status meaning is specified in service descriptions:

- E_OK
- E_OS_ACCESS
- E_OS_CALLEVEL
- E_OS_ID
- E_OS_LIMIT
- E_OS_NOFUNC
- E_OS_RESOURCE
- E_OS_STATE
- E_OS_VALUE
- E_OS_SERVICEID
- E_OS_ILLEGAL_ADDRESS
- E_OS_PARAM_POINTER
- E_OS_MISSINGEND
- E_OS_DISABLEDINT
- E_OS_STACKFAULT
- E_OS_SYS_FATAL
- E_OS_SYS_ORDER

  E_OS_SYS_xxx are not defined in the AUTOSAR Specification of Operating SystemV5.0.0 R4.0 Rev 3 specification. This is NXP OS extension of the AUTOSAR OS.

- IOC_E_OK
- IOC_E_NOK
- IOC_E_LIMIT
- IOC_E_NO_DATA

16.18.3 GetActiveApplicationMode

Syntax:	`AppModeType GetActiveApplicationMode( void );`
Input:	None.
Output:	Current application mode.
Description:	This service returns the current application mode.
Particularities:	Allowed on task level, on ISR level and in all hook routines.
Conformance:	BCC1, ECC1

16.18.4 StartOS

Syntax:	`void StartOS( AppModeType <Mode> );`
Input:	*<Mode>* - an operating mode.
Output:	None.
Description:	This service starts the operation system in a specified mode. If a *StartupHook* is configured, the hook routine is always called before starting the application.
Particularities:	Allowed outside of the operating system only. In NXP AUTOSAR OS this service returns to the caller only if it is called with the wrong parameter *Mode*. All interrupts are disabled inside the service.
Conformance:	BCC1, ECC1

16.18.5 ShutdownOS

Syntax:	`void ShutdownOS( StatusType <Error> );`
Input:	*<Error>* - a code of the error occurred.
Output:	None.
Description:	The service aborts the overall operating system. If a task is in the *running* state, *PostTaskHook* is not called. If a *ShutdownHook* is configured, the hook routine is always called (with *<Error>* as argument) before shutting down the operating system. If *ShutdownHook* returns, the operating system enters endless loop. (see System Shutdown).
Particularities:	*ShutdownOS* runs in connection with the currently active context, which may be unknown to the user. Thus only *GetActiveApplicationMode* and *GetApplicationID* services are allowed within the *ShutdownHook* routine. All interrupts are disabled inside the service. Allowed on task level, on ISR level and in StartupHook and ErrorHook hook routines.
Conformance:	BCC1, ECC1

16.18.6 OS System Timers control

These functions are intended to stop OS system timers before switch to low-power mode and start OS timers after return from that mode.

It is desirable to cancel OS alarm and stop Schedule tables (which are bound with OS counters on OS system timers) before stop of timers because HW timer counter may be cleared after following start of timers . It is user's responsibility to cancel/stop Alarms/ Schedule Tables.

NOTE

When OS system timer prescaler is set as USER (not OS), the 'start' functions only enable interrupts from Timer HW. The user shall perform remaining initialization himself before OS_StartTimers/OS_StartTimers2 call.

It is recommended to call these functions under suspended OS interrupts.

Table 16-1. OS_StartTimers

Syntax:	`void OS_StartTimers( void );`
Description:	The function provides start of HW timers which are underlying for SysTimer/SecondTimer (if configured) in the same manner as in StartOS. This function is FSL extension of AUTOSAR OS.
Particularities:	The function is allowed to be called only in CPU supervisor mode. The function shall be called only on the master core (Core 0).
Conformance:	BCC1, ECC1

Table 16-2. OS_StartTimers2

Syntax:	`void OS_StartTimers2( void );`
Description:	The function provides start of HW timers which are underlying for SysTimer2/SecondTimer2 (if configured) in the same manner as in StartOS. This function is FSL extension of AUTOSAR OS.
Particularities:	The function is allowed to be called only in CPU supervisor mode. The function shall be called only on the second core (Core 1).
Conformance:	BCC1, ECC1

Table 16-3. OS_StopTimers

Syntax:	`void OS_StopTimers( void );`
Description:	The function provides stop of HW timers which are underlying for SysTimer/SecondTimer (if configured) in the same manner as in ShutdownOS. This function is FSL extension of AUTOSAR OS.
Particularities:	The function is allowed to be called only in CPU supervisor mode. The function shall be called only on the master core (Core 0).
Conformance:	BCC1, ECC1

Table 16-4. OS_StopTimers2

Syntax:	`void OS_StopTimers2( void );`
Description:	The function provides stop of HW timers which are underlying for SysTimer2/SecondTimer2 (if configured) in the same manner as in ShutdownOS. This function is FSL extension of AUTOSAR OS.
Particularities:	The function is allowed to be called only in CPU supervisor mode. The function shall be called only on the second core (Core 1).
Conformance:	BCC1, ECC1

16.18.7 Hook Routines

The hook routines is called by the OS, but shall be implemented by the User, if configured.

Interrupts of category 2 and TP interrupt are disabled in all hook routines.

See Hook Routines for general description of hook routines.

16.18.7.1 ProtectionHook

Syntax:	`ProtectionReturnType ProtectionHook( StatusType error );`
Input:	*<Error>*- a code of the error occurred.
Output:	None.
Description:	To implement *ProtectionHook* the routine with the name 'ProtectionHook' shall be defined in the user's code. This hook is called if serious error occurs. E.g. exceeding the worst case execution time or violating against the memory protection.
Return:	• PRO_TERMINATETASKISR - the OS shall terminate running Task or ISR and perform an appropriate cleanup. • PRO_TERMINATEAPPL - the OS shall kill active OSApplication. • PRO_TERMINATEAPPL_RESTART - the OS shall kill active OS-Application and then restart RESTARTTASK of this OSApplication. • PRO_SHUTDOWN - the OS shall perform shutdown. • PRO_IGNORE - the OS shall do nothing, allowed only in case of E_OS_PROTECTION_ARRIVAL.
Particularities:	This hook is not called if the system configuration option PROTECTIONHOOK is set to FALSE. In this case the shutdown is performed in case of any violation.
Conformance:	BCC1, ECC1 within SC2, SC4

16.18.7.2 ErrorHook

Syntax:	`void ErrorHook( StatusType <Error> );`
Input:	*<Error>* - a code of the error occurred.

Table continues on the next page...

Output:	None.
Description:	To implement *ErrorHook* the routine with the name 'ErrorHook' shall be defined in the user's code. This routine is called by the operating system at the end of a system service which has a return value not equal to E_OK if not specified other. It is called before returning from the service. This hook is also called from OS dispatcher when an error is detected during task activation or event setting as a result of an alarm expiry or a message arrival, or if a rescheduling occurs after *Suspend(OS/All)Interrupts* before call to appropriate Resume function.
Particularities:	*ErrorHook* can not be nested. Therefore the error hook is not called, if a system service called from *ErrorHook* and does not return E_OK as a status value. If ErrorHook is called from inside the service, called from other hook then it inherits the context of enclosing hook and the restrictions for that hook are applied. This hook is not called if the system configuration option *ERRORHOOK* is set to *FALSE*. There is a set of special macros to get the ID of service where error has occured and it's first argument - see Macros for Errorhook.
Conformance:	BCC1, ECC1

16.18.7.3 PreTaskHook

Syntax:	`void PreTaskHook( void );`
Input:	None.
Output:	None.
Description:	To implement *PreTaskHook* the routine with name 'PreTaskHook' shall be defined in user's code. This hook routine is called by the operating system before executing a new task, but after the transition of the task to the running state (to allow evaluation of the task ID by *GetTaskID* services). This hook is called from the scheduler when it passes control to the given task. It may be used by the application to trace the sequences and timing of tasks' execution.
Particularities:	This hook is not called if the system configuration option *PRETASKHOOK* is set to FALSE.
Conformance:	BCC1, ECC1

16.18.7.4 PostTaskHook

Syntax:	`void PostTaskHook( void );`
Input:	None.
Output:	None.
Description:	To implement *PostTaskHook* the routine with name 'PostTaskHook' shall be defined in user's code. This hook routine is called by the operating system after executing the current task, but before leaving the task's running state (to allow evaluation of the task ID by *GetTaskID*).This hook is called from the scheduler when it switches from the current task to another. It may be used by the application to trace the sequences and timing of tasks' execution.
Status:	None.
Particularities:	*PostTaskHook* is not called if running task is exist and OS is aborted by *ShutdownOS* service. This hook is not called if the system configuration option *POSTTASKHOOK* is set to *FALSE*.
Conformance:	BCC1, ECC1

16.18.7.5 PreIsrHook

Syntax:	`void PreIsrHook( void );`
Input:	None.
Output:	None.
Description:	To implement *PreIsrHook* the routine with name 'PreIsrHook' shall be defined in user's code. This hook routine is called by the operating system before executing a ISR category 2 or System Timer ISR with all interrupts disabled. It may be used for debugging purposes only. This hook is not defined in AUTOSAR OS v.4.0.3 specification, it is NXP OS extension of the AUTOSAR OS.
Particularities:	This hook is not called if the system configuration option IsrHooks is set to FALSE.
Conformance:	BCC1, ECC1

16.18.7.6 PostIsrHook

Syntax:	`void PostIsrHook( void );`
Input:	None.
Output:	None.
Description:	To implement *PostIsrHook* the routine with name 'PostIsrHook' shall be defined in user's code. This hook routine is called by the operating system after executing a ISR category 2 or System Timer ISR with all interrupts disabled. It may be used for debugging purposes only. This hook is not defined in AUTOSAR OS v.4.0.3 specification, it is NXP OS extension of the AUTOSAR OS.
Status:	None.
Particularities:	This hook is not called if the system configuration option *IsrHooks* is set to *FALSE*.
Conformance:	BCC1, ECC1

16.18.7.7 StartupHook

Syntax:	`void StartupHook( void );`
Input:	None.
Output:	None.
Description:	To implement *StartupHook* the routine with name 'StartupHook' shall be defined in user's code. This hook is called by the operating system at the end of the operating system initialization and before the initialization of Interrupt Sources, System and Second timers and before scheduler starts running. At this point in time the application can initialize hardware, etc. All IPs that are used by this specific Autosar OS implementation must have the clock source enabled before StartOS() is called. This must not be delayed until StartupHook() as HW IPs are pre-initialized by OS before StartupHook() is called.
Status:	None.
Particularities:	All interrupts are disabled in *StartupHook*. This hook is not called if the system configuration option *STARTUPHOOK* is set to *FALSE*.
Conformance:	BCC1, ECC1

16.18.7.8 ShutdownHook

Syntax:	`void ShutdownHook( StatusType <Error> );`
Input:	*<Error>* - a code of the error occurred.
Output:	None.
Description:	To implement *ShutdownHook* the routine with name 'ShutdownHook' shall be defined in user's code. This hook is called by the operating system when the *ShutdownOS* service has been called. This routine is called during the operation system shut down. User can avoid return from the hook to calling level. For example reset signal can be generated in the hook.
Status:	None.
Particularities:	All interrupts are disabled in *ShutdownHook*. This hook is not called if the system configuration option *ShutdownHook* is turned off in the configuration file. This hook is not called if the system configuration option *SHUTDOWNHOOK* is set to *FALSE*.
Conformance:	BCC1, ECC1

Chapter 17
Debugging Application

17.1 Debugging Application

This chapter provides information about NXP AUTOSAR OS feature intended for debugging a user application with OS.

This chapter consists of the following sections:

- General
- Using OS Extended Status for Debugging
- Context Switch Monitoring
- ORTI Features
- Stack Debugging Support

17.2 General

AUTOSAR OS contains several mechanisms which help user to debug application.

17.2.1 Extended Status

Extended status allows checking most of errors caused by improper use of AUTOSAR services.

17.2.2 ORTI

The purpose of OSEK Run Time Interface (ORTI) implementation is giving the user extended opportunities in debugging embedded OSEK and AUTOSAR applications. The OSEK Run Time Interface implementation confirms *OSEK/VDX OSEK Run Time*

Interface (ORTI), Part A: Language Specification, v. 2.2, 14 Nov 2005 and *OSEK/VDX OSEK Run Time Interface (ORTI), Part B: OSEK Objects and Attributes, v. 2.2, 25 Nov 2005*. The current version of NXP AUTOSAR OS supports ORTI version 2.1

The ORTI shall be supported from both sides: an AUTOSAR OS and a debugger. The debugger able to display information in terms of OSEK system objects is 'OSEK aware' debugger. The internal OS data is to be made available to the debugger. For this purpose special ORTI file is generated at configuration time by a System Generator. As a result, more information will be available to the user during application debugging session.

System Generator (SysGen) uses OIL file (App.oil) as an input file. Option -o of the SysGen defines output ORTI file name. The SysGen utility generates static information in the ORTI format. This utility analyzes the application configuration and generates ORTI file. The same OIL file is used for configuring OS. After application is compiled and linked and executable and map files are created then they are loaded by the debugger. If the debugger is OSEK aware then it can load also the ORTI file for this application. The information from ORTI file provides the debugger with possibility to display information about system objects of current implementation of the OS. This process is depicted in the following figure:

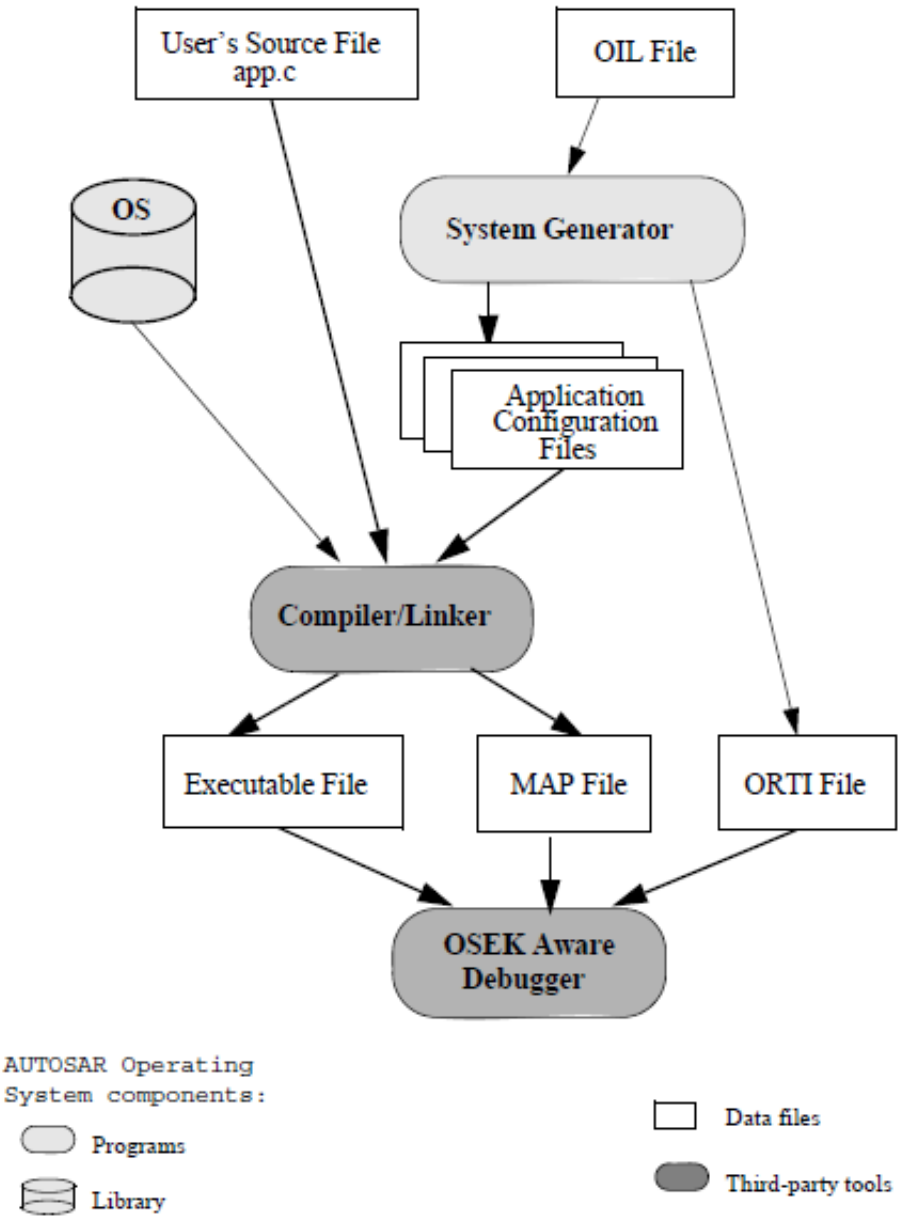

Figure 17-1. Application Building Process with ORTI Support

17.3 Using OS Extended Status for Debugging

It is recommended to use Operating System *Extended Status* when developing an application to analyze return codes of system services. Such OS configuration is more memory and time consuming but it allows the user to save time for errors eliminating. Error codes returned by the AUTOSAR OS services covers most of possible errors that can arise during development. Therefore it is useful to check these codes after a service

call to avoid error that can lead to the system crash. For example, a task can perform the *TerminateTask* service while it is still occupying a resource. This service will not be performed and the task will remain active (*running*). In case of Extended Status the *E_OS_RESOURCE* error code is returned and it is possible to detect this situation. But in the system without Extended Status there is no additional check and this error is not indicated and the application behavior will be unpredictable!

After all errors in the application are eliminated the *Extended Status* may be turned off to remove additional status checks from the application and get the reliable application of the smaller size.

17.4 Context Switch Monitoring

ORTI Trace Interfaces provides the user with the ability to trace application execution, including context switching. But there is an ability to monitor context switching without using ORTI. Breakpoints, traces and time stamps can be integrated individually into application software with the help of context switch hook routines *PreTaskHook* and *PostTaskHook*.

Additionally, the user can set time stamps enabling him to trace the program execution, for example, at the following locations before calling operating system services:

- When activating or terminating tasks;
- When setting or clearing events in the case of Extended Tasks;
- At explicit points of the schedule (*ChainTask*, *Schedule*);
- At the beginning or the end of ISR;
- When occupying and releasing resources or at critical locations.

17.5 ORTI Features

ORTI provides two kinds of interfaces to the OS data:

- Trace interface, which means getting an access to the data on a running target when it is essential to trace data changes in a real time
- Breakpoint interface, which provides an access to desirable data on a stopped target.

Note that ORTI Trace interface is designed to provide access to requested data in accomplished form that is not requiring an additional processing.

17.5.1 ORTI Trace Interfaces

There are trace ORTI interfaces in run-time: running task identification, ISR identification, AUTOSAR OS system calls identification. This interface is turned on if DEBUG_LEVEL = 2. If DEBUG_LEVEL = 3 then only running Task and ISR trace is supported.

The special attributes are generated for the OS object in ORTI file to support trace interfaces, the names of these attributes corresponds to ORTI specification supported. The following trace interfaces are available to the user:

1. running task

   This attribute specifies the name of the currently running task within the OS.

   The value of this attribute presents a one byte memory block which contains either byte numeric identifier of a task which is currently in the *running* state or the special byte value in case of none tasks are in the *running* state. The certain values of task identifiers are statically determined and enumerated in the type field of running task attribute of an OS object as well as the special value (*NO_TASK*) representing none of tasks running.

   The value of the running task attribute is updated each time CPU switches to another task or to the code corresponding to none of task running.

2. running ISR

   This attribute specifies the identifier of currently executed ISR of category 2. It occupies a one byte memory block.

   There are the following special values of the attribute:
   - NO_ISR - OS works on task or dispatcher level.
   - SYSTEM_TIMER - System Timer ISR is activate.
   - SECOND_TIMER - Second Timer ISR is activate.

   Other values correspond to ISRs of category 2 defined in the OIL file.

   The next scheme is used for ISR tracing: running ISR attribute has TASK_LEVEL value when OS works at task level. When one of ISRs category 2 starts execution, the value of running ISR is changed to ISR identifier. When SystemTimer ISR gets CPU, the value of running ISR is changed to SYSTEM_TIMER. When ISR terminates and OS returns to task level, running ISR gets value TASK_LEVEL.

3. current service

   This attribute specifies which operating service, if any, is currently being executed.

The current service attribute value presents a one byte memory block which represents a service numeric identifier. This value is updated with AUTOSAR OS service (*Service*) identifier with least significant bit set when CPU enters *Service* code.

In case of leaving AUTOSAR OS service the attribute value is set to service identifier with least significant bit cleared thus indicating end of service. The attribute is updated with such value on the following events occurred:

- CPU leaves AUTOSAR OS service code and starts to execute non AUTOSAR OS service code - i.e. the application code in the same or another task;
- CPU leaves AUTOSAR OS service code and starts to execute idle loop.

Typically trace sequence of current service looks like the following:

Table 17-1. ORTI Trace Sequence

Traced Value	Description
...	
SERVICE_1_ID	entering Service 1
SERVICE_1_EXIT	leaving Service 1 (possibly due to task switching)
...	
SERVICE_2_ID	entering Service 2
SERVICE_3_ID	entering nested Service 3 call (e.g. hook routine)
SERVICE_3_EXIT	leaving Service 3 and resuming Service 2
SERVICE_2_EXIT	leaving Service 2 (possibly due to task switching)
...	

17.5.1.1 Hardware-specific implementation

Multibyte OTM (Ownership Trace Messaging) is supprted on NXP AUTOSAR OS/S32K via Nexus Class 2+ interface.

The following trace interfaces are available to the user:

- running task
- running ISR
- current service

17.5.2 ORTI Breakpoint Interface

There is the ORTI Breakpoint interface intended to facilitate debugger access to task related data. The interface is turned on, if DEBUG_LEVEL OIL attribute does not equal to 0.

The following static (at breakpoints) ORTI services are supported for a debugger on breakpoints: access to tasks, stacks, counters, alarms, resources information.

Information needed to display the current status of objects is available for the debugger whenever the debugger is stopped (i.e. this information is not required in real-time and hence can be read from the TCB or similar structure).

Information in the ORTI file allows a debugger to display task information using values that the user sets in the OIL file.

The following information about OS is available to the user:

- current application mode

  This attribute specifies the current application mode.

  The current application mode value presents a one byte memory block which represents a mode identifier.

- running task priority

  This attribute specifies the current priority of the task referred by the running task attribute.

  The value of this attribute is provided with taking into account possible task priority changes due to a dynamic resource allocation.

- last error

  This attribute specifies the code of the last error detected. The last error attribute value presents a one byte memory block which represents an error code identifier. This value is updated with error identifier when error occurs. At start up the error code is initialized with E_OK. It is never set back to E_OK after first error.

The following task information is available to the user:

- property

  The task property indicates static properties of the task.

- priority

  The task priority value is provided with taking into account possible task priority changes due to a dynamic resource allocation.

- task state

  The task state indicates a standard AUTOSAR state the task is currently in.

- task stack

  The task stack shows which stack area is used by the task.

- event states

  The event states can be used to determine events which are currently set or cleared.

- wait events

  Wait events value contains bit set to one for event which task is waiting for.

- event masks

  The event masks define correspondence between event name and bit mask.

The following stack information is available to the user:

- size

  The size specifies the size (in bytes) of a memory area allocated for stack.

- base address

  The base address specifies the lowest address of stack memory area.

- stack direction

  The stack direction specifies the direction of stack growth and may have 'UP' or 'DOWN' as its value. 'UP' means growing from lower to higher addresses. 'DOWN' means growing from higher addressed to lower addresses.

- fill pattern

  The fill pattern attribute specifies with which pattern the stack area is initialized. This allows the debugger to evaluate the maximum stack usage.

The following counter information is available to the user:

- maxallowed value
- ticksperbase
- mincycle
- current counter value
- indicator if alarms attached to counter is activated
- property, which indicates OSTICKDURATION, the type of the counters (SWCOUNTER or HWCOUNTER) and timer hardware for system timers or indicates if it is an application counter

The following alarm information is available to the user:
- alarm state
- assigned counter
- action on alarm expiration
- time left to expire
- cycle value for cyclic alarm

The following schedule table information is available to the user:
- schedule table state
- assigned counter
- synchronisation strategy
- schedule table duration
- repeating
- schedule table explicit precision
- schedule table deviation
- schedule table local time
- time to expire (it is time to a next expiry point or to the logical end of the examined schedule table. It depends on the current local time of the schedule table)

To identify which expiry point will be next it is necessary:

- to calculate an estimated offset as sum "schedule table local time" and "time to expire" from the list below;
- to compare the estimated offset with offsets of expiry points from OS-configuration.

If the estimated offset is equal to the length of the schedule table from OS-configuration, the "time to expire" should be treated as the time to the logical end of the schedule table.

The estimated offset will be equal OS-configuration values if the implicit synchronization is used or schedule table is without synchronization. In case of explicit synchronization the estimated offset may not correspond OS-configuration values until the end of synchronization.

The following schedule table expiry point information is available to the user:
- schedule table name
- offset of expiry point

The following action information for schedule table expiry point is available to the user:
- expiry point name
- action when the alarm expires

The following resource information is available to the user:
- priority of the resource
- resource state

17.6 Stack Debugging Support

17.6.1 Stack Overflow Checking

Optional stack overflow checking can be used during run time to check tasks, ISR and main stacks usage. Main and task stacks are checked during task switch. If stack overflow is detected *ShutdownOS* with E_OS_STACKFAULT status is called. OS stack overflow control is turned on if *STACKMONITORING* attribute value is TRUE.

System checks task's and main stacks for overflow when tasks state is changed from *RUNNING* and before entering ISR of category 2. ISR stack (or main stack in case of BCC1, SC1) is checked before leaving ISR of category 2.

Chapter 18
Sample Application

18.1 Sample Application

This appendix contains the text and listing of the sample customer application developed using NXP AUTOSAR OS.

This appendix consists of the following sections:

- Description
- Source Files

18.2 Description

The Sample applications delivered with the NXP AUTOSAR OS should help to learn how to use AUTOSAR OS. The Sample's source files are located in the `sample\standard` directory - it contains all files needed to create an executable files.

The samples might be configured with help of ElectroBit tresos Studio (version is specified in the sample readme file). The 'os_ts' directory contains example of Tresos configuration project for OS module based on OS samples.

The sample applications are configured to run on the EVB from internal flash with or without debugger. Please see the sample readme file for details.

There is one sample for each supported scalability class, with similar algorithms. This release contains support for scalability classes SC1. There is script for Lauterbach debugger (`sample\standard\s32k.cmm`) that allows loading the sample onto board. LAUTERBACH_PATH variable shall be set.

The Sample is not a real application and it does not perform any useful work. But it has a certain algorithm so it is possible to track the execution. It uses most of AUTOSAR OS mechanisms and allows the user to have the first look inside the AUTOSAR OS.

Each Sample consists of six tasks. It uses three counters (HW and SW are on the System Timer and Second Timer and one 'user counter'), two alarms, ScheduleTable, two resources and two IOCs.

Generally, Sample tasks are divided into two parts. *TASKSND1*, *TASKSND2* and *TASKCNT* compose the first part (`samplets.c` file) and *TASKRCV1*, *TASKRCV2* and *TASKSTOP* are the second part (`samplerv.c` file). These two parts interact with the help of the IOCs and alarm mechanism.

The Extended task TASKRCV1 is activated by OS (autostarted). It performs the following initializations:

- init *STCOUNTER* counter with value 0,
- set absolute *STOPALARM* alarm to value 20 (when *STOPALARM* expired it activates *TASKSTOP* task),
- starts *ScheduleTable*,
- clears *MSGAEVENT* and *TIMLIMITEVENT* events,
- set relative *TIMLIMITALARM* alarm to value 20 (when *TIMLIMITALARM* expired it sets *TIMLIMITEVENT* event for this (*TASKRCV1*) task).

Then it enters *waiting* state - waiting *MSGAEVENT* and/or *TIMLIMITEVENT* events. When event occurred, *TASKRCV1* checks which event occurs.

If *MSGAEVENT* event occurred (*MsaA* arrived) then *TASKRCV1* task:

- cancels *TIMLIMITEVENT* alarm,
- gets *MSGAACCESS* resource to prevent access to *MsgA* ,
- receives *MsgAReceived*,
- releases *MSGAACCESS* resource,
- clears event and goes to waiting state again.

If *TIMLIMITEVENT* event occurred (time limit was exceeded) then *TASKRCV1* task goes to the very beginning and repeats initialization, restarting the application (but OS is not restarted, it continue running).

ScheduleTable has three steps (action points):

1. On the first step *TASKSND1* task is activated. It does the following:
   - gets *MSGAACCESS* resource to prevent access to *Ms*gA,
   - increments *MsgA* value,
   - send *MsgA* (*MsgA* sets *MSGAEVENT* event for *TASKRCV1* task),
   - releases *MSGAACCESS* resource,
   - terminates itself.
2. On the second step *TASKSND2* task is activated. Task *TASKSND2*:
   - adds 30 to 'ind' variable value,
   - if 'ind' variable value greater or equal 50 then subtracts 51 from the 'ind' value,

- stores *SYSTEMTIMER* value to body of *MsgB* copy,
- gets *MSGBACCESS* resource to prevent access to *MsgB*,
- send *MsgB* (*MsgB* activates *TASKRCV2* task),
- releases *MSGBACCESS* resource,
- terminates itself.

3. On third step *TASKCNT* task is activated. *TASKCNT* task:
   - increments *STCOUNTER* counter (when *STCOUNTER* counter value reaches 20 the alarm '*STOPALARM*' expires),
   - terminates itself.

TASKRCV2 task:

- gets *MSGBACCESS* resource to prevent access to MsgB,
- receives MsgBReceived IOC without copy,
- releases MSGBACCESS resource,
- terminates itself.

TASKSTOP task:

- stops ScheduleTable (after a while TIMLIMITALARM expires)
- terminates itself.

The user can watch 'ind' variable, messages content and so on.

18.3 Source Files

Source files for the Sample application are the following:

- `samplets1.c`

  the application code (TASKSND1, TASKSND2 and TASKCNT)

- `samplerv1.c`

  the application code (TASKRCV1, TASKRCV2 and TASKSTOP)

- `sample.h`

  header file for the application code

- `MemMap.h`

  header file for memory sections control

- `sample1.oil`

  configuration OIL file for single core

- `sample1.epc`

  sample configuration in AUTOSAR XML .

- `makefile`

make file for compiling sample using GNU make utility

- linker command file

The directory structure of the Sample application is described in the readme.txt file located in the `sample\standard` directory.

To the description of sample build please refer to the sample readme file.

Chapter 19
System Service Timing

19.1 General Notes

This appendix provides information about OS services execution time. All measurements were done on MCU.

The measurements are provided in the timing excel files located in the ***<installation path>\doc*** folder. The results in the tables were obtained on the basis of the certain OS configuration. The list of system properties and this configuration are called in the table as 'Initial'. Properties that are not listed have their default values. In the column 'Configuration' the differences from the given list ('Initial') are indicated. For each configuration the corresponded numbers are provided in the table. In the column 'Conditions' the specific details for service execution are indicated.

Four tasks are present in the application.

Chapter 20
Memory Requirements

20.1 Memory Requirements

This appendix provides information about the amount of ROM and RAM directly used by various versions of the NXP AUTOSAR OS. The numbers in the tables may be not quite correct due to last minute changes in the OS code.

This appendix consists of the following sections:

- NXP Autosar OS Memory Usage

20.2 NXP Autosar OS Memory Usage

The table below contains the data about ROM and RAM needed for the Operating System kernel and system objects. The amount of memory depends on the system configuration and on the number of certain objects (e.g., tasks, counters, etc.). The table does not reflect all possible configurations so the overall number of them is too big. Therefore, only some most important configurations are presented.

The following initial system property settings were used to determine memory requirements:

```
CC = BCC1;
STATUS = STANDARD;
STARTUPHOOK = FALSE;
SHUTDOWNHOOK = FALSE;
PRETASKHOOK = FALSE;
POSTTASKHOOK = FALSE;
ERRORHOOK = FALSE;
USEGETSERVICEID = FALSE;
USEPARAMETERACCESS = FALSE;
USERESSCHEDULER = FALSE;
STACKMONITORING = FALSE;
ACTIVATION = 1; (for all TASK objects)
SCHEDULER = FULL; (for all TASK objects)
no ISRs defined
```

This initial property list was used for the first row in the table. It conforms to the SC1, BCC1 classes without any additional mechanisms and this is the minimal AUTOSAR OS configuration. The rows below reflects memory requirements for the other OS configurations. System properties are shown in the rows which are turned on for the corresponded Conformance Class.

All other rows below the first one ('Initial') has a title 'Initial' or 'Changed:' and one or more options turned ON or OFF. If a row has a title 'Initial' it means that for such OS configuration the Initial property list is used with particular options changed as shown. If a row has a title 'Changed:' it means that for such OS configuration the setting list as for the previous row is used with particular options changed as shown.

Since each system object (a task, an alarm, etc.) requires some ROM and RAM the total amount of memory depends on the number of objects. Therefore, the formulas should be used to calculate the exact memory amount for each case. These formulas are provided in the table.

Data presented in the table do not include ISR, main and Task's stacks for RAM; the compiler library and startup code, task functions and the primary vector table for ROM. However, the stacks used by OS internally for protection handlers are included into RAM consumption numbers.

The RAM and ROM memory data does not include compiler padding of C variables for alignment, so actual RAM size may be more than calculated.

The measurements are provided in the memory excel files.

Chapter 21
System Generation Error Messages

21.1 General Notes

This appendix explains AUTOSAR OS System Generator error messages.

The System Generator checks the compatibility of properties, parameters and limits and reports about possible errors via error messages. The error messages can be associated with the wrong syntax, mistakes in the implementation definition, wrong definitions of the application objects.

This appendix consists of the following sections:

21.2 Severity Level

The messages vary in their severity level, they can be one of the following types:

- **information**,
- **warning**,
- **error**,
- **fatal error**

Usually an information message attends other type of error message and contains reference to necessary information associated with error situation. A warning message only prevents about possible error. If an error message is detected, than the operation that should be started after the current one, is will not be executed. For example, if the error messages were found in project verification, the configuration file will not be generated however project settings check will be continued. When a fatal error message is found, than anyone of build command is terminated.

21.3 Error Message Format

The error message format depends on mode in which SysGen has been running. By default an error message includes the file name, the line number, the error code and a short error description. The error messages have one of the following formats in accordance with the severity level of message:

```
[<filename>(<line_number>) : ]information ####:<message>
[<filename>(<line_number>) : ]warning ####:<message>
[<filename>(<line_number>) : ]error ####:<message>
[<filename>(<line_number>) : ]Fatal Error ####:<message>

where:
<filename>- file name
<line_number>- line number
#### - number of message
<message>- short description of the error
```

When the input file format is XML SysGen outputs the full XML path instead of [<filename>(<line_number>)] part of message.

21.4 List of Messages

0002	**The attribute is defined but not used** *Warning* The defined attribute is not used.
0003	**Option <option> is used; the other options are ignored** *Information* If command line option <option> is used, other options are ignored.
0006	**"Events" are not allowed for '<class>'** *Error* According to OSEK OS spec. the events are not supported in Basic conformance classes. This message is generate if the EVENT object is defined, but one of the Basic conformance classes is defined for OS.
0007:	**The event is defined in <class>** *Information* According to OSEK OS spec. the events are not supported in Basic conformance classes. This message is generate for each EVENT object defined when BCC is defined for OS.
0023:	**An ISR stack size shall be defined** *Error* The IsrStackSize attribute shall be defined for SC2 and for ECC1 within SC1 if the application contains ISR category 2 or System Timer.
0038:	**At least one OS/TASK/APPMODE shall be defined** *Error* At least one object of OS, TASK, and APPMODE types should be defined in an application. Only one OS object has to be defined.
0046:	**The same counter cannot be used for both a System Timer and a Second Timer**

Table continues on the next page...

	Error The same counter cannot be attached to both system and second timers.
0051:	**The extended task is not supported in BCC1** *Error* The OS conformance class is defined as BCC1 class, but extended task is defined in the application via EVENT object.
0056:	**Category 1 ISR cannot have a resource reference** *Error* The RESOURCE reference cannot be defined for ISR object if its CATEGORY attribute is 1.
0058:	**ISR cannot have a reference to an internal resource** *Error* The message is generated if the RESOURCE attribute of the ISR object refers to the RESOURCE object with the RESOURCEPROPERTY attribute with INTERNAL value.
0065:	**The basic task cannot be notified by a Set Event service** *Error* If the referenced task has no events, then ACTION attribute of ALARM and MESSAGE objects cannot be defined as SETEVENT.
0066:	**The task has no event references** *Information* The referenced task is a basic one.
0072:	**Event <name> is not referenced from task <name>** *Warning* The task has no event, which is referenced by the EVENT reference of the ALARM object.
0073:	**More than one task per priority is defined** *Error* Only one task per priority may be used, i.e., each task should have unique priority.
0086:	**The basic task does not require a stack size** *Warning* In SC1 the value of the STACKSIZE attribute is ignored for basic tasks.
0087:	**Category 1 ISR does not require a stack size** *Warning* The value of the STACKSIZE attribute is ignored for ISRs of category 1.
0088:	**In SC1 and SC2 classes, the stack size value of ISR is ignored** *Warning* In SC1 and SC2 the ISRs use common stack - "single stack" in BCC1 within SC1, common ISR stack otherwise.
0101:	**Name or ACCESSNAME shall be C-identifier** *Error* Only C-identifier can be used in the Name or ACCESSNAME attributes' names.
0107:	**Identifier '<name>' is longer than <number>. This may be the cause of compilation problems** *Warning* If object identifier is longer than 32 characters, the result of compilation depends of compiler used. Make identifier shorter if there is an error during compilation of configuration or application source files.

Table continues on the next page...

0109:	**The maximum allowed value of the hardware counter is assumed to be <number>. The defined value will be changed** *Warning* Hardware counter has hardware defined maximal value for counter, so it cannot differ from assumed value. Counter information available via API will be changed to assumed value.
0116:	**One OS/APPMODE shall be defined** *Error* One object of OS and at least one APPMODE types shall be defined in an application.
0120:	**The usage of Scheduler resource is disabled, but the resource 'RES_SCHEDULER' is defined** *Error* RES_SCHEDULER is supported if the value of the USERESSCHEDULER attribute is TRUE.
0123:	**The task can have only one internal resource** *Error* The TASK object can has only one reference to the RESOURCE object which has the RESOURCEPROPERTY subattribute with the INTERNAL value.
0201:	**<Target Name> target is not supported by your license!** *Fatal Error* The license for target platform is not provided.
0202:	**License <featurename> expires in <number> days** *Warning* The message warns when the license expires in less than 30 days. The warning appears every work session.
0203:	**License <featurename> will expire tonight at midnight** *Warning* This warning appears when the license expires at this day.
0252:	**The priority of Category 2 ISR is higher than a priority of Category 1 ISR** *Error* The priority of any ISR category 2 shall be lower than priority of any ISR category 1.
0253:	**The priority of Category 1 ISR is lower than a priority of Category 2 ISR** *Error* The priority of any ISR category 2 shall be lower than priority of any ISR category 1.
0254:	**The priority of <name> is higher than a priority of Category 1 ISR** *Error* The priority of any System or Second timer (considered as ISR category 2.) shall be lower than priority of any ISR category 1.
0264:	**Channel <number> of timer <name> can not be defined for several timers** *Error* The same hardware can be used by the System and Second timer only if different Channel attribute values are defined.
0500:	**An ORTI license is not found. The ORTI file will not be generated** *Warning* ORTIFULL license shall be installed for ORTI files generation.
0501:	**The event mask does not fit into <number> bits**

Table continues on the next page...

	Error The number of supported event is 32 per task, but this error can be generated also in case if some events have AUTO mask but other events have user defined mask with several bits set.
0502:	**The event mask can not be equal to zero** *Error* The number "0" can not be used as a mask.
0507:	**<name> <object type> is declared but not used** *Warning* The resource Ceiling priority is calculated automatically on the basis of information about priorities of tasks using the resource. If RESOURCE object is defines but not used, calculation of resource Ceiling priority is incorrect.
0508:	**Resource <name> is linked in a cycle** *Error* The RESOURCE can not be linked to itself.
0509:	**The internal resource can not be linked** *Error* This message is generated if a RESOURCE object with RESOURCEPROPERTY = LINKED refers through the LINKEDRESOURCE subattribute to a RESOURCE object with RESOURCEPROPERTY = INTERNAL.
0513:	**The Second Timer is a hardware type, but the System Timer is not** *Error* The SecondTimer attribute can be SWCOUNTER if the SysTimer is equal to SWCOUNTER or HWCOUNTER. The SecondTimer can be HWCOUNTER if the SysTimer attribute is HWCOUNTER.
0514:	**A System Timer shallshall be defined** *Error* The SysTimer attribute is not defined.
0515:	**SecondTimer hardware cannot be the same as SysTimer** *Error* The TimerHardware attributes inside SysTimer and SecondTimer attributes along with Timer and Channel attributes identify a hardware used for System and Second Timers. This hardware shall be different for each Timer, i.e. each Timer shall have its own interrupt source.
0516:	**Prescaler attributes shall be both 'USER' or 'OS'** *Error* The Prescaler attribute in the SysTimer attribute scope and the Prescaler attribute inside the SecondTimer attribute shall have the same value.
0750:	**Event <name> and event <name> share a bit and both used by task <name>** *Warning* Sharing of the same bit by event masks inside one task may cause unexpected event notification and/or misinterpretation of the event.
0753:	**The period is out of representable range (<number>)** *Warning* The message is generated if the period size is more than 32 bits, the value of OSTICKDURATION attribute is set to 0xFFFFFFFF.
0755:	**The name of an OS property file is not defined explicitly, <filename> will be used** *Warning*

Table continues on the next page...

	Sysgen uses default file name, use the -p option to define location of OS property file.
0756:	**The name of an OS header configuration file is not defined explicitly, <filename> will be used**
	Warning
	Sysgen uses default file name, use the -h option to define the name of OS header file.
0757:	**The name of an OS source configuration file is not defined explicitly, <filename> will be used**
	Warning
	Sysgen uses default file name, use the -c option to define the name of OS source file.
0758:	**The name of an ORTI file is not defined explicitly, <filename> will be used**
	Warning
	Sysgen uses default file name, use the -o option is used to define the name of ORTI file.
0759:	**The debug level equals to 0. Option -o is ignored. An ORTI file will not be generated**
	Warning
	If the DEBUG_LEVEL attribute is set 0, then ORTI is not supported, and ORTI file is not generated.
0765:	**The event is treated as it is used by all the tasks**
	Information
	The EVENT object is not referred in any Task description. This means that event can be used by any task.
0766:	**The event is declared but never used, and there are no extended tasks**
	Error
	The EVENT object is defined in OIL file, but tasks have no references to this event. The task is considered as extended, if any event is referenced.
0775:	**The specified period cannot be represented in the hardware. Period <period value> has been calculated**
	Warning
	If the Period specified by user cannot be represented precisely in hardware, SysGen calculates closest period to it.
0800:	**The number of priorities exceeds <number>**
	Error
	Number of defined Task objects exceeds OS capability.
0801:	**The number of resources exceeds**
	Error
	Number of RESOURCE objects defined in system is restricted by . (including RES_SCHEDULER).
0802:	**The number of alarms exceeds**
	Error
	Number of ALARM objects defined in system is restricted by
0804:	**The number of counters exceeds**
	Error
	Number of COUNTER objects defined in system is restricted by
0805:	**The alarm time of an autostarted alarm can not be more than the maximum allowed value of the counter referenced from that alarm**
	Error
	The value of the AUTOSTART/ALARMTIME attribute of the ALARM object shall be less than the value of the MAXALLOWEDVALUE attribute of the COUNTER object.

Table continues on the next page...

0806:	**The cycle time for an autostarted alarm shall be not less than the minimum cycle and less than the maximum allowed value of the counter referenced from that alarm** *Error* The value of the AUTOSTART/CYCLETIME attribute of the ALARM object shall be more than the value of the MINCYCLE attribute and less than the value of the MAXALLOWEDVALUE attribute of the COUNTER object.
0807:	**The number of application modes should not exceed 8** *Error* Number of APPMODE objects defined in system is restricted by 8.
0808:	**A reference to an application mode shall be defined** *Error* The APPMODE attribute shall be defined for TASK/ALARM with AUTOSTART = TRUE if there are more than one APPMODE defined in the application.
3000:	**<name> - the attribute is not defined in the implementation definition** *Error* This attribute is not specified in the implementation definition.
3001:	**Error while performing I/O: '<filename>'** *Error* The file does not exist, or there are no permissions to open the file or directory.
3002:	**Inconsistency between internal sysgen files detected** *Error* Generation module is inconsistent.
3003:	**OIL Reader error. <error type>** *Error* Syntax or format error in the input file.
3004:	**Unable to open the '<filename>' output file** *Error* The output file cannot be opened if there is no enough free disk space or there is no appropriate permission to create or write the output file.
3005:	**Invalid command line option <option>** *Error* The unknown command option found in command line.
3006:	**Option <option> requires an argument** *Error* The following options should be defined with an argument: -c, -h, -i, -o, -p -m, -L, -M, -e, -g.
3007:	**Only one input file allowed** *Error* One and only one input file shall be specified on the command line.
3008:	**An input file shall be defined** *Error* The input OIL file cannot be found if it does not exist or there are no permissions to open the file or directory.
3009:	**Object type '<name>' is not defined in the implementation definition**

Table continues on the next page...

	Error Only standard objects defined by OIL specification shall be used in application definition. New object types are not allowed.
3010:	**<name> is already defined with the different type: <type>** *Error* The given name has already been used for a system object of the other type.
3011:	**Attribute <name> is already defined** *Error* The attribute with specified name has been already defined.
3012:	**Unknown reference type <name>** *Error* This error may arise only if the OS implementation file is corrupted.
3013:	**An invalid reference to object named <name> of <type> type** *Error* The referenced object with specified name is not found.
3014:	**Attribute <name> shall be defined** *Error* The standard parameters and attributes with NO_DEFAULT specifier defined in implementation definition shall be defined in the application definition (except references).
3015:	**<ObjectType> <Name> shall belong to exactly one application** *Error* Each OS object shall belong to exactly one OS-Application
3017:	**No timing protection is allowed in SC1 and SC3** *Warning* The Timing protection is used only in SC2 and SC4 classes.
3018:	**No protection hooks are allowed in SC1** *Error* The Protection Hook can not be used in SC1.
3020:	**Attribute '<name>' has SUB-CONTAINERS with SHORT-NAME duplicates** *Error* The attribute contains sub-attribute with duplicate name.
3021:	**The attribute value <value> is out of range** *Error* The attribute value does no fit in the allowed range.
3022:	**The attribute value '<value>' does not match the expected type** *Error* The value does not correspond to attribute type.
3023:	**A stack size shall be defined in <object>** *Error* The STACKSIZE shall be defined for each ISR and Task object in the SC2..SC4 classes and for ISRs and Extended Tasks in SC1, ECC1.

Table continues on the next page...

3024:	**UnsatisfiedLinkError : <xxxx>**
	Error
	Java Runtime environment error. Possibly FlexLM DLL (lmgr8c.dll) can not be found - check OS installations
3025:	**The alarm callback is allowed in SC1 only**
	Error
	ALARMCALLBACK can not be used in the Scalability classes with protection.
3028:	**The attribute value '<value>' is out of enum range**
	Error
	The attribute value is wrong.
3100:	**Timer <name> is not supported**
	Error
	This error may occur only if the OS implementation file is corrupted.
3101:	**The value of <name> in all the timers shall be equal to each other**
	Error
	GlobalFTMPrescaler and Prescaler attributes shall be equal in all timers.
3102:	**Not enough hardware parameters are defined to calculate the period. Missed: <parameters> for <objects>**
	Error
	The <parameter> shall be defined
3104:	**The period and all the hardware timer parameters are not allowed together**
	Error
	Timer Period shall have one definition: explicitly or via HW parameters.
3108:	**The resource lock cannot refer to an internal resource**
	Error
	Timing Protection can not be defined for Internal Resource.
3110:	**The maximum resource lock time cannot be 0**
	Error
	Locking time can not be set to 0.
3111:	**The counter type mismatch**
	Error
	HARDWARE counter is assigned to OS Timer defined as "SWCOUNTER" or vise versa
3112:	**Counter <name> cannot be a hardware counter if it is not used by a hardware timer**
	Error
	If the Counter type is set to HARDWARE then this counter shall be assigned to one of System Timers.
3116:	**The attribute value shall not be greater than the schedule table's length**
	Error
	Any of numeric attributes of ScheduleTable can not be greater then its period.
3117:	**The referenced counter shall be a software counter**
	Error
	Counter assigned into "ACTION = INCREMENTCOUNTER" shall have the TYPE = SOFTWARE.
3118:	**The alarm is driven (directly or indirectly) with a counter incremented by this alarm**

Table continues on the next page...

	Warning Cycling of Counters via Alarms may lead to infinite loop.
3119:	**The application that has access to <object> <name>, does not have access to task <name>** *Error* <object> is ALARM, MESSAGE or SCHEDULETABLE. The object belongs to the application and refers to the task. The task shall belong to this application or have this application in its ACCESSING_APPLICATION list.
3129:	**The number of tasks exceeds 64** *Error* Number of defined Task exceeds OS capability.
3130:	**The linker configuration file is invalid. A System comment is not found:** < *expected comments printed here* > Error
3131:	**Timing protection cannot be requested for Category 1 ISRs** *Error* Timing Protection is not applicable for ISRs of category 1.
3134:	**The linker configuration file is not set** *Error* Linker configuration file is required as SysGen input for SC3, SC4 classes.
3137:	**<name> is defined but not used** *Warning* Attribute <name> is not required for the current OS configuration.
3142:	**<ISRname> cannot be assigned to IRQ number <number> because it is already used** *Error* This error is generated if two ISRs (or OS Timer) are configured to use the same IRQ channel
3145:	**The maximum interrupt lock time value for all interrupts is ignored** *Error* NXP AUTOSAR OS does not support GPT as driver for System Timers due to inconsistency in definition and usage of term "HARDWARE COUNTER" in AUTOSAR OS specification.
3146:	**The duplicate of a constant name** *Error* The name of each TIMECONSTANT shall be unique.
3147:	**The offset value shall be less than the schedule table duration value** *Error* The value of the OFFSET attribute shall be less than the value of the DURATION attribute of the SCHEDULETABLE object if REPEATING = TRUE.
3148:	**At least one task or ISR object shall belong to the application** *Error* The OS does not support OS-Applications without Task/ISR cat 2.
3150:	**Timer <name> is used internally by the OS** *Information* Used Timer names are displayed when switch -T is specified

Table continues on the next page...

3152:	**Attribute Freeze in <name> and <name> shall be equal to each other as their timers have the same hardware** *Error* The Freeze attribute shall have the same value when both Timers (SysTimer and SecondTimer) use the same HW.
3158:	**Each expiry point shall have a unique offset** *Error* Any two EXPIRYPOINTS attributes belonging to the SCHEDULETABLE object shall have different OFFSET attribute values.
3159:	**The offset of the initial expiry point shall be zero, or be not less than the minimum cycle and less or equal to the maximum allowed value of the counter referenced from that schedule table** *Error* The value of the EXPIRYPOINTS/OFFSET attribute of the SCHEDULETABLE object shall be zero or be not less than MINCYCLE attribute value and less or equal to MAXALLOWEDVALUE attribute value of the corresponding COUNTER object.
3160:	**The number of ticks between two adjacent expiry points shall be not less than the minimum cycle and less or equal to the maximum allowed value of the counter referenced from that schedule table** *Error* All the EXPIRYPOINTS attributes of a SCHEDULETABLE object may be ordered in accordance with their EXPIRYPOINTS/OFFSET attribute values. The number of ticks (X) between two adjacent expiry points of that ordered sequence is a difference of relevant EXPIRYPOINTS/OFFSET attribute values converted into timer ticks. X shall be not less than MINCYCLE attribute value and less or equal to MAXALLOWEDVALUE attribute value in corresponding COUNTER object.
3162:	**The value of duration shall be equal to one plus the maximum allowed value of the counter referenced from that schedule table** *Error* The value of the DURATION attribute of the SCHEDULETABLE object (with SYNCSTRATEGY = IMPLICIT) shall be equal to the value (1 + MAXALLOWEDVALUE) of the corresponding COUNTER object.
3163:	**The duration value shall be less or equal to one plus the maximum allowed value of the counter referenced from that schedule table** *Error* The value of the DURATION attribute of the SCHEDULETABLE object shall be less or equal to the value (1 + MAXALLOWEDVALUE) of the corresponding COUNTER object.
3164:	**The autostart may be enabled only if an autostart type is ABSOLUTE** *Error* The error message is generated if SYNCH is set to FALSE and AUTOSTART.TYPE is set to SYNCHRON.
3165:	**An adjustable expiry point shall be defined** *Error* If SCHEDULETABLE has SYNCSTRATEGY = EXPLICIT, the ADJUSTABLEEXPPOINT attribute shall be set to TRUE.
3166:	**The value of the offset minus the maximum retard of the expiry point shall be more than the offset of the previous expiry point plus the minimum cycle of the counter referenced from this schedule table** *Error* Difference of the value EXPIRYPOINTS/OFFSET attribute and the value ADJUSTABLEEXPPOINT/MAXRETARD attribute shall be greater than (OFFSET + MINCYCLE), where MINCYCLE is the attribute of the corresponding COUNTER object.

Table continues on the next page...

3168:	**The value of RELOFFSET shall be not equal to zero and shall be not less than the maximum allowed value of the counter referenced from that schedule table, minus an initial offset** *Error* The value of the AUTOSTART/RELOFFSET attribute of the SCHEDULETABLE object shall be less than the difference of the value MAXALLOWEDVALUE attribute of the corresponding COUNTER object and the initial value of EXPIRYPOINTS/OFFSET attribute (the OFFSET attribute value of the first expirypoint in sequence) of the SCHEDULETABLE object.
3169:	**The ABSVALUE shall be not less than the maximum allowed value of the counter referenced from that schedule table** *Error* The value of the AUTOSTART/ABSVALUE attribute of the SCHEDULETABLE object shall be less than the value of the MAXALLOWEDVALUE attribute of the corresponding COUNTER object.
3170:	**The priority of category 1 ISR is equal to <ISRname> priority** *Error* Priority of ISR category 1 shall be greater than priority of any ISR category 2 or System Timers.
3171:	**The timing protection is configured together with Category 1 ISR** *Warning* ISRs category 1 should not be used in protected environment.
3175:	**The number of cores shall be equal to the corresponding OS attribute** *Error* The value of the NUMBER_OF_CORES attribute shall be equal to a number of cores in multicore configuration.
3176:	**ISR shall not use RES_SCHEDULER or resource linked to RES_SCHEDULER** *Error* The ISR object can not have the RESOURCE reference equal to RES_SCHEDULER.
3177:	**The adjustable expiry point is ignored as schedule table <name> is not explicit** *Warning* If the value of the SYNCSTRATEGY attribute of the SCHEDULETABLE object is not EXPLICIT, the value of the ADJUSTABLEEXPPOINT attribute is ignored (shall be set to FALSE).
3179:	**Activation task <name> is configured twice at the expiry point** *Warning* The message is generated when the same task assigned into more than one "ACTION = TASKACTIVATION" inside one entry point of the SCHEDULETABLE object.
3180:	**The non-preemptive task <name> can not have an internal resource: <name>** *Error* The message is generated if the SCHEDULE attribute of the task object is set to NON and the RESOURCE attribute refers to the RESOURCE object with the RESOURCEPROPERTY attribute with INTERNAL value.
3183	**The referenced restart task shall belong to this application** *Error* Reference task in RESTARTTASK attribute for given OS-Application shall belong to the same OS-Application.
3185:	**The number of applications exceeds <number>** *Error* Number of defined OS-Application objects exceeds OS capability.
3191:	**The received pull callback function and the action are configured together in the same IOC**

Table continues on the next page...

	Error Attributes RECEIVER_PULL_CB and ACTION with not NONE value can not be defined for IOC object at the same time.
3192:	**The non-trusted application is ignored in SC1 and SC2** *Warning* OS-Application has the attribute TRUSTED set to FALSE. The value "FALSE" is ignored. Any OS-Application is considered as "tursted" in SC1 and SC2.
3200:	**Different values of the period for the timer and the counter referenced from this timer** *Error* This error is generated if period of system timer is not equals to period of referenced counter. Attributes COUNTER/SECONDSPERTICK and OS/<TargetMCU>/...Timer/...COUNTER/Period may refer to the same physical value. It can be defined one of them, but if both they shall be equal.
3201:	**No spinlocks are allowed in the single core mode** *Error* The spinlocks shall be defined in multi core mode only (NumberOfCores > 1).
3202:	**IRQ number <value> is bound with the Semaphore Unit used by the OS** *Warning* This warning message is generated if user ISRs are configured to use the Semaphore Unit IRQ channel.
3203:	**The value of the final delay shall be not less than the minimum cycle and less or equal to the maximum allowed value of the counter referenced from this schedule table** *Error* This error message is generated if timing parameters of the schedule table and the corresponding counter are inconsistent.
3204:	**The value of the final delay shall be less or equal to the maximum allowed value of the counter referenced from this schedule table** *Error* This error message is generated if timing parameters of the schedule table and the corresponding counter are inconsistent.
3205:	**The value of the delay between the previous expiry point and this expiry point plus the advance maximum of this expiry point shall be less than the maximum allowed value of the counter referenced from this schedule table** *Error* This error message is generated if timing parameters of the schedule table and the corresponding counter are inconsistent.
3206:	**In SC1 and SC2 classes, the trusted function is ignored** *Warning* Memory protection is not allowed in SC1 and SC2, so trusted functions are ignored.
3207:	**The value of the advance maximum of this expiry point shall be less than the duration** *Error* The value of the MAXADVANCE attribute shall be less than the value of the DURATION attribute of the SCHEDULETABLE object if ADJUSTABLEEXPPOINT = TRUE.
3209:	**Duplicated reference to the object** *Error* This error message is generated if the object has two references to the same object.

Table continues on the next page...

3218:	**Function IocSendGroup/IocWriteGroup is not supported. The data index value is ignored** *Warning* DATA_PROPERTY_INDEX is used to define in which order the data is send, e.g. whether IocSendGroup(A,B) or IocSendGroup(B,A) shall be used. It is ignored for the OS.
3219:	**In queued communication, initial data value is ignored** *Warning* INIT_VALUE is used t odefine Initial Value for the data to be transferred on the IOC communication channel. It is ignored in queued communication.
3220:	**IOC is implemented as a macro. The implementation kind value is ignored** *Warning* FUNCTION_IMPLEMENTATION_KIND defines whether this communication is implemented as a macro or as a function. Only "MACRO" value (or " DO_NOT_CARE") is applicable for the OS.
3222:	**The duration value shall be limited by <value> in this scheduletable configuration** *Error* A schedule table that is explicitly synchronised shall have a duration no greater than modulus of the OS counter (MAXALLOWEDVALUE of the corresponding COUNTER object).
3223:	**Cannot write file '<name>' (the absolute path is '<name>')** *Error* Output file can not be written.
3224:	**I/O error. Detailed message: <text>** *Warning* Sysgen can not write/read a file.
3226:	**Cannot find the '<name>' file (the absolute path is '<name>')** *Error* The file does not exist.
3227:	**Cannot read from file '<name>' (the absolute path is '<name>')**
3228:	**Error opening '<name>' for reading (the search path is '<name>')**
3237:	**I/O error on file '<name>' (absolute path '<name>'). Detailed message is: <text>** *Error* The file can not be read.
3229:	**Syntax error '<text>'. Expected <text>** *Error* Error in the input file.
3230:	**Lexer error '<error>'** *Error* Error in the input file.
3231:	**Null XML path passed to a file location constructor** *Error* Error in XML path.
3232:	**Invalid line passed to a file location constructor** *Error*

Table continues on the next page...

	Internal error.
3233:	**Invalid column passed to a file location constructor**
	Error
	Internal error.
3234:	**Unable to initialize the OIL Model's Document**
	Fatal Error
	Internal fatal error.
3235:	**Type '<type>' will be treated as a reference type**
	Warning
	Internal warning.
3238:	**Error validating XML file '<name>' (absolute path '<name>') against XML schema file '<name>' (absolute path '<name>'). Detailed message: <text>**
	Error
	Error during XML file processing.
3239:	**Error opening '<name>' (absolute path '<name>') for writing. Detailed message: <text>**
	Warning
	The file can not be written.
3240:	**Transformer configuration exception. Detailed message: <text>**
	Warning
	Internal warning.
3241:	**XPATH error: <text>**
	Error
	Internal error.
3242:	**Transform exception. Detailed message: <text>**
	Error
	Internal error.
3243:	**File '<name>' is empty and using an empty top level OIL file is not allowed**
	Error
	Error during OIL file processing.
3244:	**OIL generation failed**
	Warning
	Error during OIL file processing.
3245:	**'<name>' should have cardinality less than or equal to 1**
	Error
	Internal error.
3246:	**Encountered a parser configuration exception while attempting to parse file '<name>'. The exception's message was: <text>**
	Error
	Internal error.
3247:	**Exception encountered while parsing file '<name>' (absolute path '<name>'). Exception message was: <text>**

Table continues on the next page...

	Error
	Internal error.
3248:	**Feature <name> is permanent**
	Information
	<name> feature is valid.
3249:	**Feature <name> is not supported by your license**
	Information
	The license for <name> is not provided.
3250:	**Missing attribute '<name>'**
	Error
	An attribute shall be defined.
3251:	**Object '<name>' is duplicated**
	Error
	The same name is defined for different objects.
3252:	**Duplicated attribute '<name>'**
	Error
	The attribute with specified name has been already defined.
3253:	**Wrong attribute value '<value>'**
	Error
	The attribute has incorrect value.
3254:	**Multiplicity check error for '<name>'**
	Error
	Internal error.
3256:	**Error opening '<name>' for reading (no search path provided)**
	Error
	The file can not be found.
3257:	**Unable to obtain the value for the '<name>' environment variable**
	Warning
	The environment variable '<name>' is not defined.
3258:	**Using absolute paths for the angle-bracketed include is not allowed**
	Error
3259:	**Event '<name>' and event '<name>' share a bit**
	Warning
	Sharing of the same bit by event masks inside one task may cause unexpected event notification and/or misinterpretation of the event.
3260:	**Path '<name>' does not point to a file**
	Error
	Incorrect path for a file.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number TR2OSASR4.0R1.0.0
Revision 1.3